IMPROVED SOFTWARE VERIFICATION THROUGH PROGRAM PATH-BASED ANALYSIS

BY

PARTH VIVEK SAGDEO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Advisor:

Assistant Professor Shobha Vasudevan

ABSTRACT

This thesis describes the generation and use of program invariants to improve software reliability. It introduces PRECIS, a technique for automatic invariant generation based on program path guided clustering. The invariants generated by PRECIS can be directly used by programmers for regression testing and improved code documentation. The generated invariants can also be used as part of hardware error detectors, by checking variables key to program output. PREAMBL, a bug localization technique, is introduced as a way of providing increased utility to the generated invariants in diagnosing post-release bugs.

The benefits of these uses of the generated invariants are shown through experiments. The high control-flow coverage of generated invariants is demonstrated for the Siemens benchmark suite, and higher quality is indicated when compared with Daikon, a prior technique. Fault injection experiments show high error detection coverage for several types of manifested errors. Results for PREAMBL show higher scoring for localized paths than previous approaches.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I owe a great thanks to Shobha Vasudevan, for advising me from day one of my Master's work. As the first and only student in our group focusing on software, it was sometimes difficult to know where to get started. Shobha has not only provided that guidance, but also inspiration, moral support, and of course technical advice. More than that, she taught me what it means to do truly great research.

Heartfelt thanks also go to Viraj Athavale, for all his support in developing the PRECIS methodology and the original thesis. Software work such as PRECIS was never part of his main research work or thesis, and yet he always gave our work more effort than I could reasonably expect from anyone.

I would also like to thank Nicholas Ewalt for the implementation work he provided to the PRECIS automation implementation. Despite the complexity inherent to PRECIS, Fjalar, and Valgrind, Nick picked up the technologies quickly and was a great help in developing the instrumentation automation framework used for PRECIS and later the bug localization and error detection work.

Thanks also must go to the rest of my research group. Jayanand Asok Kumar and Lingyi Liu provided their advice and support. David Sheridan explained GoldMine to me step-by-step during the first days of my MS, which seems like an eternity ago. Samuel Hertz, Seyed Nematollah (Adel) Ahmadyan, and Chen-Hsuan Lin, offered their technical expertise, tea (Adel), and support.

Finally, I thank my parents, Anjali and Vivek, and my brother, Nakul. I would not be what I am today without them.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Developing programs that work as intended and keeping them that way after numerous changes are two of the most fundamental goals of software engineering. Despite this, the goal of "bug-free" software remains as elusive as ever. As software grows in both complexity and importance, from managing the national power grid to running implanted pacemakers, the ability to validate that software works as expected is critical.

1.1 Software Development

The modern software development process is divided into roughly four parts: design, implementation, verification, and operation and maintenance. A diagram showing the typical software development life cycle (SDLC) is shown in Figure 1.1.

The design stage involves writing formal or informal specifications for how the software is intended to work. This typically includes a functional requirements document, a high-level design document, and a UI document. Depending on the project, a specification of the behavior in a formal modeling language may also be written during this stage.

The following stage is implementation. In this stage the code which runs the software is written. In the case of new software, the code may be started from scratch. The code may also be modified from an existing codebase, if the product is a new version of existing software. Very basic testing is done during this stage to ensure that existing functionality does not break (regression testing).

Once the software is feature complete the third stage, verification, begins. During this stage the software is tested using a variety of techniques to ensure that it behaves according to the specifications defined in the design stage.

Verification is often the most difficult stage in terms of time and resources; it is common for software to spend more time in verification than the first two stages combined.

The final stage of the software development life cycle is operation and maintenance. This stage involves the end-user's use of the software, as well as maintenance required to keep the software functional in the face of a changing environment and use cases. Regression tests are also very important during maintenance; existing functionality can fail when the execution environment changes, such as after an operating system update.

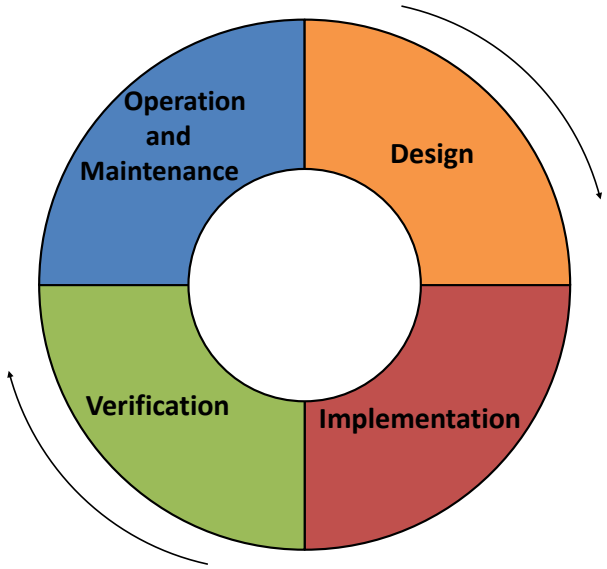


Figure 1.1: A diagram showing a typical software development life cycle (SDLC).

1.2 Software Verification

Of all the stages in the SDLC, verification continues to be the most time-consuming and the most difficult. The research field of software verification is as old as software itself. Roughly speaking, techniques can be divided into two categories. Static techniques involve the analysis of program code, such as in formal verification and source code analysis. Dynamic techniques involve the analysis of a program's runtime state, such as in unit testing, invariant generation, and bug localization.

Formal verification techniques analyze program source code to construct a model of program behavior. This model can be used to verify that the program obeys a previously defined *formal specification*, through a process called model-checking. Formal verification tools have been used to ensure functional correctness [1, 2, 3]. While formal verification tools can provide a provable guarantee that a program adheres to a specification, they suffer from the *path-explosion problem*. The path-explosion problem reflects the fact that as programs get larger, the number of possible paths grows exponentially. Since formal tools must represent each such path in the constructed model, this means the time and memory requirements for analysis quickly become untenable. Thus, even the most advanced formal model-checking tools have considerable scalability issues.

Source code analysis is another form of static analysis. However, instead of using program text to build a model, the program text itself is checked to adhere to various constructs. Industry tools such as Coverity and HP Fortify [4, 5] use source-level analysis to check for code conventions and language-level "rules" such as: "For each `open()` call in a function there must be a corresponding `close()` call." Research tools such as PR-Miner have extended that approach to automatically discover such rules and check the program for them [6, 7, 8]. While source code analysis is considerably lighter-weight and more efficient than model-checking tools, it is limited in its ability to verify program correctness, since it does not construct a model of the program or monitor its execution.

Of the dynamic techniques to software verification, manual testing is perhaps the oldest and most basic. Manual testing involves hand writing test cases for a program, running them on the code, and verifying that the output is correct. Despite this simplicity, there are many different scopes of manual testing. Unit testing has the smallest scope; a single function or class is tested independently. Integration testing involves testing the interface between several units linked together. System testing involves testing the entire program or application. Manual testing has several advantages: it is intuitive and easy to verify whether the program works correctly for the tested cases. However, manual testing requires human effort proportional to the amount of testing required, and it is usually impractical or impossible to perform enough tests to verify the entire input space.

To address these concerns, automated testing is an active area of research.

Traditional automated testing has used fuzzing, that is, random generation of test inputs [9]. More recent work has been done in the field of concolic testing [10, 11, 12, 13]. Concolic testing involves executing a program with a given input, while tracking the path condition for that input. This means that at the end of each execution a concolic tool will have a Boolean expression which evaluates to true if and only if the input follows the same path through all control flow as the given execution. Through multiple executions with inversions of terms in the path condition, test cases can be generated which cover more program functionality than fuzz testing. However, given large programs it can still be impractical or impossible to cover all unique paths, given their exponential growth.

Automatic invariant generation has been an important goal toward increasing confidence of programs for many decades. Recent techniques, such as Daikon and DIDUCE, observe dynamic data from program runs and infer statistically relevant relationships [14, 15, 16]. Some recent techniques use symbolic execution to infer invariants [17, 18].

Inevitably bugs will escape detection until after a program is released. Automated bug localization tools use a dynamic statistical analysis of captured program state to determine where a bug likely is. Recent tools such as CBI and HOLMES capture predicate and path information and calculate a score representing the association of a given predicate or path with the bug [19, 20, 21].

1.3 Software Verification in Industry

Despite the breadth and depth of research, industry practice for the verification of program correctness generally remains limited to manual testing.

Model checking techniques are used extensively in certain safety-critical applications, such as in the aerospace industries and for medical devices, but widespread adoption is very low. This is due to the large cost in time and resources required to maintain formal specifications for each program, and because model checking techniques suffer from computational capacity issues for even moderately sized programs due to their attempt to enumerate all possible executions.

Several source code analysis tools have adoption in industry, such as HP

Fortify and Coverity, though their use is generally limited to enforcing code conventions and identifying the most obvious language misuse, as they are not powerful enough to identify bugs in functionality.

Simple bug localization techniques have been used in production code for decades. For example, Microsoft has integrated a crash reporter into its operating systems since Windows XP [22]. However, modern crash reports tend to be simplistic: they contain information such as the stack trace at program failure, version of software, and hardware. This information is often difficult to use to localize the root cause of a bug.

Invariant generation tools have also not seen widespread adoption in industry, as static tools do not have the scalability required, and dynamic tools often generate spurious, low-quality invariants.

1.4 Approach

This thesis proposes an alternative paradigm of software verification that addresses many of the issues preventing existing research from gaining widespread adoption in the software industry. Specifically, it describes and explores the idea of combining the paradigms of static and dynamic analysis through the capture of statically captured program path structure and dynamically instrumented program state. This white-box, incremental approach to software verification works well for several reasons:

- Software usually exists in an "almost-correct" state. Rarely in the software development process is a program so incorrect that it never produces the correct answer. Manual testing, which is well-practiced by virtually all software developers, makes it easy to determine whether a program works correctly for the common case. It is the uncommon cases, however, where most time and effort is spent to identify and fix bugs. This means that relying a program's typical behavior provides a good insight into how the program should behave.

- Program text implicitly describes expectations. Formal verification of independently defined program specifications is perhaps one of the most ironclad ways of ensuring software works as intended under all circumstances. Yet industry use of formal methods is limited to very specific

applications, such as aerospace software and medical device controllers. In other words, circumstances where the need for reliability justifies the very large time and resource commitment required to maintain formal specifications. Since programs tend to behave correctly in the general case, an alternative to a programmer defining formal specifications is to automatically infer them from the existing program's behavior. While this is necessarily just an approximation to the truly desired behavior of the program, it allows programmers to check behavior against the common case, and to check changes in behavior between versions, with a fraction of the resources required for formal methods.

- Program path information provides an intuitive way to isolate specific program behavior. Programs taken as a whole are complex entities. Model checking approaches to software verification run into scalability issues for even modestly sized programs, due to the path-explosion problem. However, isolating the behavior of each single program path allows an analysis to separate *when* an action is performed (the conditions under which the path is executed), with *what* the action is (the end result of execution of that program path). These analyses can then be recombined to provide an insight into the full behavior of the program.

The thesis focuses on three techniques that use this idea in three subfields of software verification: invariant generation, hardware error detection, and bug localization. For each of these three techniques, the path information gathered improved the dynamic analysis beyond past work. The following subsections introduce the techniques developed in this thesis, and describe their advantages compared to previous work.

These techniques are not a standalone answer to software verification. Rather, they should be part of a verification strategy with techniques such as manual testing and automatic test suite generation. However, when added, they improve the software development process and program reliability, with minimal extra human effort.

1.4.1 Program Path Based Invariant Generation

PRECIS (PREdicate Clustering for Invariant Synthesis) is an invariant generation technique that uses program path information to guide the statistical analysis of dynamic data. PRECIS generates invariants on function-level conditional linear input-output relationships. Function-level relationships tend to be more *actionable*, since developers generally unit test or document programs modularly at function boundaries. Further, conditional relationships allow for the piecemeal analysis of complex functions, since rarely does a single relationship hold for all executions of a function. These relationships tie function inputs to outputs directly, which allows for the capture of an entire function's behavior for a given output using a single expression. Figure 1.2 illustrates the use of PRECIS in the software development lifecycle.

Focusing specifically on linear relationships captures interesting function behaviors. A vast majority of functions return error codes or status values that are constants or simply a linear combination of the inputs. For example, `errno` in Unix is a global variable that is set to fixed constant values when an error is encountered along a path in the function. Similarly, most functions in easy or default cases return values or status codes that are linear functions of the input (often one of the inputs itself) or are constants. Note that while this PRECIS employs linear regression by default, it can be replaced by an arbitrary curve fitting strategy, to capture more complex invariants.

PRECIS generates invariants on a per output (or observable) basis. It instruments the program to capture inputs, target outputs and path conditions. For paths with sufficient trace data, it uses a regression strategy to infer an invariant relating the outputs of a function to the inputs. PRECIS encodes every path as a path predicate word. Capturing path information through dynamic analysis ensures that the generated invariants represent true paths in the program.

One of the primary usage requirements for inferred invariants is succinctness. PRECIS achieves this through a novel clustering of the inferred invariants. Paths are examined as candidates for clustering with *neighboring groups*. Neighboring groups are identified from the path predicate words using a notion of distance between words. Among the neighboring groups, those that can be represented by an inferred invariant that describe the input-output behavior along *all* the paths are clustered. By clustering, we

reduce the number of predicates in the inferred invariants capturing behaviors in succinct ways along those paths. The clustering process in PRECIS enables the topological grouping of a subgraph of paths in a control flow graph. The invariant inference is repeated for each output value of interest in the function.

PRECIS uses static, deterministic knowledge to guide dynamic, statistical methods. This offsets the capacity issues of the static analysis. The domain awareness and structural knowledge from static analysis provides context and focus to the statistical technique. The statistical method we use is *agglomerative clustering* [23] but instead of the standard notion of nearest data points as neighbors, we define a notion of neighborhood that agrees with actual structure of the control flow graph. This synergistic interaction between two solution spaces produces invariants that are simultaneously high-accuracy, high-coverage, and easy for programmers to understand.

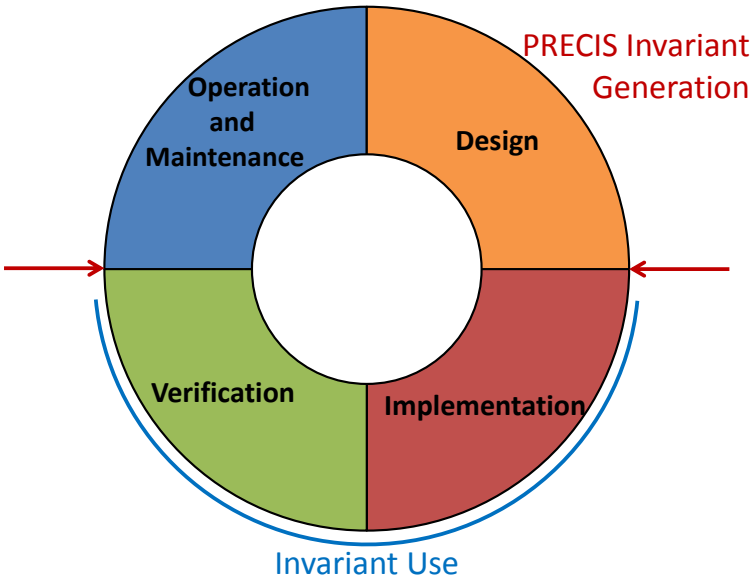


Figure 1.2: A diagram showing a PRECIS's use in the SDLC.

1.4.2 Hardware error detection

For high-reliability systems, the presence of hardware faults and their potential to influence software systems can be an important concern. This thesis describes a technique that utilizes PRECIS for invariant generation to derive path-specific invariants at function boundaries. These invariants check the

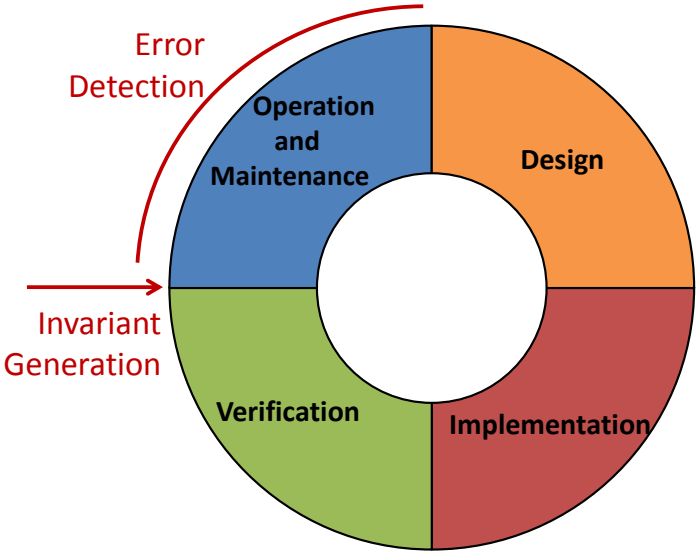


Figure 1.3: A diagram showing the generated error detectors' use in the SLDC.

output values of a function, and so attempt to detect any corruption in state that happened during function execution. This corruption can be caused by a number of reasons: transient errors in memory, defects in functional units, and interference in communication buses, for example.

PRECIS is well-suited to generate application-level error detectors for hardware faults. PRECIS invariants tend to be relatively succinct, and they have a low false positive rate. Further, PRECIS invariants check equality: this means that a corrupted state is much more likely to be caught than with an inequality or range-based invariant.

After a program is verified to be functionally correct, invariants are generated by PRECIS to typify the correct behavior of the program. Each invariant is then integrated into the program source as an assertion that checks to see if a variable holds the expected value. Figure 1.3 shows the usage of the error detectors in the SDLC. If the observed value does not equal the expected value, a data corruption is detected. Once the corruption is identified, corrective steps can be taken by the system administrators, such as restarting the system or replacing faulty hardware.

The detection overage of hardware errors is evaluated through fault injection. Experimental results show high error detection coverage (over 60% of fail-silent violations and 90% of hangs) with a low false positive rate (under

2.5%) for the applications evaluated.

1.4.3 Bug localization

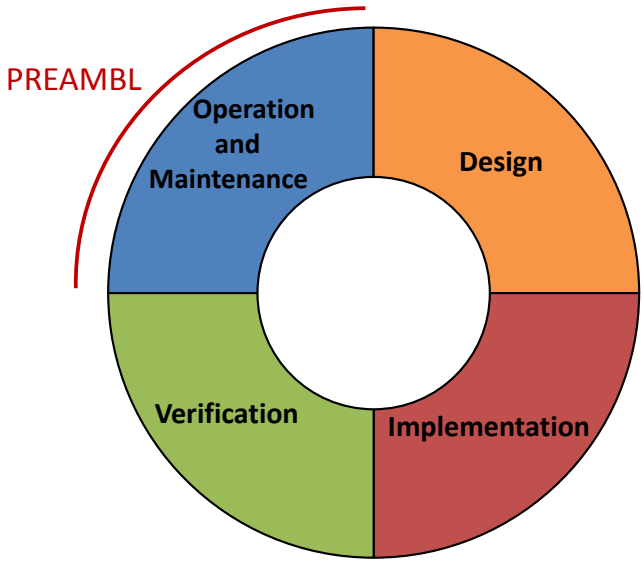


Figure 1.4: A diagram showing a PREAMBL's use in the SDLC.

Invariants are typically removed from shipped production code due to efficiency concerns, but the need to catch and diagnose bugs remains. This thesis also presents PREAMBL (PREdicate Analysis for Multi-phase Bug Location), an integrated approach to invariant generation and statistical debugging. PREAMBL uses PRECIS infrastructure for capturing program predicate information to guide statistical debugging. It then performs a multiprocedural analysis and localize bugs to paths of variable lengths. The use of PREAMBL in the SDLC is shown in Figure 1.4.

The result of PREAMBL is focused, relevant localization. These localized post-deployment bugs on paths can be mapped to pre-release invariants generated along that path. Together, this information provides for a more educated and effective bug fixing experience for the developer. Experimental results demonstrate the efficacy of the use of PRECIS for regression analysis, as well as the ability of PREAMBL to zone in on relevant segments of program paths more efficiently than the state of the art, as can be seen by higher scores when judged by the Importance metric introduced in previous work.

1.5 Contributions

In summary, this thesis describes the following contributions:

- **A novel invariant generation methodology.** PRECIS, an invariant generation methodology based on predicate clustering, is introduced. The three-step process of data generation, predicate clustering, and invariant generation that comprises PRECIS is described. How PRECIS handles complex program constructs is explained.

- **An evaluation of PRECIS' effectiveness in detecting hardware errors:** The application of invariants generated by PRECIS to hardware error detection is described. Hardware and software implementations are outlined, and experimental results for detection coverage are presented.

- **Regression detection using PRECIS.** We present a method for integrating automatically generated invariants into program code. We show a case study illustrating its benefit in identifying regressions, and evaluate its effectiveness in detecting buggy versions of several benchmark applications.

- **A bug localization methodology based on PRECIS.** PREAMBL, a bug localization technique based on program path information, is introduced. Metrics used to score the importance of each path are listed, and a data structure that optimizes the calculation of these metrics is presented.

- **An evaluation of PREAMBL's effectiveness.** PREAMBL is tested by introducing bugs into various applications. PREAMBL's ability to localize the bugs to a particular code path is compared with other techniques using metrics introduced in previous literature.

- **Automated implementations of PRECIS and PREAMBL.** The implementations of PRECIS and PREAMBL are detailed. Important optimizations and lessons learned are described.

1.6 Outline

The rest of this thesis proceeds as follows:

Chapter 2 introduces PRECIS, a program path based invariant generation tool. A running example of a minimum/maximum function describes the three-step process of invariant generation: the data generation, predicate clustering, and invariant generation steps. The chapter describes how PRECIS handles more complicated program constructs, such as loops, functions, and pointers. In the experimental results section, PRECIS is applied to the Siemens benchmarks. The comprehensiveness of the generated invariants is evaluated and the invariants are compared qualitatively with Daikon, a similar technique.

Chapter 3 describes the application of PRECIS to hardware error detection. It describes the motivation behind detecting hardware errors, then describes a software and a hardware approach to using the invariants generated by PRECIS as part of error detectors. The error detectors are evaluated through fault injection experiments, and error detection coverage is measured for various types of manifested errors, such as crashes, fail-silent violations, and hangs.

Chapter 4 describes a holistic software testing technique, PREAMBL, based around the path predicate information used by PRECIS. The technique is twofold; it proposes the use of PRECIS for pre-release software development such as regression testing, while performing a lighter instrumentation processes post-release to localize bugs to specific interprocedural path segments and map them to relevant generated invariants. Two case studies show examples of PREAMBL used to detect bugs. PREAMBL is applied to several benchmark applications to evaluate its effectiveness.

Chapter 5 covers previous work related to the techniques discussed in the thesis. The chapter is divided into three sections, each of which describes the related work of one chapter. Section 5.1 describes prior automatic invariant generation work and compares it with PRECIS. Section 5.2 describes previous work involving automated error detection. Finally, Section 5.3 describes previous bug localization work and compares/contrasts it with PREAMBL.

Chapter 6 describes the implementation work of PRECIS tool. It examines the important design decisions that were made in developing the tool, and implementation details of the various stages of the tool. Standard file

formats used for storing trace data and generated invariants are shown. The fault injection and predicate capture frameworks used in Chapters 3 and 4, respectively, are described.

CHAPTER 2

PROGRAM PATH GUIDED INVARIANT GENERATION

The integration of *assertions* into programs is a time-tested software verification practice [24]. Assertions are statements that describe an *invariant property* of program execution: one that is always true at a given point in program execution. These statements are evaluated at runtime to ensure they always hold. The use of assertions has several advantages as a means of guaranteeing certain behavior during the execution of the program. Other than their obvious benefits of testing variables to make sure they are acceptable, assertions establish a design-by-contract system that formalizes the inputs, intermediaries, and outputs of a system. This helps catch undesirable behavior caused by security exploits and serves to document code, leading to greater programmer understanding.

However, most software written today makes very limited use of assertions; they are typically used to check relatively trivial edge-cases such as null-pointers or out-of-bounds array references, if they are used at all. Although this is unfortunate, it is somewhat understandable; writing good assertions takes developer time away from implementing new features, and is generally given a lower priority.

Automatically inferring invariants from program behavior addresses this issue of program verification being given a lower priority in industry, while providing documentation, correctness, and security benefits. An argument

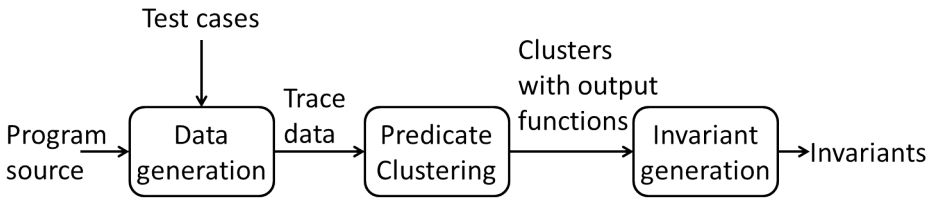


Figure 2.1: PRECIS flow for generating invariants through predicate clustering.

can be made that assertions generated from a program are only as good as the program itself. However, most program behavior is correct by the time extended testing begins, and a significant proportion of bugs are regressions: broken functionality that was working in an earlier version of the code. In addition, assertions can be inspected by an engineer to both verify correctness and ensure intent.

This chapter introduces PRECIS, an invariant generation technique based on program path information. A hallmark of a useful invariant generation tool is that it generates succinct invariants and also generates as few spurious assertions as possible. The distinctive strength of PRECIS can be attributed to the following principles:

- First, PRECIS focuses on invariants that describe the return values of a function as well as side effects to global variables or heap values upon completion of a function execution. These values are referred to as the *outputs* of a function. Writing invariants exclusively on outputs allows for the description of an entire function's behavior while requiring invariants on a relatively small set of variables.

- Second, input-output relationships in a function tend to often be simple (and often linear). Further, confidence in such inferred relationships can be reinforced through these observed input-output relationships recurring over the same path in the function in multiple tests in a test suite.

- Third, PRECIS takes advantage of the path information from the control flow graph of a function to intelligently cluster invariants together.

The goal of this chapter is to show that applying the aforementioned principles generates invariants that are valuable to the developer and succinct in their description. Specifically, Sections 2.1-2.3 describe the steps that comprise PRECIS through a running example of a simple minimum/maximum function. Section 2.4 describes how PRECIS handles more general program constructs, such as loops, internal function calls, arrays, and pointers. Section 2.5 evaluates the technique on the Siemens benchmarks, and compares the generated invariants to Daikon, a similar technique.

```
int min, max; //outputs

void minmax(int a, int b, int c) { //Inputs
   min = a, max = a;
   if (min > b) //p0
      min = b;
   else if (max < b ) //p1
      max = b;
   if (min > c) //p2
      min = c;
   else if (max < c) //p3
      max = c;
}
```

Figure 2.2: An example program to find the minimum and maximum of three integers. `a`, `b`, `c` are the inputs and `min`, `max` are the outputs. Predicates `p0`,`p1`,`p2`, and `p3` are defined for each of the branch conditions.

PRECIS Methodology Overview

Figure 2.1 illustrates the flow of PRECIS to generate invariants. It consists of three main steps: instrumenting the target function to generate the required dynamic trace data for the function, applying our method of path condition (predicate) clustering on this data, and, finally, generating invariants based on the identified clusters.

We discuss each of these steps in detail with the help of the running example shown in Figure 2.2, which consists of a function to find the minimum and maximum of three integers, `a`, `b`, and `c`. We then discuss how we handle more complex program constructs such as loops, pointers, arrays, and function calls.

2.1 Data Generation

The first step in PRECIS flow is to instrument each target function in the program with its inputs, path conditions, and outputs that are relevant to record as part of the function's trace data. This instrumentation is easily automated for any function whose source code is available, since we currently track all possible inputs, predicates (path conditions), and outputs of a function.

We define inputs to be the data on which the outputs of a function directly depend. We consider the function arguments, the results of internal function calls, global variables, and arbitrary heap accesses. In Figure 2.2, the inputs are simply variables `a`, `b`, and `c`. We define outputs to refer to the after-effects of a function. Function return values are the most obvious, but we also consider changes to global variables and persistent heap modifications, since these are often considered the true effects of a particular function. In the example, the outputs are `min` and `max`, both global variables. Finally, *predicates* are the conditional statements which control the path through the program, also referred to in literature as *path conditions*. In other words, knowing the values of all predicates is equivalent to knowing the path taken in the function. Loops are handled in a different manner as described later in Section 2.4. For the example program, we consider: (`min` > `b`), (`max` < `b`), (`min` > `c`), and (`max` < `c`) as predicates, which we will refer to as `p0`, `p1`, `p2`, and `p3`, respectively. Since we want to capture the behavior of the target function, we record the specific value each input, output, and predicate takes for each run through the function. Inputs and outputs of basic atomic types are converted to a numerical floating point representation before recording their values so that they can be used for linear regression, as described in Section 2.2. Pointers, arrays, and composite data types are handled differently, as shown later in Section 2.4.

When the program is exercised using a test case, the instrumented code captures the input, output, and predicate values along a particular program path. This path captures a behavior of a single path in each of the functions in the program. Using a battery of tests, we get more data points for inputs, outputs, and predicates. Further, a good test suite can achieve a greater coverage of paths in the program that are exercised by these tests. Ideally, a good test suite exercises all interesting function behaviors, exhibiting high code coverage. In our example, the test suite may simply be 100 sets of random values for variables `a`, `b`, and `c`. The end result will be a set of *behavior tuples* of the form <input_values,predicate_values,output_values> which describes the behavior of the function over the test suite. Table 2.1 shows a sample trace data for the example program.

Table 2.1: Sample trace data for the example program from Figure 2.2.

Inputs			Predicates				Outputs	
a	b	c	p0	p1	p2	p3	min	max
55	4	27	1	X	0	0	4	55
26	15	12	1	X	0	0	12	26
95	9	88	1	X	0	0	9	95
67	23	65	1	X	0	0	23	67
12	34	45	0	1	0	1	12	45
20	10	30	1	X	0	1	10	30

2.2 Predicate Clustering

The second step, *predicate clustering* lies at the crux of PRECIS invariant generation. We first define some terms that describe the predicates that are key to our inferred invariants. We then describe an algorithm to infer reliable path-based invariants. We then demonstrate how we can use clustering to generate succinct invariants, avoiding unnecessary predicates.

Before we describe the process of predicate clustering we must first define some key entities:

- A *predicate word* is a string containing the values taken for all function predicates during the execution of the function.
- A *predicate group* is the set of <input_values, predicate_values, output_values> behavior tuples in trace data that share the same predicate word.
- An *output function* is a linear expression for a target function output in terms of the inputs.
- A *seed cluster* is the set of behavior tuples in a predicate group that has an associated output function that can be obtained by a *successful* regression.

2.2.1 Basic invariant generation

We first discuss our approach to inferring invariants simply from the observed input and output values along a single path in a function. Here we apply our intuition that program traces with the same program path through a function have the same, often linear, expression for function outputs in terms of the inputs.

Since we evaluate and record the value each predicate takes during runtime, we can encode a specific program path into a *predicate word*. Thus, we consider all behavior tuples in our trace data with the same predicate word taken together as a *predicate group*. We then run a linear least-squares regression on each predicate group, attempting to write each output as a linear combination of the inputs. A regression is said to *succeed* (making it a *successful regression*) if there is a linear function that represents an output in terms of the inputs to the function that satisfies *all* the behavior tuples in a predicate group.

For example, consider the program path in which the first `if` statement is taken, thus the second is not evaluated, the third is not taken, and neither is the fourth. This path corresponds to the predicate word $p = 1X00$ (since the predicate `p1` was not evaluated, we represent its value with an X). In our sample trace data, we have four behavior tuples in this predicate group, as shown in Table 2.1. Since we have (at least) one more tuple than we have input variables, we can perform a linear regression to attempt to derive a linear combination of the inputs that describes each output. For `min`, we find that the coefficients to the linear combination are [0,1,0,0]; in other words, `min = b`.

The seed cluster is computed for each predicate group with enough behavior tuples to perform a regression. The result is a set of *seed clusters*, each representing a single predicate group with some function for the output as a linear combination of the inputs. We will refer to this function as the *output function* of that cluster. At this stage, if there are multiple seed clusters with the same output function, they are merged into a single seed cluster containing all relevant behavior tuples.

Algorithm 1 Generate Seed Clusters

Require: A set *predicate_groups*, containing all predicate groups in trace data
Ensure: A set *seed_clusters*, containing all seed clusters

```
{Generate seed clusters}
seed_clusters := {}
for all predicate_group in predicate_groups do
  (coeff, error) := LinReg(predicate_group)
  if error = 0 then
    {Add new seed cluster}
    new_cluster := (coeff, predicate_group)
    seed_clusters := seed_clusters ∪ new_cluster
  end if
end for
```

2.2.2 Predicate clustering

After the generation of seed clusters, there can still be cases where there are insufficient behavior tuples in our predicate group to generalize its behavior. To help address this case, we introduce the concept of distance between predicate words. We define the distance as an analogue of the Hamming distance [23]. The distance between two predicate words is equal to the number of predicate values by which they differ. Distance is a useful concept in relation to predicates because it helps to quantify the similarity/dissimilarity between two paths. It is our experience that program paths that are almost the same, but only differ in a few branches, often share the same behavior for a particular output variable. Thus a pair of predicate groups with a low distance is much more likely to have the same output function.

We use this heuristic to identify the clusters which most likely share the same behavior as predicate groups without enough tuples to independently derive an output function. We attempt to merge such groups with the seed clusters in single-link agglomerative fashion [23] starting from the least-distance clusters. We call this process *predicate clustering*. In Table 2.1, note that the predicate group $1X01$ only has one tuple, which is clearly not enough to perform a linear regression. However, since it is only 1 distance away from $1X00$, which already has a derived output function for `min`, we attempt to merge the tuples.

A merge is said to be *successful* if a linear regression on the combination of the existing cluster and the new predicate group is *successful*. In other words, a merge is successful if the output function derived for the existing

Algorithm 2 Predicate Clustering

Require: A set *pred_groups*, containing all predicate groups in trace data besides those that have been converted into seed clusters, and a set of *seed_clusters* as generated by Algorithm 1.

Ensure: A set *pred_clusters*, containing all predicate clusters merged according to behavior, a set *pred_groups* containing all predicate groups that could not be merged

```
for all pgroup in pred_groups do
  failed_merges := {}
  repeat
    {Find min dist. cluster whose merge has not failed}
    min_dist := MinDist(pred_clusters,failed_merges)
    {Attempt to merge cluster with predicate group}
    success := attemptMerge(cluster, pgroup)
    if success then
      cluster := Merge(cluster, pgroup)
      pred_groups := pred_groups − pgroup
    else
      failed_merges := failed_merges ∪ (cluster, pgroup)
    end if
  until pred_clusters − failed_merges is empty
      or success
end for
```

cluster matches all tuples for the predicate group. A successful merge creates a cluster that contains both the behavior tuples in the existing cluster as well as those in the predicate group being merged.

Predicate clustering algorithm

Compute the distance between each unclustered predicate group and all the predicate clusters. For a particular predicate group, rank the clusters by distance and test the predicate group with the first cluster for a successful merge. If it succeeds, then merge the clusters. If unsuccessful, the predicate group will be tested against the next-closest cluster. In the example above, predicate groups $1X01$ and $1X00$ are successfully merged and the resulting output function for the cluster is `min = b`. This iterative process continues until all predicate groups that follow the same output function as any cluster are successfully merged.

The end result after predicate clustering is a set of clusters, each likely containing multiple predicate groups, each of which expresses a different *out-*

put function of the program. Certain predicate groups may fail regression, either because the output is a nonlinear function of the inputs, or because it could not be merged with a seed cluster and the cluster does not have enough tuples for an independent regression. These predicate groups do not belong to any cluster, and thus no invariants will be generated to describe their behavior.

2.3 Invariant Generation

In the final step of PRECIS flow, we generate invariants for clusters which have output functions. Since by the end of clustering all predicate groups present in the trace data that share the same output function are merged, this gives us a complete summary of the conditions under which a particular output behavior of the function occurs.

For example, the following invariant is generated for the sample trace data from Table 2.1:

$(p \in \{1X01, 1X00\}) \Rightarrow$ (`min = b`).

The antecedent of the invariant, $p \in \{1X01, 1X00\}$, describes the set of predicate words that the invariant covers. If the program path taken during a particular function execution corresponds the predicate word $1X01$ or $1X00$, then the consequent of the invariant specifies that the output `min` must take on the value of the input `b`. If the predicate word is neither of the two, the invariant does not put any constraint on the output value.

If we convert these invariants back into a symbolic form corresponding to function variables, we often see that our method eliminates irrelevant predicates to capture a function's deeper meaning. Consider the predicate word in our example $1X01$. Symbolically, it refers to program runs where:

$$\#4(min > b) \wedge \neg\#8(min > c) \wedge \#10(max < c)$$

The number in front of a statement indicates the line in program code at which it should be evaluated. Likewise, the predicate word $1X00$ corresponds to program runs where:

$$\#4(min > b) \wedge \neg\#8(min > c) \wedge \neg\#10(max < c)$$

Since both predicates are in the same cluster, we can conclude that the last predicate, corresponding to $\neg\#10(max < c)$ is irrelevant, since its value does not affect the output function. Thus, from the predicate words we can conclude that $\#4(min > b) \wedge \neg\#8(min > c)$ alone is the necessary and sufficient condition for the output `min` to be assigned to `b` at the end of the function. This matches our intuition that if `min` (which is initially `a`) is greater than `b` at line 3, and `min` (which is now the value of `b`) is less than `c` at line 9, then `b` must be the overall minimum of `a`, `b`, and `c`. The same process allows us to make similar conclusions about the cases where `a` and `c` are the minimum and maximum of the three input variables.

2.3.1 Extending the method to multiple output variables

Until now, we have described the predicate clustering process for a single output. In reality, most functions have multiple outputs, each with different behavior that depends on different predicates. Thus, to generate assertions for the complete program, we repeat the previously described process for all outputs. We then apply the invariant generation technique described in step 3 to each set of generated clusters, and combine the resulting assertions generated to express the behavior of the program for all outputs.

2.4 Handling Complex Program Constructs and Types

The example in Figure 2.2 is a simple program only containing `if/else` conditions and assignments to local or global variables of basic types. However, programs usually contain more complex control flow constructs such as `while/for` loops, internal function calls as well as pointer, array and user-defined data types. We briefly describe how we handle each of these in the PRECIS flow.

2.4.1 Function calls

Since an internal function call, such as `rand()` in Figure 2.3, represents behavior that we consider outside the scope of invariants covering a target function, we use outputs from the function call as inputs for the purpose of

```
int min, max; //outputs

void max_rand(){
   //function inputs: N/A
   int i = 0;
   while (i < 100) {
      //inputs: i, orig(L1.i), L1.rand().ret
      int rand_num = rand() % 1000;
      if ( rand > max ) //p0
         max = rand_num;
      i++;
      //outputs: L1.i, L1.max
   }
   //loop outputs = inputs: L1.i, L1.max
   max = max + 1;
   //function output: max
}
```

Figure 2.3: A function that uses a loop to generate 100 random numbers between 1 and 1000, selects the largest among them, and returns that number plus one.

invariant generation. The most straightforward output is the return value, which we represent in Figure 2.3 X as `L1.rand().ret`. We also consider pointer values passed as arguments to the called function, since modifications to those can also influence the output of the target function. These outputs are recorded as new target function inputs at the call site.

2.4.2 Loops and functions

Consider Figure 2.3. Our high-level approach to handling such functions is to write an invariant for the desired output variable directly before the loop starts, write another invariant within the scope of the loop that captures the changes in state that can happen during a single iteration, and finally define the value of the variables at the termination of a loop as new inputs to the enclosing block (which can be a function or enclosing loop).

In our example, this first means processing the contents of the loop. We refer to the instrumented inputs in this case as: `L1_pre.i`, `L1.i`, `L1.max`, and `L1.rand().ret`. These variables refer to the value of `i` before the loop starts, the value of `i` at the start of the current iteration, the current maximum

random value seen, and the random number generated by the iteration's call to `rand()`, respectively. Our two outputs in this case are `L1.max` and `L1.i`, because they are the variables that persist across iterations. We have a single predicate `p0`, which is the conditional that checks if the generated random number is bigger than the current `max`.

When we run this block through the PRECIS flow, the result will be invariants that express the relationship between loop iterations. For example, one invariant will indicate that if `L1.rand().ret` is greater than `max`, then `max` is assigned to that value. Another will express the increment-by-one behavior of `i` on each iteration. Together these invariants will attempt to capture the properties that hold across loop iterations.

To generate meaningful invariants for function outputs that depend on operations in the loop, we define new inputs corresponding to each of the outputs of the loop; in this case `L1.post.i` and `L1_post.max`. These new inputs are recorded as part of the larger function instrumentation process immediately after exiting from the loop. We use these as a basis to write invariants for our function outputs. For example, at the end of `max_rand`, `max` is assigned to the largest random number generated in the loop plus one. PRECIS represents that by generating the assertion `max == L1_post.max + 1`.

Our method for handling loops takes inspiration from the formal process of writing loop invariants. While we do not rigorously prove our generated invariants correct and complete, we rely on the same basic premise that if an output is computed correctly until the start of the loop, likewise across iterations, and is handled correctly from the exit of the loop to the end of the program, then the variable is correct overall.

2.4.3 Arrays and pointers

Reading an element of an array is considered similar to a function call in that we define a new input to hold the value read. Reads from a memory location referenced by a pointer are handled similarly.

A write to an array element or to a memory location referenced by a pointer is a persistent memory modification and hence is considered a function output (Section 2.1). However, the value at such a memory location might be

Table 2.2: Summary of benchmark programs and PRECIS invariant coverage.

Program	Lines of Code	Functions	Invariants	Average Invariant Coverage
`gzip`	7477	141	1281	49.9%
`replace`	413	33	48	77.15%
`schedule`	564	24	104	67.25%
`schedule2`	323	22	97	78.0%
`space`	6199	165	1583	91.4%
`totinfo`	565	21	161	84.3%
`BST.java`	168	1	12	73%

changed by another pointer later in the function, which would lead to an incorrect value being recorded as being output. This problem is known as the *aliasing problem*. In order to avoid this, we only store the location in memory of the address that was modified at the point in the program where the write is performed. We read the value at the that address only at the end of the target function to ensure we record the true output value.

2.4.4 User-defined types

For `structs` in `C` programs as well as `classes` in object-oriented languages, each field or member variable of basic types is used to define an input or output appropriately.

2.5 Experimental Results

We evaluate the PRECIS methodology through experiments on the Siemens benchmarks [25]. All experiments were carried out on a machine with Intel Core i5 750 with 16 GB RAM. Time taken for data generation and clustering was under 10 seconds for each experiment.

We use *invariant coverage* to denote the number of program paths covered by an invariant. We evaluate the generated invariants w.r.t. this metric. Table 2.2 summarizes the programs used for experiments along with the average invariant coverage of PRECIS invariants for these programs.

Table 2.3: Summary of results after running PRECIS on functions from `replace` and `schedule`. Columns 3 shows the percentage of covered paths (P) while column 4 shows the number of invariants generated (N). †For loops, the number of paths inside the body of the loop are considered. Total number of paths would be higher than that shown in Column 3.

Function/Loop	**Output variable**	**P(%)**	**N**	**Comments**
`esc`	`result`	100	5	Complete coverage for both outputs, no clustering for `result`
`('replace.c')`	`*i`	100	2	
`while` loop† in `makepat` `('replace.c')`	`lj`	100†	2	Complete coverage and clustering for 4/7 outputs
	`j`	100	2	
	`junk`	100	3	
	`getres`	100	3	
	`done`	100	2	
	`escjunk`	100	2	
	`lastj`	100	2	
`while` loop† in `dodash` `('replace.c')`	`escjunk`	100†	2	Complete coverage and clustering for 3/5 outputs, no clustering for `*j`
	`junk`	100	5	
	`k`	100	2	
	`*i`	0	0	
	`*j`	100	5	
`del_ele` `('schedule.c')`	`d_list`	100	1	Complete coverage for 4/6 outputs, clustering for all outputs
	`d_list->first`	100	2	
	`d_list->last`	100	2	
	`d_list->mem_count`	100	1	
	`d_ele->next->prev`	50	1	
	`a_ele->prev->next`	50	1	

2.5.1 Siemens benchmarks

Table 2.3 summarizes the results obtained by running PRECIS on some functions from Siemens benchmarks '`replace.c`' and '`schedule.c`'. Results for remaining functions are not shown here due to space constraints.

In the majority of cases, the invariants are found to cover all paths, i.e., completely describe the output behavior in terms of inputs. For some outputs such as `d_ele->next->prev`, the output is only defined in some of the paths. Therefore the generated invariants do not have complete invariant coverage.

Table 2.4 shows the invariants generated for output `*i` in the `esc` function from `replace.c`. At a high level, `esc` processes escape sequences. It takes two arguments: a string, `char* s`, and an index passed by reference, `int* i`. If the character at the index `*i` reflects the start of an escape sequence

Table 2.4: Example invariants that PRECIS generates for output `*i` of the function `esc` from '`replace.c`'.

Output Variable	Invariant
`*i`	`(s[orig(*i)] != ESCAPE) || (s[orig(*i)] == ESCAPE && s[orig(*i) + 1] == ENDSTR) ⇒ (*i == orig(*i))`
	`s[orig(*i)] == ESCAPE && s[orig(*i) + 1] != ENDSTR && s[orig(*i) + 1] != 'n' && s[orig(*i) + 1] == 't') || (s[orig(*i)] == ESCAPE && s[orig(*i) + 1] != ENDSTR && s[orig(*i) + 1] != 'n' && s[orig(*i) + 1] != 't') || s[orig(*i)] == ESCAPE && s[orig(*i) + 1] != ENDSTR && s[orig(*i) + 1] == 'n') ⇒ (*i == orig(*i) + 1)`
`result`	`s[orig(*i)] == ESCAPE && s[orig(*i) + 1] != ENDSTR && s[orig(*i) + 1] != 'n' && s[orig(*i) + 1] == 't') ⇒ (result == TAB)`
	`(s[orig(*i)] != ESCAPE) ⇒ (result == s[orig(*i)])`
	`(s[orig(*i)] == ESCAPE && s[orig(*i) + 1] != ENDSTR && s[orig(*i) + 1] != 'n' && s[orig(*i) + 1] != 't') ⇒ (result == s[orig(*i)+1])`
	`(s[orig(*i)] == ESCAPE && s[orig(*i) + 1] != ENDSTR && s[orig(*i) + 1] == 'n' ) ⇒ (result == NEWLINE)`
	`(s[orig(*i)] == ESCAPE && s[orig(*i) + 1] == ENDSTR) ⇒ (result == ESCAPE)`
`d_ele->next->prev`	`(!(!d_list || !d_ele) && (d_ele->next)) ⇒ (d_ele->next->prev == d_ele->prev)`

(either `\n`, `\t`, or `\0`), then the function parses the sequence. At the end of the function, `*i+1` refers to the character after the just processed character or sequence.

Two invariants are generated for `*i`: one that describes executions in which `*i` does not change, and the other covering executions where `*i` is incremented by one. An example of clustering is the disjunction of the two cases in which `*i` does not change: either the character indexed by `*i` is not an escape character or it is, but the subsequent character is invalid. The same process also allows PRECIS to identify and combine all paths that cause `*i` to increment.

2.5.2 Comparison with Daikon

We now compare the invariants generated by PRECIS and Daikon [14] for Siemens benchmarks '`replace.c`' and '`schedule.c`'.

Since PRECIS invariants express outputs as a function of inputs, we compare them to the function postconditions inferred by Daikon for the same

outputs. It is possible that one PRECIS invariant represents multiple Daikon invariants and vice versa. Therefore we ranked the collection of invariants for each output as a whole. For objective comparison, we define four ranks to evaluate the quality of the invariants:

- **Rank 0:** Spurious invariants that are true in the given trace data but do not hold in the program in general are assigned Rank 0.
- **Rank 1:** These invariants provide useful information about outputs, but do not express the outputs as exact functions of other variables.
- **Rank 2:** Rank 2 invariants express output as an exact function of inputs. However, the functions hold only on a subset of program paths.
- **Rank 3:** These invariants express the output function completely i.e. along all possible paths.

Figure 2.4 shows the percentage of total invariants belonging to each of the described ranks for PRECIS and Daikon. Daikon generates some (13%) spurious invariants, while PRECIS generates none. All the invariants generated by PRECIS belong to Ranks 2 and 3. In some cases, Daikon is found to generalize a PRECIS invariant of Rank 2 or 3 into a Rank 1 invariant.

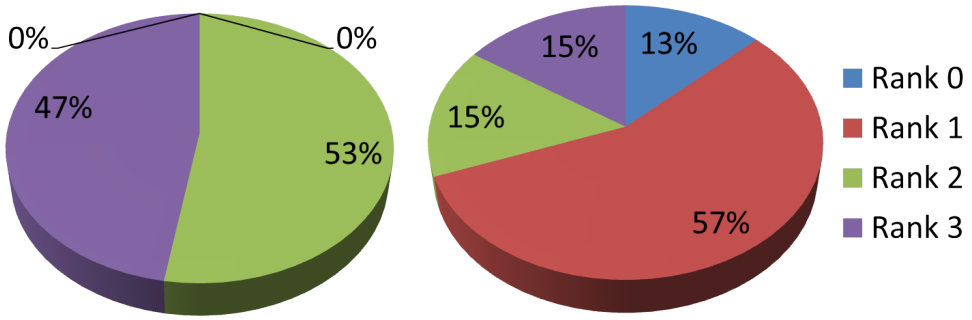


Figure 2.4: Percentage of total invariants belonging to each rank inferred by PRECIS (left) and Daikon (right).

CHAPTER 3

APPLICATION OF INVARIANTS TO HARDWARE ERROR DETECTION

This chapter presents a technique for generating hardware error detectors for C and C++ programs based the invariants generated by PRECIS. The detectors identify data and control errors in an application. There are several possible faults that could cause these errors: cosmic ray strikes, process defects, and mechanical device malfunction are examples. In each case, program state is corrupted from the correct value due to a hardware malfunction. In particular, these detectors identify two manifestations of these faults: data errors and control errors [26]. Data errors cause a divergence in data values from the error-free program. Control errors represent an incorrect path through the control flow graph (CFG) of a program. Detecting these errors allows corrective measures to be taken and reduces the chance of irrecoverable errors in application state.

The rest of the chapter is structured as follows: Section 3.1 discusses the motivation behind detecting hardware errors using invariants. Section 3.2 provides a high-level overview of the way PRECIS invariants are used to construct error detectors. Sections 3.3 and 3.4 provide details on a software and a hardware implementation of the error detectors, respectively. Section 3.5 provides experimental results describing the fault injection methodology and provides data on error coverage, the rate of false positives, overhead, and the relationship between CFG coverage and error detection coverage.

3.1 Motivation

Traditional hardware-based techniques like ECC in memory have limited use in detecting errors that manifest at the software level for several reasons. First, an error that manifests in software can be caused by a malfunction in any of several hardware components, for example: functional units in the

CPU, cache memory, main memory, hard disks, and the buses and links that interconnect them. A single, software-level approach is considerably less complex than integrating error correction schemes for each of these components. Further, a software-based approach allows the selective use of error detectors on only those applications and application subcomponents where reliability is a concern; those for which reliability is not of high importance can run on the same hardware without the overhead of error detectors.

In addition, a software-level approach has the potential to provide much higher error coverage. ECC and other parity-check codes are designed to only detect a certain number of bit-flips from the correct value. In contrast, the error detectors proposed in this thesis check for exact values, and thus have a much higher likelihood of detecting an error.

Replication techniques [27] have also been used for software-level checking of hardware errors. The replication causes an unreasonably high overhead if faithfully implemented for every line of code. In cases where the replication is limited to certain program points, the coverage is consequently low. These replication techniques are also not selective with their detection; they frequently detect errors that are not catastrophic in nature [28]. Static program or model analysis techniques suffer from computational capacity issues due to their attempt to enumerate all possible executions. Dynamic statistical techniques are computationally efficient, and have the benefit of being able to reason statistically about variables whose inherent static relationship is very complex. However, they frequently lack context and produce large volumes of information, including irrelevant or unimportant inferences.

Prior invariant generation based on dynamic techniques such as [29, 14, 15] generate invariants between variables based on certain *templates* by observing program traces. Invariants generated in this manner often include spurious behavior; the derived relationships may hold for the observed data but not the program in general. This is because many statistically correlated, but actually noncausal behaviors are captured. This lack of focus results in incorrect data values as well as incidentally inferred control information.

Static techniques for error detection [1, 3] attempt to infer invariants by analyzing the source code. These techniques generate formally correct invariants. However, due to the path explosion issues in software, such techniques are likely to suffer from computational capacity problems.

3.2 Approach

In this thesis, we use PRECIS (PREdicate Clustering for Invariant Synthesis) [30] to derive the core relationships checked by our detectors. PRECIS is an invariant generation technique that uses program path information to guide the statistical analysis of dynamic data. The generated invariants are on a per output, per path basis. These invariants are then used as error detectors to ensure program state remains correct throughout execution.

To generate invariants with PRECIS, we instrument the program to capture inputs, target outputs and path conditions (which are captured as *predicates*). The program is then run through a training phase, generating trace data representing normal behavior of the program. For paths with sufficient trace data, we use a regression strategy to infer an invariant relating the outputs of a function to the inputs. Similar paths are then clustered together, leading to general, high-coverage invariants. Capturing path information through dynamic analysis ensures that the generated invariants represent true, feasible paths in the program.

We evaluate the generated detectors through fault injection experiments that simulate the effects of transient errors, e.g. flipping the bits, on the data states of applications. The data states, which are the injection targets of fault injection, are all local and global variables used during the execution of the target applications. After each fault is injected to the target variable, the behavior of the target applications will be observed and classified into one of four categories of errors: crashes, hangs, a fail-silent violation (FSV), or not manifested. Crashes include all abnormal program terminations, such as segmentation faults. Hangs are errors which cause a program to never terminate. FSVs represent an incorrect output of the program. An error is not manifested if it does not cause an abnormal termination or an incorrect output, and is not detected.

In the tested applications, our detectors cover an average of 23% of injected crashes. They cover 61% of fail-silent violations and silent data corruptions. For those applications with significant number of hangs, our detectors cover 95%. The performance overhead generated by our technique depends on the extent of checking, and whether the implementation is in software or hardware. On average for software, it is about 100%. We provide a possible hardware implementation to decrease overhead, but do not evaluate its

performance characteristics.

Another important advantage of using PRECIS is the extremely low rate of false positives in the detection (less than 2.5% in the tested applications). Detectors that are very broad in scope, like variable value based, or interval based detectors, suffer from a high false positive rate [29, 14, 15]. In our case, we are able to generate more focused detectors, since we generate invariants for a single program path. This would normally present the problem of "too much focus," causing lower error coverage per detector. However, we circumvent this problem by using clustering, which increases the scope of the invariant from a single concrete path to a cluster of similar paths. Our invariants are able to have a negligible false positive rate without compromising much on scope.

Using our methodology we also study the relationship between the control flow graph coverage of an invariant and corresponding error coverage. The CFG coverage of an invariant corresponds to the percentage of CFG paths that when executed, trigger the invariant. An invariant is triggered when the antecedent condition evaluates to true. We show that invariants with high CFG coverage tend to also have high error coverage. This can be used as a heuristic for the selective inclusion of invariants in the deployed set.

3.3 Software Implementation

This subsection describes the process through which the invariants generated are integrated into the target application code to support runtime error detection. It first describes the process at a high level then illustrates it with an example.

Recall that PRECIS generates invariants on input-output relationships. This means that the function exit points are a natural place to put our checks. Since our checks are of the form "antecedent" implies "consequent," we can represent the checks in two steps. Since the expression in the consequent is only binding if the antecedent is true, then we can put the condition in the antecedent in an `if` statement. In this case, the consequent can be checked through an `assert` statement in the body of the `if`. For readability, we generate a new checking function for each original function in the program, and call it <F>_check, where <F> is the original function's name. The

checking function is called at all exit points in the original function.

```
int min, max; //outputs

void minmax (int a, int b, int c) { //inputs
   min = a, max = a;
   int in0=a, in1=b, in2=c;
   int p0,p1,p2,p3;
   int out0,out1;

   if (p0 = (min > b))
      min = b;
   else if (p1 = (max < b))
      max = b;
   if (p2 = (min > c))
      min = c;
   else if (p3 = (max > c))
      max = c;

   out0 = min, out1 = max;
   minmax_check(in0, in1, in2, p0, p1, p2, p3, out0, out1);
}

void minmax_check(int in0, int in1, int p0, int p1, int p2,
                                        int p3, int out0, int out1){
   if (p0 == 0 && p2 == 0) {
      assert(out0 == in0);
   } else if (p0 == 1 && p2 == 0) {
      assert(out0 == in1);
   } else if (p2 == 1) {
      assert(out0 == in2);
   }
   //Similar structure for out1
}
```

Figure 3.1: An instrumented and self-checking version of `minmax`. The `minmax_check` function serves as the function's error detector. The assertion structure for `out1` is not shown for brevity but is of the same format as `out0`.

To further illustrate this procedure, we continue the `minmax` example introduced in [30]. A self-checking version of `minmax` is shown in Figure 3.1.

During the instrumentation phase, the values of the `in` (PRECIS inputs), `p` (predicates), and `out` (output) variables during each execution are appended to a text file in comma-separated format. The "`in`" variables will be evaluated when they are introduced to the function, the "`p`" variables will be evaluated with their corresponding predicate, and the "`out`" variables will be evaluated at the function exit(s). As the program is run through a test suite or regular execution, this will build training data which can later be used for the predicate clustering process.

Once the trace data is fed into the predicate clustering tool, we can derive three invariants to describe the output `min`:

- $(p0 = 0 \wedge p2 = 0) \Rightarrow (out0 = in0)$
- $(p0 = 1 \wedge p2 = 0) \Rightarrow (out0 = in1)$
- $(p2 = 1) \Rightarrow (out0 = in2)$

To integrate these invariants into the code as assertions, we perform a two-step process. First, we define a function `minmax_check`, which is responsible for verifying that these invariants hold at runtime. One way of representing these checks is an `assert` statement inside an `if` statement. This corresponds intuitively to the format of the invariants generated: the antecedent represents the "`if`" path condition under which the invariant holds, and the consequent is the output value that is being "`assert`ed" to be true. Figure 3.1 shows this integration into a C program for output `out0`, which corresponds logically to the function output `min`.

Second, we insert function calls to the checking function at all function exits. In Figure 2.2, the check call is inserted at the only exit point (at the end of the function).

3.4 Hardware Implementation

The following section provides an alternate implementation of the derived error detectors: one which introduces additional hardware in the form of a coprocessor to provide checking functionality. While we use the software

Table 3.1: An example Instrumentation ROM.

Program Counter [32b]	Predicate [1b]	Predicate Number [4b]
0x6a45b320	1	9
0x3485ba53	0	12

implementation described in Subsection 3.3 for the rest of the thesis, we also propose the hardware implementation as a means of reducing time and space overheads for performance-critical applications.

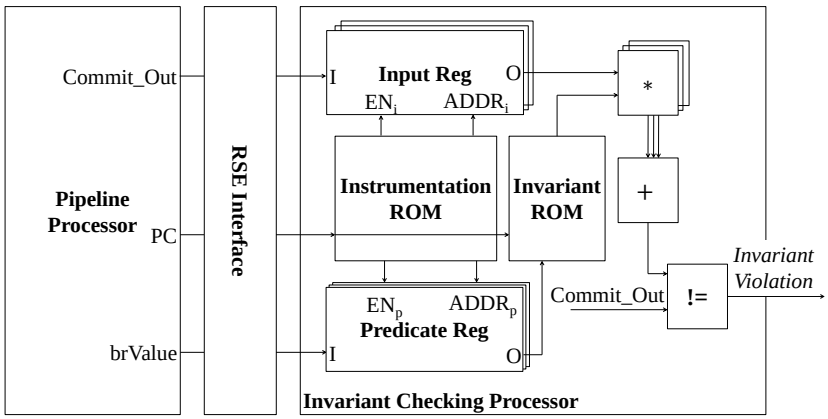


Figure 3.2: A high-level architecture showing the proposed hardware implementation of the error detectors.

In this section we propose a hardware-based architecture of the error detectors in order to reduce overhead in performance-critical systems focused on high reliability. The design is based on the Reliability and Security Engine (RSE) framework described in [31]. This design has not been implemented, but serves as a proof-of-concept for applications where performance is of the highest importance.

Specifically, we propose a invariant checking coprocessor that takes signals from the main processor and uses them to check the invariants in parallel with regular execution. A high-level diagram of the architecture is shown in Figure 3.2.

The proposed architecture is general, but with the shown ROMs loaded with *application-specific* values. During compilation, the PRECIS inputs and predicates are identified. The Program Counter (PC) values representing the location in machine code that a given input or predicate is computed is saved. Each mapping from a PC to the corresponding input or predicate number is stored in the Instrumentation ROM, as shown in Table 3.1.

Table 3.2: An example Invariant ROM.

PC [32b]	Predicate Word [16b]	Input0 Coeff [32b]	...	Input15 Coeff [32b]
0x00126a7bc	0x0000	0	...	1
0x00126a7bc	0x0001	1	...	0
0xff162ab4	0x3F26	1	...	5

During program execution, the instrumentation ROM will allow the invariant checking processor to identify when an input or predicate is being computed. If there is an entry corresponding to the current PC, the write enable (WE) flag will be set for the register file corresponding to that entry (either the Input Reg or Predicate Reg). This will cause the relevant value (Commit_Out for inputs and brValue for predicates) to be saved to the register file. In the example shown in Table 3.1, when the PC is 0x6145b320, the Commit_Out value would be saved to Input Register 9. When the current PC cannot be found in the table, no value is saved to either register file.

While the Instrumentation ROM stores the location of inputs and predicates, the **Invariant ROM** stores the location of outputs and the generated invariants that describe them. Similarly to inputs and predicates, during program compilation the compiler will save the PC at which each output is computed. Invariants are then generated using the standard PRECIS software-level flow. These invariants are then integrated into the Invariant ROM.

An example invariant ROM is shown in Table 3.2. An invariant is evaluated when the current PC corresponds to an output invariant stored in the table, and the predicate word (the contents of the Predicate Register) corresponds to a predicate word entry stored for that output. For example, for the output PC 0x00126a7bc and the predicate word 0x0000, the invariant ($out = in15$) will be evaluated. This is done by multiplying each stored input with the corresponding Input Coefficient. The resulting products are then summed and compared with the observed output value, which is the current value at the Commit_Out wire. If they are not the same, an invariant violation is detected.

The proposed architecture is for the invariant checking phase, and does not support generation. This is because we envision the invariant generation process to be done once per application, before widespread deployment.

Thus, performance is not as critical during that phase. The Instrumentation ROM and Invariant ROM tables can be generated once and loaded onto several systems. Once loaded, the Invariant Checking Processor can be used to provide error detection in real time.

3.4.1 Language support

The instrumentation process described was designed for `C` and `C++`, but it can be easily extended to Object-Oriented Languages such as Java and C#. In both cases, the instrumentation process is considerably simpler because it does not need to handle pointer-based memory modification; rather, objects are treated the same way as `struct`s are treated, and print all of their sub-attributes as additional outputs.

Data Generation in the PRECIS flow simply involves compiling the instrumented source file and running the established test suite on it. The result will be a file in CSV (Comma Separated Value) format which contains the input, predicate, and output values for a particular execution on each row.

3.5 Experimental Results

This section describes the evaluation of the invariant generation and integration methods described earlier in the thesis. We first provide a brief description of the experimental setup, and an overview of the tool which we base our fault injection framework on, Fjalar [32], . We then provide experimental results for *error detection coverage*: of the total number of errors of a given type (crash, fail-silent violation, etc.) induced in the application, the percentage that the generated invariants detect before the error is manifested. This is followed by a discussion of the false positive rate for the detector, and figures for the overhead of the technique. Finally, an empirical relationship between an assertion's control-flow coverage and its fault coverage is explored.

Table 3.3: Breakdown of total number of fault injections and frequency of each error type encountered.

Source File	Total Injections	FSVs	Crashes	Hangs	Not Manifested
`replace.c`	3894	359	1281	8	2246
`schedule.c`	1407	284	93	84	946
`schedule2.c`	2824	683	16	40	2085
`tot_info.c`	2926	760	686	0	1480

3.5.1 Experimental setup

We evaluate our method using four applications from the well-known Siemens benchmarks [25]. Specifically, we use `replace.c`, a program for regular expression string replacement, `tot_info.c`, a statistics calculation program, `schedule.c`, a scheduling program for tasks of different priorities, and `schedule2.c`, a different implementation of the same functionality as `schedule.c`.

For each target application, we first generate invariants using PRECIS, as described in Chapter 2. We use the first 50% of test executions as training data. We then integrate these invariants into the target applications as assertions. To measure error coverage, we injected between 2000 and 5000 single-bit errors, one per run, into the checked and unchecked versions of the application as described in Subsections B and C. A summary of the fault injection campaign is shown in Table 3.3.

3.5.2 Fault injection framework

We developed a custom fault-injection framework to simulate the effect of source-level faults on program execution. It is based on Fjalar [32], a tool developed for binary instrumentation of applications for use with the Daikon invariant generation engine [14]. Fjalar is itself based on Valgrind [33], an instrumentation framework that allows for memory analysis of running programs.

At a high level, the framework is used by generating a tuple (*Program Point, Tag, Variable*) specifying when during program execution a target variable is to be injected. Specifically, *Program Point* can be the start of any basic block in the program, *Tag* is specific to a single execution through

the program point, and *Variable* specifies a random source-level variable for injection. This variable can be a local, global, or formal parameter, and its value can logically be stored either on the stack, heap, or in static memory. However, the fault we simulate could happen elsewhere, such as when the variable is loaded into a register or during computation. When the specified program point is reached, the target variable is corrupted according to a predetermined fault model.

Doing fault injection at the source-level allows us to gain insight into source-level concepts such as which program variables are highly correlated with certain types of failures. In addition, it allows us to easily reproduce the *same* source level fault on different application binaries. This is especially useful in providing a comparison between binaries with and without the error detector integrated.

3.5.3 Fault Model

We use fault injection to simulate the effects of transient errors, represented by single-bit flips on the target variables. The targeted variables can be any stack variables (not necessarily just those on the top frame), globals, or heap data pointed to by these variables. To measure the extent to which the generated assertions detect errors, we first execute the program without fault injection. The frequency of execution of each program point is captured and used to determine a set of random injection points, weighted by their relative frequency of execution, and a random variable valid during each of those points to be corrupted. For each generated tuple (*Program Point, Tag, Variable*), we execute the program twice again, injecting the given fault. The first injection is *unchecked* version – the program is executed without the error detection integrated. The resulting type of error (whether crash, hang, FSV, or not manifested), is then recorded. The second injection is on the *checked* version, with the error detection. If an invariant is triggered during execution, the error is considered detected.

Table 3.4: Generated invariant statistics.

Source File	# of Functions or Loops	# of Outputs	Avg. # of Predicates per Output
`replace.c`	34	44	2.41
`schedule.c`	24	82	1.68
`schedule2.c`	22	90	1.24
`tot_info.c`	21	131	2.00

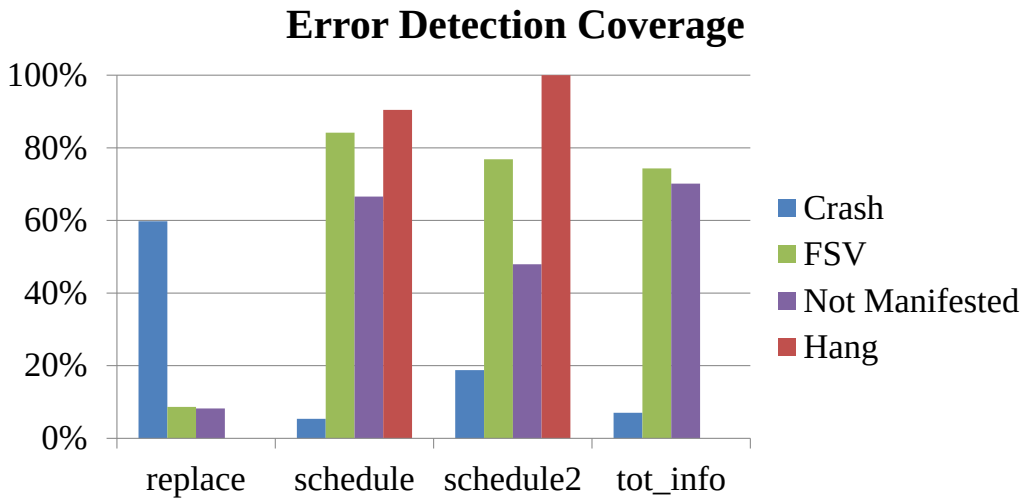


Figure 3.3: Crash, FSV, and total error coverage.

3.5.4 Data Fault Coverage

A primary measure of the effectiveness of our method is the proportion of data errors the generated assertions can detect. To this end, we inject a number of data faults into checked and unchecked versions of each target application and record the types of errors manifested. The goal is to catch the errors with our generated assertions before they can propagate to more serious errors. Specifically, we evaluate error detection coverage for the following classes of errors:

- **Crashes** are the result of faults that cause program execution to end prematurely, such as through a segmentation fault or permissions violation.

- **Hangs** are faults that cause a program to never finish execution, through corruption of a loop condition or some other control flow subversion.
- **Fail-Silent Violations**, or FSVs, results from faults that do not cause a program to exit irregularly, but manifest through a difference in program output from the correct value.
- **Non-manifested faults** are faults that corrupt some program state but which do not ultimately cause program output to be any different from a correct execution.

Statistics for the generated invariants are shown in Table 3.4, and the resulting error coverage is shown in Figure 3.3. The results show that the error coverage for Fail-silent Violations is relatively high – over 60% in three of the four applications tested. We found that this is a strong positive result for the method; FSVs are generally the most insidious type of program misbehavior, since they often are not immediately apparent to the users of a system and thus can accumulate over time. The higher rate of detection of FSVs shows the method's strength in identifying hardware errors that might otherwise go undetected.

In contrast, crash coverage for the tested applications is generally low: for the majority of applications tested the coverage was less than 20%. In practice we found that this was because the corrupted programs often failed even before the first assertion was reached. Although this result is not ideal, it is acceptable; since crashes are often easily identified by a system's users and administrator, corrective measures for errors manifesting in crashes would likely be taken anyway.

We also find that the error detectors caught an average of 48% of faults that ultimately did not manifest in the unchecked version of the program. Even though such faults do not cause errors in execution, identifying them may still be helpful to system administrators. For example, identifying the presence of faults may help identify problematic hardware on a certain system before it fails entirely.

Finally, we found statistically significant results for hang-causing faults for two of the tested applications: `schedule` and `schedule2`. For both applications, the hang coverage was relatively high, over 90%. This indicates that the generated detectors were effective in identifying faults that caused

Table 3.5: Time taken to execute the target applications with and without the error detection integrated.

Source File	Original Time (s)	Self-checking Time (s)	Overhead Factor
`replace.c`	1.155	4.751	4.113
`schedule.c`	7.279	8.222	1.130
`schedule2.c`	9.021	9.932	1.101

Table 3.6: False positive rate for target applications.

Source File	% False Positives
`replace.c`	2.1 %
`schedule.c`	2.0 %
`schedule2.c`	1.8 %
`tot_info.c`	1.4 %

stalled execution. Thus, they may be useful in maintaining system uptime by automatically restarting the system once the error is detected.

In particular, we found that `replace.c` had considerably different results than the rest of the tested applications. Both the relative frequency of manifested errors and the error coverage values did not match those of the other three applications. We theorize this is because of the highly data-intensive nature of `replace`. As a string replacement algorithm, any errors in the state representing text data are likely to manifest as fail-silent violations, since such errors do not generally subvert control flow. However, PRECIS does not typically generate invariants on entire arrays, since it generally only instruments the first element of each array. This explains the lower FSV coverage encountered. We found crash coverage was higher than the average mainly because data corruption was less likely to cause an immediate crash before the error detectors were reached.

3.5.5 False positives

A potential downside to any dynamic technique such as PRECIS is the generation of spurious invariants: properties which may hold for the training set but do not for the program in general. We define the invariant generation *false positive* rate for each application as the number of spurious invariants divided by the total number of generated invariants. The false positive rates

for the target applications are shown in Table 3.6.

3.5.6 Overhead

Since the error detectors are integrated into the application source code, they have overhead in terms of code size, memory usage, and performance. Table 3.5 shows the percentage increase in these metrics for the three of the target applications. Figure 3.4 shows the distribution of this overhead in instrumented, non-instrumented, and checking functions.

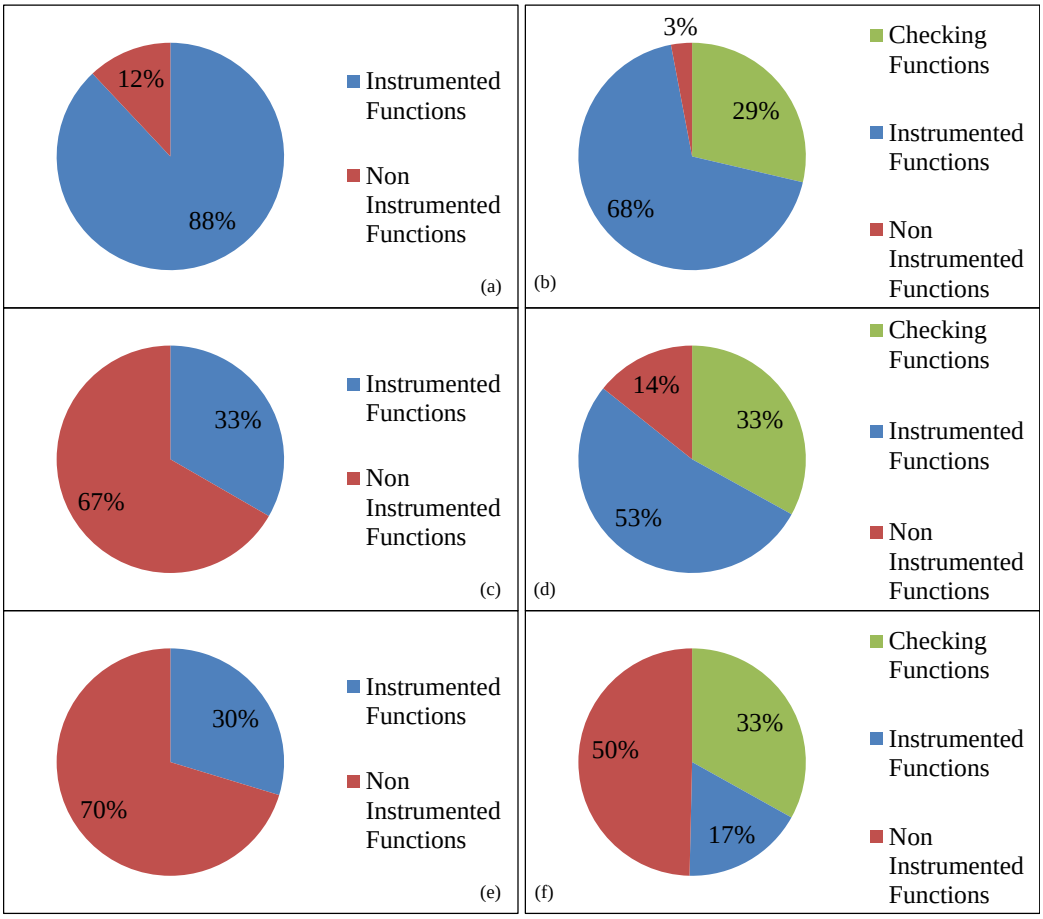


Figure 3.4: The breakdown of exclusive time spent in various functions in original and error-detecting versions of each target application. (a) and (b) show the distribution for `replace`, (c) and (d) for `schedule`, and (e) and (f) show the distribution for `schedule2`.

The CPU overhead varies considerably between applications. Among the applications tested, `replace.c` proved to require the largest increase in exe-

cution time. This is understandable when the location of the detector logic is considered. Since `replace` tends to have complex loop structures with multiple nested loops and control flow inside these loops, the assertion-checking code tends to be more complex and add a significant number of instructions to the most frequently executed regions of code. In contrast, `schedule.c` and `schedule2.c` have very different implementations, but both have relatively simple inner loops. This means the detectors do not add significant overhead.

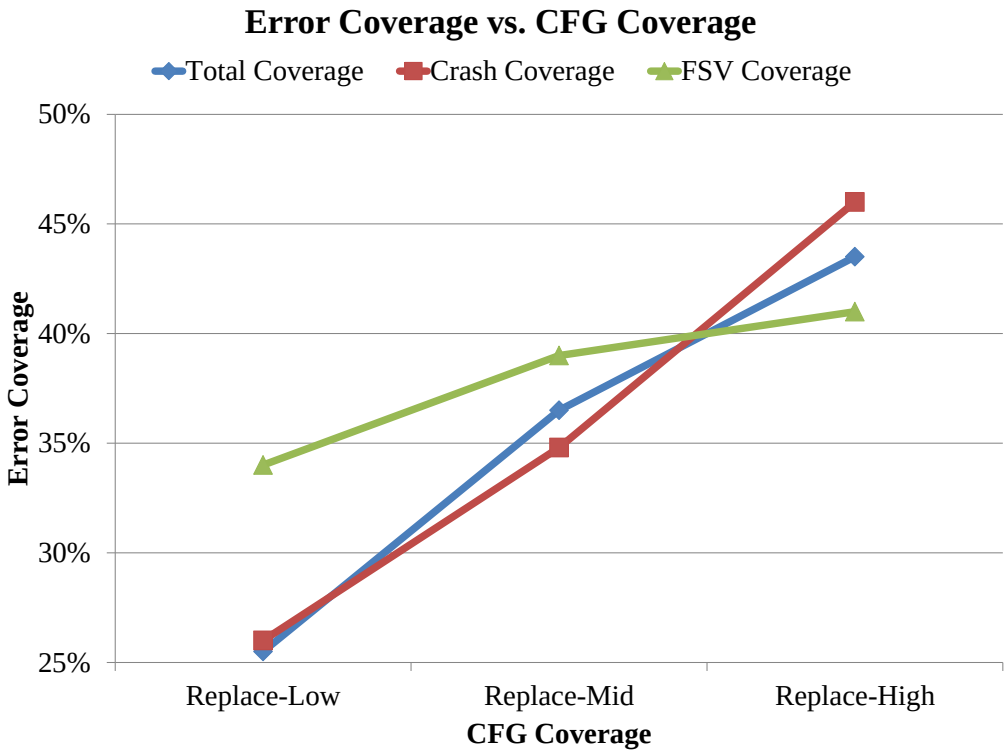


Figure 3.5: Crash, FSV, and total c overage as path coverage is increased.

3.5.7 Path coverage as a means to predict fault coverage

As discussed earlier in the thesis, the assertions generated through our technique are path specific. This means that it is possible to ascertain their **path coverage**: the proportion of program paths in which the invariants take effect. In terms of the instrumented code, this is equivalent to the proportion of program paths for which the `assert` statement will be evaluated for a

particular output. Assertions with high coverage will cover a large proportion of the program paths in a function, and the opposite for low-coverage assertions. If path coverage correlates with fault coverage, a possible advantage to our technique would be to qualitatively judge a detector for fault coverage without time-consuming fault injection experiments. To obtain the degree of correlation for our target applications, we designed an experiment to measure the degree to which fault coverage varies if we control for path coverage.

Specifically, we created three versions of `replace.c`. The first, version is identical to the fully checked version used elsewhere in this section. It has high path coverage: 94.4%. We modified this file to create a medium-coverage version with approximately 50% path coverage, and another with 15% path coverage. We then injected faults into each of them, and recorded the crash coverage, FSV coverage, and total fault coverage for each. The results are shown in Figure 3.5.

Figure 3.5 shows a clear correlation between path coverage and error coverage. With 15% path coverage, `replace-low` detects slightly over 25% of manifested errors. In contrast, `replace-high` detects over 45% of errors. Intuitively, this correlation makes sense. If an assertion checks a higher percentage of paths, it is more likely to catch an error.

3.6 Chapter Summary

This chapter describes a novel technique to derive function-level invariants and integrate them into applications as error detectors. The invariants that comprise the error detectors exhibit high specificity, as seen through the discussion on *tightness*.

Experimental fault injections show that the generated detectors have high error detection coverage for fail-silent violations (FSVs) and hangs. Furthermore, they have a low false positive rate for the applications tested. In addition, the path-specificity of the generated detectors allows for an easy gauge of the expected error detection coverage; experiments show that high path coverage assertions tend to have high error coverage. This is useful in evaluating error detectors without setting up a costly and/or time-consuming fault injection framework.

CHAPTER 4

APPLYING PRECIS FOR BUG LOCALIZATION AND REGRESSION DETECTION

Previous chapters have introduced PRECIS as a technique for automatic invariant generation. PRECIS uses static program path information guided statistical analysis to generate invariants. These invariants are computationally inexpensive and capture meaningful relationships between path predicates and outputs of a function. Section 2.5 showed PRECIS invariants to be demonstrably more valuable than other invariant generation techniques [14, 17, 15] in terms of expressiveness, coverage and scalability.

In this chapter, we present two novel, significant applications for PRECIS invariants during different phases of the software development life cycle – (a) for regression testing and (b) during bug localization. We call the umbrella technology driving the application of PRECIS predicates and invariants as PREAMBL (PREdicate Analysis for Multi-phase Bug Location).

PREAMBL uses the invariants generated by PRECIS to locate bugs in the regression testing phase during development, and the predicates as well as invariants of PRECIS for bug localization in the post deployment phase.

PREAMBL for Regression Testing

Regression testing involves determining if program behavior has changed between versions. The goal is to identify *regressions*, i.e., program functionality that was correct in a previous version and incorrect in a current version. Regression testing is often done by creating a set of test cases and expected outputs which cover the expected behavior of the program [34, 35]. These test cases are then run whenever the program has changed. In practice, however, maintaining and updating a comprehensive test set is highly time and labor intensive [35, 36].

With PREAMBL, we introduce the idea of using invariants for regression testing. Invariants summarize key relationships and functionality of the

program. They can be thought of as detailed specifications of the program intent. While each test can exercise only a single program execution, each invariant can correspond to a set of executions. Hence satisfaction of an invariant in successive versions of a program is a more comprehensive check than traditional testing. Additionally, in regressions, tests would capture input/output equivalence between versions. In contrast, invariants provide a behavioral or functional check. Invariants for regression testing are intended to complement, not replace traditional regression test suites in any way.

However, the tradeoff with using invariants for regression testing is that unlike tests, they do not port easily across versions. We describe a methodology to apply PRECIS invariants for regression testing in Figure 4.1. The principle we use is that invariants from a previous version (n) should be satisfied, i.e. hold true in its next version (n+1). Hence, porting of invariants is only relevant between two successive versions. For such a pair of versions, invariants from version n undergo a check for whether they are admissible in n+1. Each invariant generated by PRECIS expresses a function output as a linear combination of function inputs, predicated on some program paths. Every PRECIS invariant has a reference source code, from which it was inferred. If this reference source code is unchanged in version n+1, the invariant can be used. We can detect this case automatically in PREAMBL. If there are name changes to input/output variables or predicates in a reference source code, but the functionality is intended to be retained, then annotations can be provided by the programmer. Such annotations are very specific and quick in PREAMBL. In the case that the output computation changes in n+1, but the control flow structure remains the same as n, the PRECIS invariant can be used to detect the regression. In the case that the control flow structure of the reference code changes, the invariant will not be admissible for version n+1.

If an admissible invariant fails, then it can mean that a regression has been detected. Alternately, it can mean that a bug from version n was fixed in version n+1. The programmer can inspect the invariant to judge this, and then regenerate it for version n+2 using the corrected n+1 version. If an invariant is not admissible in n+1, it can be regenerated from the changed source code in n+1, for use in n+2. It should be noted that the unsoundness of PRECIS invariants does not affect its application to regression testing, since the faithfulness between two successive versions is being checked, not absolute

correctness of these invariants. We present a case study demonstrating the use of PREAMBL in regression testing in Section 4.4.

PRECIS invariants are very aptly suited for regression testing. PRECIS generates invariants independently at the function level. Hence, it only requires source code for individual functions in the case of limited access. Also, a user has the option to selectively generate invariants only for functions of interest. This is an inherently scalable usage model. In contrast to dynamic techniques that generate inequalities or range based invariants, PRECIS generates strict equalities, that are obviously more specific and accurate. Since invariants are generated at function interfaces, they are more easily transferred between program versions.

PREAMBL for Bug Localization

Bug localization attempts to answer a simple question: given a program misbehavior (bug), which lines of code cause it? Identifying the cause of a bug in program code is often a time-consuming part of debugging, so several automatic techniques have been proposed to give developers information about locations in code highly correlated to the buggy behavior [19, 21, 20].

Assertion checks are typically removed from a program when it is released, due to the overhead caused by those checks. Thus, instead of integrating assertions into release code, PREAMBL performs a lightweight instrumentation of program branch points.

Our vision of PREAMBL for post-deployment bug localization and analysis is shown in Figure 4.2. In the case of a failure, the sequence of branch points leading to the crash is used to decipher the program path that resulted in the crash. We describe a statistical analysis that identifies which program subpaths are most highly correlated with failure. The statistical analysis takes into account the proportion of total failures occur on that path and the percentage of executions through that path that result in failure. Our analysis uses a path frequency tree to identify the *maximum importance subpaths*. Hence, we are able to determine subpaths of varying lengths. Highly correlated subpaths can be presented to developers directly to aid in debugging. This assists in localizing the bug in a smaller, more relevant region, than searching through the entire path during debug. A failing path can be

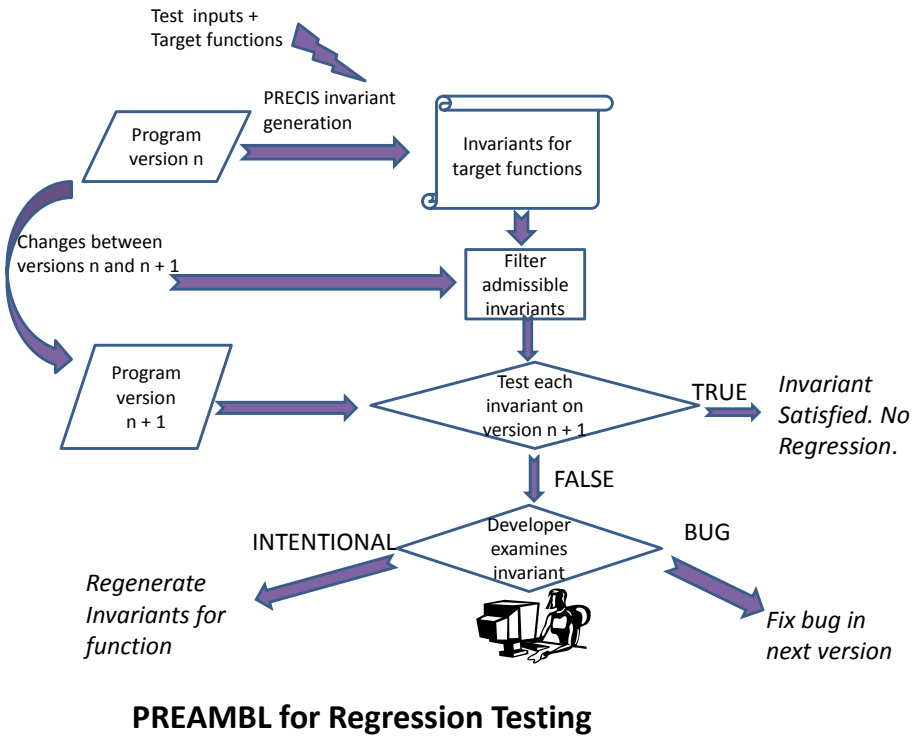


Figure 4.1: PREAMBL: Automated regression testing.

associated with the corresponding PRECIS invariant(s) that summarized it in the pre-deployment phase. From the invariants, the developer will have information about the inputs, outputs and variables defined along that path. This provides more information than is normally available during the debugging phase. Since PRECIS invariants summarize behavior along a path, they can greatly enhance program understanding, especially for maintenance of legacy code or for a programmer dealing with an unfamiliar code base. In addition, examining why a buggy invariant passed or a bug free one failed can reveal insights about the errors in the program.

The scalability of PREAMBL is not a major concern, since it is based on a dynamic, statistical methodology which does not perform any exhaustive analysis. Also, the PRECIS invariant generation is only within a function, making it easy to scale. Hence, our results are presented at a function level for regression testing and bug localization.

To summarize, this chapter makes the following contributions:

- *Regression detection using invariants.* We present a method for integrating automatically generated invariants into program code (Sec-

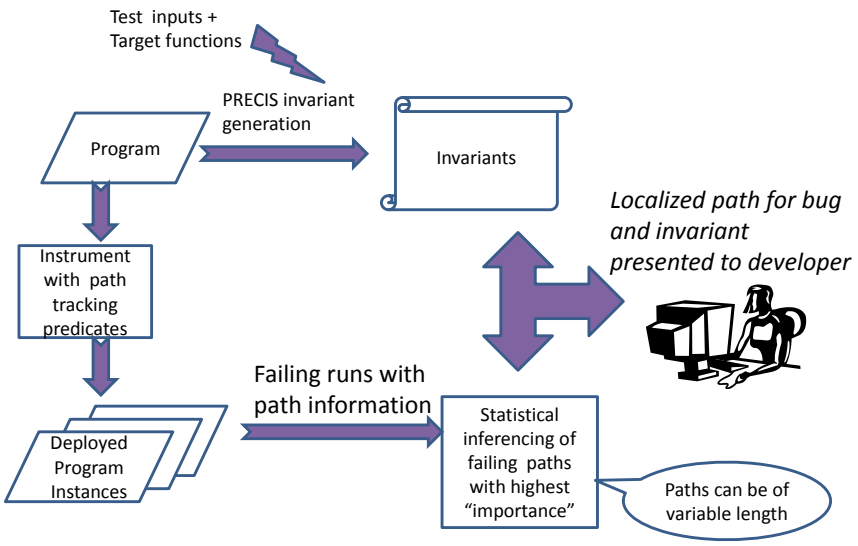


Figure 4.2: PREAMBL enables bug localization and program understanding

tion 4.1). We show a case study illustrating its benefit in identifying regressions (Subsection 4.4.1).

- *Subpath to invariant association.* We present a method for mapping program subpaths identified by bug localization to automatically generated invariants (Section 4.2.4). We show that this information can be used to identify the root cause of a bug more effectively than existing approaches (Subsection 4.4.2).

- *Inter-procedural path-based bug localization.* We present a novel bug localization technique that captures inter-procedural program paths (Section 4.2), and demonstrate its ability to localize bugs that existing path-based techniques cannot (Subsection 4.4.2). We evaluate its effectiveness in localizing bugs in several open-source applications (Section 4.5.4).

- *Extension and automation of PRECIS.* We present the framework for automating the PRECIS tool, as well as its extensions to generic C/C++

constructs (Section 4.1).

4.1 Regression Testing

Automatically generated invariants provide a compact way of representing the behavior of a function or program. Whereas a single test case checks the behavior of a program for a single point in the input space, a PRECIS invariant validates the output for all input values following a particular program path. This means PRECIS invariants can much more compactly express a large set of program behavior. If an invariant generated for an earlier version of the program fails on a new version, then we have detected a regression.

However, unlike test cases, invariants are not inherently portable between program versions. Variables are added and removed, control structure is altered, and code is shifted. Therefore, a mapping has to be made, where possible, between the semantics of the original invariant expression and the those used in the new version. For an invariant to be admissible for a new version of a program, all predicates, inputs, and outputs referenced in that invariant must have a mapping.

In practice, most predicates do not change in a new version of a program. We automatically map such predicates to the corresponding old-version predicates using a string literal match. A map is created between the matching strings that are closest in position in the old and new source files.

At times, however, the expression of a predicate is changed between versions. In this case, PRECIS relies on annotations given by the programmer to describe the intent behind the change. If the change in the predicate expression is not intended to change the meaning of a particular branch point, the programmer can annotate the predicate as being equivalent to a given predicate in an earlier version. One example of this might be prefixing a branch expression with a null pointer check to solve a null dereferencing bug. This will create a mapping between predicates, even if their constituent expressions are different. If the programmer intends to change the behavior of the predicate, then no annotation would be necessary, and invariants using that predicate would be dropped.

A variable name change may need a programmer annotation depending on the type of variable. If the variable is an intermediate variable (not an

input or output as defined by PRECIS), it does not need a mapping because it will not appear directly in generated invariants. For input and output variables, a mapping similar to that for predicates is created. The current implementation PRECIS requires programmers to explicitly denote input or output variables which have changed. However, in practice, this can be done automatically using refactoring tools built into most IDEs. For example Visual Studio and Eclipse both support a variable renaming operation that renames all references to a declared variable. This can be extended to update the mapping maintained by PRECIS.

Overall, we describe the three cases that can result from mapping an invariant, depending on the degree of programmer effort required for them to be admissible in a new version:

- **Case 1:** invariants can be mapped between two versions fully automatically. This is the case if all predicates can be mapped automatically, and no input or output variables change names.
- **Case 2:** invariants require some programmer effort to be valid in the new versions. This may be due to an annotation required for a predicate, or an input or output variable name change.
- **Case 3:** invariants cannot be mapped between versions, due to fundamental code changes. For example, if a predicate or variable referred to in the original invariant is removed altogether from the program, then no valid mapping for the invariant can be created.

We explore the frequency of each of these cases and the resulting coverage in Section 4.5.

4.2 Bug Localization

While invariants are often effective in identifying bugs during software development, they are usually disabled for release to end-users due to their execution overhead. Rather than evaluating full-form invariants as in PRECIS, PREAMBL supports the conditional instrumentation of predicates in released applications to localize bugs. PREAMBL then analyzes the captured program path data to localize bugs to certain subpaths in the application.

Once localized, these bugs are associated with invariants generated by PRECIS. This gives the developer improved context to diagnose and fix bugs. The following section provides detail on the methodology of the approach.

4.2.1 Capturing path data

PREAMBL captures path profiles by instrumenting control flow branch points (`if` statements, `case` statements, etc.) as *predicates*, using the same criteria as PRECIS. Thus, predicates correspond to nodes in the control flow graph, and the value of the predicate represents the direction through the graph. During program execution, the value of each predicate is recorded as the corresponding branch is executed. Taken together for a program run, a sequence of predicates and their values represents the path taken through the program. In this thesis, we will use the format: $(p_0 = v_0) \Rightarrow ... \Rightarrow (p_{n-1} = v_{n-1})$ to indicate a program path that takes the p_0 branch if $v_0 = 1$ and does not if $v_0 = 0$, then evaluates a number of other predicates, and finally evaluates the p_{n-1} branch to be the value v_{n-1}. Each such sequence is marked as *Succeeded* or *Failed*, depending on the status of the given execution.

To quantitatively score paths, PREAMBL uses the algorithm originally proposed by [19] and also used by [21]. In this algorithm, we collect four measurements for each path p:

- $S_o(p)$: The number of successful runs in which the path p was observed.
- $F_o(p)$: The number of failed runs in which the path p was observed.
- $S_e(p)$: The number of successful runs in which the path p was executed.
- $F_e(p)$: The number of failed runs in which the path p was executed.

A path is considered *executed* if all of its constituent predicates were evaluated to their specified value. It is *observed* if the first predicate in the path was evaluated, whether or not the result was the one corresponding to the start of the path. Using these measures and F, the total number of failed executions, we calculate four scores to quantitatively judge to what degree a path is likely the cause of a bug.

$$\text{Sensitivity}(p) = \frac{\log(F_e(p))}{\log(F)}$$

$$\text{Context}(p) = \frac{F_o(p)}{F_o(p) + S_o(p)}$$

$$\text{Increase}(p) = \frac{F_e(p)}{F_e(p) + S_e(p)} - \text{Context}(p)$$

$$\text{Importance}(p) = \frac{2}{\frac{1}{\text{Increase}(p)} + \frac{1}{\text{Sensitivity}(p)}}$$

At a high level, Sensitivity(p) for some path p assigns a score between 0 and 1 depending on the proportion of failures when p was executed relative to the number of total failures. Increase(p) for some path p measures the proportion of executions of p that were associated with a failure, corrected by the proportion of failures on all subpaths of p with one fewer edge.

The goal of PREAMBL is to identify program paths p which are executed in a large proportion of all failures (Sensitivity(p)) and when executed result in failure much often then when they are observed and not executed (Increase(p)). We balance both of these criteria through the Importance metric, by finding paths that maximize the harmonic mean of Sensitivity(p) and Increase(p).

4.2.2 Building the path frequency tree

In order to efficiently generate variable-length paths, PREAMBL must be able to quickly determine $S_o(p)$, $F_o(p)$, $S_e(p)$, and $F_e(p)$ values for any path. To do this, it generates a *path frequency tree* for each observed predicate. Each path frequency tree T_{p_i} is a data structure that holds the number of times a path starting with some predicate p_i is executed in both failed runs, $F_e(p)$, and succeeded runs, $S_e(p)$.

Under the root node for the tree is a node for each observed value for p_i. In general, each node in the tree represents the value of single predicate $(p_k = v_k)$ in a program path. The parent of the node is the predicate value that immediately preceded the node in the path. Each child represents a predicate value executed directly after the current one. As we read each executed path from the path trace file, we increment counters for $S_e(p)$ and $F_e(p)$ on each node as we traverse the tree for the given path. In addition, we increment a tree-level counter for either $S_o(p)$ or $F_o(p)$, depending on whether the execution was a success or failure. The end result for each

predicate frequency tree T_{p_i} after processing all paths is final counts on each node for $S_e(p)$ and $F_e(p)$ for the path starting at p_i and ending on that node, and counts for $S_o(p)$ and $F_o(p)$ for paths starting with p_i.

To lookup these values for a path $(p_x = v_x) \Rightarrow (p_y = v_y) \Rightarrow ... \Rightarrow (p_z = v_z)$, PREAMBL would first begin by finding the path frequency tree for p_x. We would then traverse the tree, starting from the child of the root node $p_x = v_x$, to the child node $p_y = v_y$, and continue traversing the tree until we reached the node corresponding to $p_z = v_z$. The measures for $S_e(p)$ and $F_e(p)$ stored at this node would indicate how many times the path was executed in successful and failed runs, respectively. We can use these values and $S_o(p)$ and $F_o(p)$ stored at the root node to calculate Increase, Sensitivity, and Importance for the path.

Generating the path frequency tree has a computational complexity of $O(n \times l^2)$, where n is the number of paths captured and l is the maximum length of these paths. In practice, PREAMBL takes less than a minute to analyze all tested applications.

4.2.3 Generating high-importance subpaths

Once a path frequency tree is generated for each predicate, we traverse the tree searching for the maximal-Importance paths. Specifically, for each node we calculate Increase(p). Nodes corresponding to paths with Increase $<$ 0 are immediately discarded. Others are added to a priority queue which ranks paths by their Importance score. This process continues until all path frequency trees are fully analyzed. At this point, the top results can be queried from the priority queue. The worst-case efficiency of this process is $O(n \log(n))$, where n is the number of distinct paths captured.

4.2.4 Mapping bugs to invariants

The top-scoring paths will likely tell a developer *where* a bug is in program code but not necessarily *what* is going wrong. Thus, PREAMBL links this localization information with the invariants generated by PRECIS. These invariants summarize what the expected output behavior is, and so allow the developer to understand what code is computing more succinctly than

re-reading the program code. Furthermore, since the generated invariants are path-specific, only those specific to the failing path are shown. If the invariants capture undesirable behavior, having the behavior summarized in the form of an invariant may make it easier for the developer to identify a problem with implementation. If the invariant appears correct, it can be checked by the developer by reproducing failed program runs. Since it is possible that the original test suite did not cover the failing path, re-checking invariants specifically for failing runs may identify which variable is violating expected behavior.

4.2.5 User-driven debugging

Previous bug localization work has usually focused on sending reports after program crashes. Crashes are an important set of bugs, and are well-suited for automatic localization because they are easy to automatically detect. However, many bugs lie in other misbehavior, such as incorrect output. In practice, these types of bugs are more difficult to programmatically detect, so PREAMBL implements *user-driven debugging* to allow program end-users identify misbehaving executions and submit bug reports containing path data labeled for both buggy and successful executions. This allows applications with a large set of motivated but possibly non-technical users to aid in debugging, and allows them to contribute directly to fixing a bug that affects themselves.

In PREAMBL, user-driven debugging works by encapsulating a command-line program in a script. This script first executes the program with path instrumentation enabled. After the program completes, the user is prompted as to whether the execution was successful or buggy. Depending on the user's answer, the path captured for the execution is marked as *succeeded* or *failed*. A number of such labeled paths can be generated by a single user and sent as part of a bug report. Multiple users having the same issue can combine their path traces for further improvement in localization.

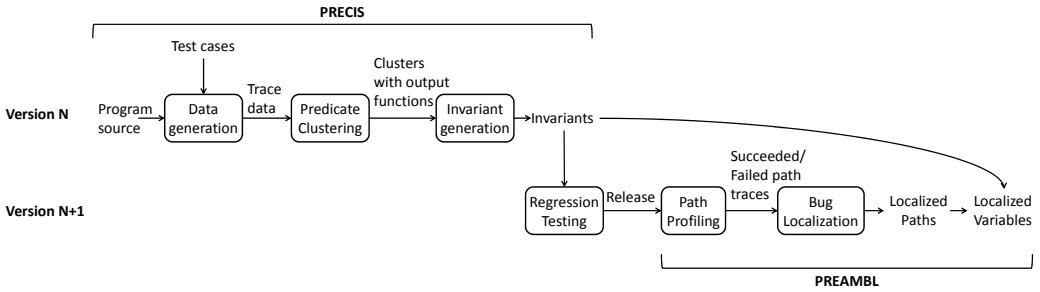


Figure 4.3: The flow of PREAMBL that includes PRECIS as discussed in this thesis. First, PRECIS is used to generate invariants for a version N of the target application. These invariants can be used to regression test the next version, N+1, of the application. On release, PREAMBL is used to localize bugs to high-Importance program paths. These program paths are then associated with the generated invariants to provide the developer with localized variables.

4.3 Implementation

A high-level flow of the methodology discussed in this thesis is shown in Figure 4.3.

We instrument our programs to capture inputs and outputs for PRECIS. The instrumentation environment uses Fjalar [32], a tool based on Valgrind [33], initially developed to support invariant generation in DAIKON. We extend Fjalar to handle pointers, loops, and loops. In order to capture values of predicates, we perform source to source transformations. We consider only path conditionals as predicates in this work. For every if-else and case statement, we substitute the statement content with function calls that log the value of the if-statement. If a path is taken during a program execution, the predicate corresponding to that path thus gets recorded.

An important feature of a truly automated invariant generation engine is automatic integration of the generated invariants into program code. Without this, programmers would be required to examine each generated invariant and manually add a check for it. This is especially true for invariants targeting regressions; a set of invariants comprehensive enough to detect most regressions would necessarily be verbose.

To extend PRECIS to automatically detect regressions, we define a machine-readable format for generated invariants. For every output, the generated invariants specify how the output can be expressed as a linear combination of

inputs. Although we derive linear relationships in our implementation, it is possible to derive quadratic/cubic relationships as well. This is orthogonal to the method itself. The invariant file is read into the Fjalar tool. If during the execution of a program, an invariant fails, the failing invariant is reported, as is the stack trace that caused the failure. This is more information than simply providing binary information about a program crashing or not. An example for the type of information that can be retrieved using PRECIS is shown in Section 4.4.1.

In order to capture the path profiles for bug localization, we use a similar framework for capturing predicates. This is more lightweight, since we do not need to capture inputs and outputs. Instead of executing the application program inside Fjalar, we add a function call from the branches that records if a path was taken or not. Hence, the format of the captured predicates is the same, but the overhead required to run the bug localization is much less than PRECIS.

4.4 Case Studies

We explain the benefits to our approach using two case studies; one for each phase of the software lifecycle: development and release. To show PREAMBL's benefit to software in active development, we provide a case study of a hypothetical scientific computing application. We give an example of a possible regression that might occur during development. We show that PREAMBL is effective in detecting the regression, and provides useful data to the developer as to its root cause. We also show the effectiveness of PREAMBL on post-release software by showing how it localizes a post-release bug and associates it with generated invariants.

4.4.1 Regression Detection

Figure 4.4 shows an implementation of the function `convert_units` in a hypothetical scientific computing application. The intended behavior is simple: the function is passed three arguments, `value`, which is the measured value, `src`, an enumeration which represents the passed value's unit of measurement, and `tgt`, which is the unit of measurement to be converted to.

```
int convert_units(int val, Unit src, Unit tgt){
   int result = val;
   if (src == YD && tgt == IN) { //p0
      result *= 36;
      //Overflow check:
      if (val != result/36) { //p1
         result = -1;
      }
   } else {
      result = -1;
   }
   return result;
}
```

Figure 4.4: A unit conversion function that might be part of a scientific computing application. This first, correct implementation is the basis for automatically generated invariants by PRECIS.

The returned value is the input value converted to the corresponding value in the target unit of measurement. The function has checks in place to catch the case of integer overflow, and to catch unsupported conversions. In both cases, the function returns -1, which is considered an invalid value.

The function `convert_units` is tested thoroughly, and is released in the first version of the application. At this point, PRECIS is run on the application, and the following invariants are captured for `convert_units`:

$$(p0 = 1 \wedge p1 = 0) \Rightarrow (result = 36 \times val)$$

$$(p0 = 1 \wedge p1 = 1) \Rightarrow (result = -1)$$

$$(p0 = 0 \wedge p1 = X) \Rightarrow (result = -1)$$

These invariants represent the derived functional specifications for `convert_units`. Any future changes to the function will be judged against this baseline.

Let us suppose the developer is now working on the next version of the application. The developer wants to make the code more generic. Rather than hard-coding 36 as the conversion ratio between yards and inches, the developer defines it as a preprocessor macro and uses this definition in all relevant conversion code. This new version is shown in Figure 4.5. Unbeknownst to the developer, the ratio `YD_TO_IN` has been defined incorrectly as 35.

```
#define YD_TO_IN 35

int convert_units(int val, Unit src, Unit tgt){
   int result = val;
   if (src == YD && tgt == IN) { //p0
      result *= YD_TO_IN;
      //Overflow check:
      if (val == result/YD_TO_IN) { //p1
         result = -1;
      }
   } else {
      result = -1;
   }
   return result;
}
```

Figure 4.5: A second version of the unit conversion function. In attempting to make the function more generic, the developer has introduced a bug by defining the conversion ratio incorrectly.

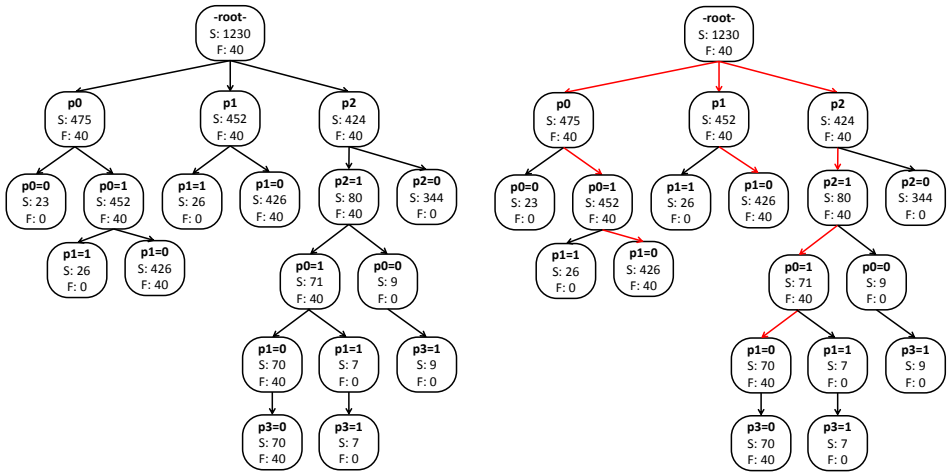
When the developer runs the test suite, the second and third invariants will not trigger, as the behavior under those circumstances has not changed. However, the first invariant will trigger, as when the first `if` statement is taken and the second is not, the result is not $36 \times val$. This quickly alerts the developer to the bug in the code. More importantly, unlike a simple comparison of output values for the application, the failed assertion also tells the developer specifically which function output (in this case `convert_unit`'s `result` is problematic, but under what conditions (a yard to inch conversion with no overflow) it is failing. This helps the developer quickly and effectively isolate the bug.

4.4.2 User-driven bug localization

```
int convert_units(int val, Unit src, Unit tgt){
   int result = val;
   if (src == YD && tgt == IN) { //p0
      result *= YD_TO_IN;
      //Overflow check:
      if (val == result/YD_TO_IN) { //p1
         result = -1;
      }
   } else {
      result = -1;
   }
   return result;
}

void handle_input(){
   ...
   if (task == UNIT_CONVERSION){ //p2
      long in_val = input();
      long out_val =
            convert_units(in_val, in_u, out_u);
      if (result != -1) { //p3
         printf("%ld %s", out_val, str(out_u));
      } else {
         printf("Could not convert\n");
      }
   }
   ...
}
```

Figure 4.6: Two functions, `convert_units` and `handle_input`. One functionality of `handle_input` is to convert a specified value between two units of measurement. However, there is a mismatch in implementation: `handle_input` passes a `long` but `convert_units` expects an `int`. Under certain circumstances, this may cause an overflow where the converted value is not correct.



(a) Example path frequency tree

(b) Path frequency tree with derived paths marked

Path	$S_o(p)$	$F_o(p)$	$S_e(p)$	$F_e(p)$	Inc(p)	Sens(p)	Imp(p)
$(p2 = 1) \Rightarrow (p0 = 1) \Rightarrow (p1 = 0)$	424	40	64	40	0.298	1.000	0.460
$(p2 = 1) \Rightarrow (p0 = 1)$	424	40	71	40	0.274	1.000	0.430
$(p2 = 1)$	424	40	80	40	0.247	1.000	0.396
$(p0 = 1) \Rightarrow (p1 = 0)$	475	40	426	40	0.008	1.000	0.0162
$(p1 = 0)$	452	40	426	40	0.005	1.000	0.009
$(p0 = 1)$	475	40	452	40	0.004	1.000	0.007

(c) Ranking of derived program paths

Figure 4.7: Example of the analysis done by PREAMBL on the code snippet shown in Figure 4.6. (a) shows the constructed path frequency tree for an example test suite. Predicates not explicitly defined in Figure 4.6 are not shown. In (b) the paths that are candidates for localization are colored red. (c) shows the metrics used in calculating the Importance(p) of each candidate path.

We also show how PREAMBL helps in debugging released applications through an example. Consider a user of the same simple open-source scientific computing application discussed in Section 4.4.1, who notices an intermittent bug which is affecting the results. This user does not know how to program and is not at all familiar with the codebase, so diagnosing the issue is difficult. However, the user is interested in helping the developers identify the bug, and so enables PREAMBL path instrumentation on the application.

Once path instrumentation is enabled, the user continues using the application normally. At the end of each execution, the user labels the execution as buggy or normal, depending on whether the bug manifested. By reproducing the bug several times, the user builds two path data files: one containing

buggy path traces and one containing normal path traces. Once the user feels the bug has been reproduced a sufficient number of times, the user submits a bug report, sending both files to the developer for further analysis.

Unknown to the developer at the time, the bug the user is encountering is due to an occasional integer overflow in the same unit conversion function discussed in Section 4.4.1, shown in Figure 4.6. The developer has now fixed the function so it uses the correct conversion ratio. However, a newly constructed function, `handle_input()` passes a `long`, rather than an `int` for the value to be converted. If the value is large enough, and on certain architectures where `long` and `int` are not represented by the same number of bits, this can cause an overflow. In other words, the value held by `val` will not be the same numeric value that was held in `in_val`.

Once the developer receives the bug report, the developer automatically localizes the bug by running PREAMBL on the data. In the following section, we will show how PREAMBL is able to localize the bug to the interaction between the `handle_input` and `convert_units` functions.

Building the path tree

The first step in PREAMBL is to construct a path tree that stores the frequency of each executed path in both successful and failed executions. To do this, PREAMBL reads the path trace files and increments the corresponding path frequency tree nodes for each path it sees. An abridged version of the path tree is shown in Figure 4.7(a).

Once the path frequency tree is constructed, the path derivation process starts. Starting from length-1 paths, PREAMBL traverses the tree for nodes with increasing values for $Increase(p)$ as described in Section 4.2. The paths represented by these nodes, depicted in Figure 4.7(b) and listed in Figure 4.7(c), represent the paths most highly correlated with the bug. In this case, we can see that the highest ranking path is $(p2 = 1) \Rightarrow (p0 = 1) \Rightarrow (p1 = 0)$. This accurately matches the observation that the bug manifests when `convert_units` is called from within the if statement `p2` and a valid non-overflowing conversion is performed.

Table 4.1: A brief description of each application tested.

Application	Lines of Code	Buggy Versions	Description
replace	564	7	Performs pattern matching and substitutions on strings.
schedule	412	9	Schedules tasks based on priority.
schedule2	323	9	Alternate implementation of schedule.
tot_info	565	7	Computes statistics given input data.
space	6199	8	Interprets an Array Definition Language.
gzip	7447	7	Compression / decompression program.

Associating paths with invariants

In order to further help the developer identify *what* is the problem in code rather than simply *where* it is, PREAMBL automatically associates the localized path with invariants previously generated for that path. In this example, an invariant generated for `convert_units()` is:

$$(p0 = 1 \wedge p1 = 0) \Rightarrow (result == 36 \times result).$$

Since this invariant describes the behavior of `result` on the path that the bug was localized to, it is associated with the bug and presented to the developer. Thus, PREAMBL presents both the localized path for the bug and the corresponding expected behavior to the developer. In this case, the developer may immediately notice the type mismatch between the `long` passed to `convert_units` and the `int` type used.

4.5 Experimental Results

The experiments in this thesis are focused on answering the following questions about PREAMBL:

- What is the quality of invariants generated by PRECIS?

- What proportion of regressions can PREAMBL detect?
- How effective is the bug localization technique used in PREAMBL in localizing bugs?

4.5.1 SIR applications

To judge the effectiveness of PREAMBL in localizing bugs, we test it on several applications in the Software-artifact Infrastructure Repository (SIR) [37]: a set of applications from the Siemens suite [25] and space [38]. We use the SIR as a standardized repository of applications and bugs to compare with previous bug detection tools [19, 21]. A brief description of each target application can be found in Table 4.1.

4.5.2 Invariant quality

Table 4.2: A summary of generated invariant quality.

	Invariants	CFG Coverage	FP Rate
replace	48	45.7%	2.1%
schedule	104	82.8%	3.8%
schedule2	97	78.0%	5.5%
totinfo	161	84.3%	5.5%
space	1583	91.4%	7.6%
gzip	1281	49.9%	3.3%

We first examine the quality of the invariants generated by PRECIS for the tested applications. This is done by running one-third of the available test suite on each application to generate trace data, then performing predicate clustering to derive a set of invariants. Table 4.2 summarizes the results. The number of invariants generated grows proportionally with the size of the program, but varies considerably even for similarly sized programs. We found this had to with each programs structure. For example, programs which make heavy use of global variables tend to have a high number of invariants generated, as the number of function outputs is higher.

We define an invariants control-flow graph (CFG) coverage to be the percentage of program paths in which the invariant is triggered. Coverage ranged

considerably between applications, and was highest for control-flow oriented programs where data flow could mostly be described through linear operators.

We also examine the degree to which the generated invariants are spurious. Specifically, we define an invariant false positive rate to be the percentage of invariants generated by PRECIS, on one third of the test set, which are invalidated when running the full test set. PRECIS has a low false positive rate for generated invariants: less than 10

4.5.3 Regression detection

Table 4.3: Metrics for the effectiveness of regression testing.

	Admissible Invariants w/o Annotations	Admissible Invariants w/ Annotations	Detection Coverage
replace	82.4%	85.7%	56.6%
schedule	98.5%	99.9%	55.6%
schedule2	95.75%	96.4%	33.3%
space	100%	100%	62.5%
gzip	100%	100%	57.1%

We next evaluate the usefulness of PREAMBL for regression testing. We do this through the use of buggy program versions included with each of the applications tested. The first step is to generate invariants for the correct version of the program. Then, for each buggy version, we map all invariant predicates, inputs, and outputs to the corresponding value in new version. The average proportion of invariants that are admissible to the new version, with and without user annotation, is shown in the first two columns of Table 4.3.

To evaluate whether the invariants are effective in detecting regressions, we run the test suite while checking the generated invariants. If any invariant evaluates to false during a test run, then we have detected the regression. The resulting regression detection coverage results are shown in Table 4.3.

While the percentage of admissible invariants is high (all tested applications had higher than 80%), there is range between the highest and lowest-scoring applications. In particular, replace had the lowest score without an-

notations, at 82.4%. This was due to a high proportion of control-flow centric bugs encountered. Since changes to control flow require annotation, replace had an especially low proportion of invariants which could be transferred to the next version. In contrast, space had a high proportion of data-flow bugs. Consequently, a high proportion of invariants were admissible to the next version.

Detection also varied considerably between applications. In general, applications with a higher proportion of admissible invariants tended to have higher detection coverage. This is intuitive; the more invariants can be applied to the next program version, the more likely that one will catch the regression. However, the detection coverage is ultimately limited by the manifestation of the bug; if the regression is in a program output that is not captured by any invariant, then the detection coverage will be lower. For example, several regressions in schedule2 involved wrong output from printf statements. Since the output of such statements is not checked by invariants, the detection coverage for schedule2 was much lower than other applications.

Tables 4.4 and 4.5 show a detailed breakdown of two representative applications: replace and schedule. In both applications, the majority of buggy versions have modifications from the original version that fall into either Case 1 or Case 2 as defined in Section 4.1. Both the percentage of admissible invariants and the detection coverage is highest for Case 1 modifications. This is intuitive, as invariants affected by such changes can be automatically mapped between versions. PREAMBL has reduced coverage for Case 2 modifications; during such changes PREAMBL needs programmer annotations to identify equivalent predicates or variables between versions. Finally, Case 3 modifications show the lowest detection percentages. This is because such modifications mean the relevant invariants cannot be admissible, and so the likelihood of detecting a regression through them is very low.

4.5.4 Bug localization

We evaluate the effectiveness of the path-based localization technique used by PREAMBL, and measure what percent of localized bugs can be associated with invariant information. To do this, we measure the percentage of these bugs that can be localized to a specific subpath, and provide aggregate

Table 4.4: Detailed breakdown by case of regression testing effectiveness for replace. For Case 2 and Case 3, the values shown are the percentages of admissible invariants assuming no annotation. x%/y% represent the percent of admissible invariants in the affected function and overall, respectively. The detection coverage value assumes annotation.

replace			
	Case 1	Case 2	Case 3
Proportion of Cases	3/7	3/7	1/7
Admissible Invariants	100%	0%/92.3%	0%/90%
Detection Coverage	66%	55%	0%

Table 4.5: Detailed breakdown by case of regression testing effectiveness for schedule. For Case 2 and Case 3, the values shown are the percentages of admissible invariants assuming no annotation. x%/y% represent the percent of admissible invariants in the affected function and overall, respectively. The detection coverage value assumes annotation.

schedule			
	Case 1	Case 2	Case 3
Proportion of Cases	4/9	4/9	1/9
Admissible Invariants	100%	8.25%/96.8%	66%/99%
Detection Coverage	75%	50%	0%

information on the specificity of these subpaths.

Localization quality

Profiling multi-procedure, variable-length paths allows PREAMBL to better localize bugs to relevant paths. We compare PREAMBL to previously proposed profiling schemes: the fixed-length intraprocedural path used in [21], and a simple single-branch profiling scheme. All three techniques are scored by the average importance score for the top five derived paths. Table 4.6 shows the aggregated results on several benchmarks. In each case, PREAMBL generates higher-scoring results.

We also discuss the average path length for highly ranked paths in both PREAMBL and fixed-path profiling. Figure 4.8 shows the distribution of path lengths for both. On average, PREAMBL tends to generate shorter paths than fixed-path profiling. This indicates that path-length is optimized to not include predicates that are not relevant to the failure. Since it also on

Table 4.6: The average Importance score for the best-scoring paths using PREAMBL, fixed path profiling, and branch profiling. In all tested applications, PREAMBL derives higher-Importance paths.

	space	replace	schedule	tot_info
PREAMBL	0.640	0.502	0.553	0.636
Fixed path	0.537	0.310	0.196	0.585
Branch	0.419	0.138	0.121	0.260

average generates higher Importance paths, shorter paths are preferable for developers: the fewer blocks of code developers need to inspect to identify the bug, the better.

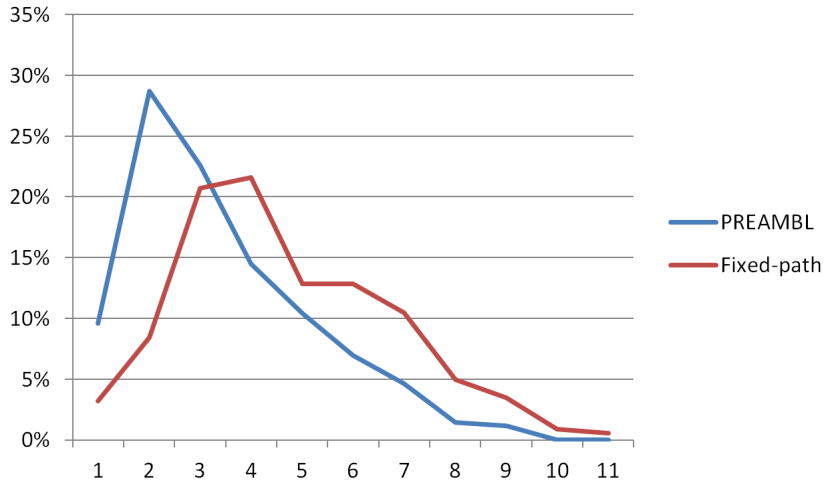


Figure 4.8: The distribution of path length in top-scoring paths for both PREAMBL and a fixed-path profiling technique. A higher proportion of PREAMBL-generated paths are shorter, which indicates more effective localization.

4.6 Chapter Summary

In conclusion, this chapter proposes a technique, PREAMBL, to unite two principal aspects of software reliability and debugging. Instead of dividing the program debugging into water-tight pre-release and post-release phases, we propose a flow where debugging information from these phases can be used synergistically. Using this technology, both the invariant generation and bug localization are made path-centric. Thus, focused information can

be given to the developer for bug identification and repair. The generated invariants may be used beyond regression testing, for both bug association and providing context. Passing invalid invariants and failing valid invariants can be used to help a developer identify logical bugs instantly through the reuse of invariant information from the pre-release phase.

CHAPTER 5

RELATED WORK

This chapter discusses previous work related to the topics discussed in this thesis. The chapter is divided into three sections. Section 5.1 describes previous work in automatic invariant generation and compares it to PRECIS. Section 5.2 describes related work in automatically generating error detectors. Finally, section 5.3 describes related bug localization work and compares it to PREAMBL.

5.1 Invariant Generation

Various approaches for automatic invariant generation for programs have been proposed, involving both dynamic [14, 15] and static [17, 18] approaches. In this section, we discuss key differences between PRECIS and several of these related techniques.

5.1.1 DIDUCE

DIDUCE [15] is an online technique to learn invariants on variables in the program at specific program points. Each invariant is for a single variable and represents the set of values of the variable in the observed executions. Due to the simple form of the invariants, they do not track the dependence of the variable on other variables or program paths. PRECIS invariants, on the other hand, summarize the higher-level relationships between outputs, inputs and program paths.

5.1.2 Daikon

The Daikon tool [14] discovers invariants by analyzing program execution traces. At function entry/exit points and other specified program points, Daikon instantiates a set of invariant templates such as comparison with constant value, linear relationships, ordering, etc. For each template, different combinations of variables including function parameters, return values are tried. Corresponding invariants are reported if they hold for all execution traces.

The main advantage of PRECIS over Daikon is that it employs predicate word information to guide the analysis of execution traces. As a result, PRECIS is able to discover path-specific invariants not inferred by Daikon. Although the `CreateSpinfo` utility in Daikon has some support for using program predicates to generate conditional invariants, it is not able to combine the predicates to encode program paths and therefore does not generate such invariants with complex path conditions.

Daikon includes some complex invariant templates such as membership (e.g. $\texttt{x} \in \{0, 1, 3\}$) and sortedness, not supported by PRECIS currently. However, we believe that our program path based clustering method can be extended to consider additional templates such as these.

5.1.3 DySy

DySy [17] introduces *dynamic symbolic execution* in order to infer function preconditions and postconditions. DySy executes test cases and simultaneously also performs symbolic execution of the program. The program path followed is recorded as a symbolic expression which gets refined with execution of the set of tests. The resulting invariants express output as a function of inputs conditioned on the program path.

Although PRECIS generates invariants similar in form to those of DySy, it does it without symbolic execution. Program paths are tracked with lightweight instrumentation of branch conditions. As a result, it has much less overhead and does not require dealing with typical issues with symbolic approaches such as handling floating point numbers.

DySy simplifies the path conditions using symbolic engines before presenting the invariants to the user, which is not possible with PRECIS. However

the predicate words present in PRECIS invariants provide a direct mapping to the program paths the invariant covers which can be useful to the user.

5.1.4 Krystal

Krystal [18] generates data structure invariants through a variant of symbolic execution called *universal symbolic execution*, which is argued to be more powerful in deriving complex invariants over heaps. Since PRECIS defines an input corresponding to each heap modification, it can also generate invariants that capture dependence of output on heap modifications.

5.1.5 Statistical debugging

Statistical Debugging is a dynamic technique that employs predicates extracted from branch conditions in the program, wherein the predicates are used in a very different context than in bug predictors [39, 40, 41]. These techniques analyze the correlation of predicate values to the success or failure of a program run and evaluate the utility of predicates for bug localization. PRECIS employs predicates to encode program paths for the purpose of generating invariants. In [40], predicates are clustered for localizing multiple bugs. It should be noted that unlike this approach, PRECIS clusters *predicate words* that encode program paths and not predicates themselves.

5.2 Fault Detection

When deciding which invariant generation technique to use for detectors, the two broad categories of existing approaches are static and dynamic techniques. Static approaches [1, 2, 3] generate concise and correct invariants, but have scalability issues for real-world programs. Dynamic approaches like Daikon and DIDUCE [14, 15] are more scalable and modular than static or symbolic approaches. However, many of the generated invariants can be spurious, since a property may hold for the gathered training data but not for all valid executions. In addition, these techniques generate a large quantity of invariants on relatively trivial relationships. Both of these factors make automatic integration of the generated invariants into code difficult.

The work of Pattabiraman et al., "Automated Derivation of Application-Specific Error Detectors Using Dynamic Analysis" [29] has a similar approach to our work. Both methodologies involve generating some set of assertions during a training phase and using these as a means of error detection.

A significant difference, however, is in type of assertions generated. Pattabiraman et al. is very similar to Daikon [14] in that there are predefined generic "rule classes," similar to Daikon's templates. This means it suffers the same problems as Daikon: the high number of false positives and the large amount of low-quality invariants. These shortcomings are especially troublesome in an error detector, because false positives can mean necessary interruption and re-execution of critical software.

Furthermore, since the rule classes from which the assertions are generated are fixed, they cannot adapt to the specifics of the target application. Instead, they must be sufficiently broad so that they are applicable in a wide range of use cases. For example, one rule class generated by the method in [29] is $x > y$. In this case, rather than checking what the value of x or y *should be*, an assertion is effectively checking what the value *should not be*. This contributes to the false positive problem discussed earlier; since the assertion generated is broad, an incorrect one may not be invalidated in a limited number of executions. In addition, this means that the assertions are less useful as error detectors; the broader the assertion, the more likely that an error will not be detected by it.

In contrast, the assertions generated by our technique are succinct yet expressive. Instead of requiring behavior to fall into one of the predefined rule classes, we capture expected program behavior in terms specific to the program itself. Rather than attempting to generate an assertion that captures all behavior of a variable, capturing program path information allows us to consider only a single straight-line execution through the program at a time. Since these straight-line executions are much simpler than the total behavior of the program, this allows the generated assertion to be much more strict than an assertion that singlehandedly attempts to cover all executions. And since we simultaneously merge the models constructed for a single path through predicate clustering, we can narrow down the behavior to the minimal set of such assertions that describe program behavior. This leads to a low false positive rate, and high coverage per assertion.

5.2.1 Tightness

We can quantitatively examine the degree to which our generated assertions more specific through the measure of **tightness** introduced in [29]. In that work, tightness is used as a means to predict which assertion of multiple candidates will be most likely to detect an error.

Tightness is not a mathematically sound measure of the coverage of an invariant; the following analysis makes assumptions that may not be true in real-world code. However, we use the same set of assumptions as in [29] to show how the format of invariants generated by PRECIS are quantitatively higher quality by the same metric as used in the previous work.

To make our analysis general, we will make the following assumptions about the nature of errors and detectors:

- The distribution of errors is uniform throughout the set of N possible values for any given variable.
- Variables are independent; an error in a variable does not affect the likelihood of an error in other variables.
- Detectors are independent; the distribution of other detectors in code does not affect the likelihood of an error occurring at any other detector.
- An error is equally likely to occur on any single program path, and that all generated invariants have the same likelihood of detecting an error in their output variables.

If we define *tightness* as $(1 - P(Inv))$, where $P(Inv)$ is the probability of an error in output O being caught by invariant I which describes that output under the program path being executed, we can write an expression for the tightness of the invariants our technique generates:

$$P(Inv) = P(Ante) * P(Cons|Ante) + (1 - P(Ante)) * P(Covered|\neg Ante) * P(Cons|Ante)$$

where:

- $P(Ante)$ is the probability that the antecedent in the invariant expression will be evaluated correctly (to true) with an incorrect value.
- $P(Cons|Ante)$ is the probability the evaluated consequent will detect the error given the antecedent is evaluated correctly.

- $P(Covered|\neg Ante)$ is the probability there is another invariant on output O whose antecedent (incorrectly) evaluates to true given the incorrect value.

More fundamentally, we can think of this equation as expressing a series of contingencies for catching the error. The most likely way to catch an error is if we can correctly detect which program path was traversed, and evaluate the consequent for that path on the corresponding output variable. Since the antecedent and consequent check for a strict linear equation, the likelihood of an error being detected is $(N-1)/N$, since any different value than the expected output will be flagged.

However, it is possible that the antecedent is not evaluated correctly due to the error. In this case, the antecedent expression may evaluate to another program path, or may not be covered at all. The likelihood that it evaluates to another program path given our assumptions is the control-flow coverage of our output variable. In this analysis, we will conservatively assume this to be 1/2. Experimental results show this number to be much higher. In this case, the probability of detecting the error is the probability that the consequent of this incorrectly applied invariant happens to evaluate is again $(N-1)/N$.

Thus, if we evaluate the $P(Inv)$ expression with these parameters, we find that $P(Inv) = (N-1)^2/N^2 + (1-(N-1)/N) * (N-1)/2N$, which is considerably tighter than the bounded range rule used in the example in [29].

5.3 Bug Localization

To our knowledge, no previous technique proposes the combination of bug localization and invariant generation for improved software reliability as we do in this thesis with PREAMBL and PRECIS. However, both automatic invariant generation and bug localization are active topics of research with a number of related works.

The two main categories of automatic invariant generation tools are static [17, 18] and dynamic [14, 15, 30] analysis. DySy combines these approaches to generate invariants similar in form to those generated by PRECIS. DySy

performs a concurrent dynamic and symbolic executions of the target program, keeping track of the symbolic value of each variable during execution. These are used to construct an expression for each output during the given execution. This expression is considered to be an invariant for a given path, and the conjunction of all observed paths is the derived output invariant.

Static or symbolic approaches like DySy work well for small, self-contained programs but often have difficulty scaling to real-world applications. To reduce overhead, PRECIS can be applied to some subset of functions in the program. However, by their nature static approaches will need to build a model of the entire program, which may prove infeasible for large applications. A related concern is the use of binary libraries. Because a source-level implementation of the library may not be available, a model for its behavior cannot be statically derived and it cannot be symbolically executed.

Another approach to invariant generation is dynamic analysis of program executions. The Daikon tool instruments a target program to capture the values of variables during execution [14]. Invariants are generated by attempting to apply *templates* – common unary, binary, or ternary relations – to each single, pair, or triple of instrumented variables. In practice, the invariants generated by Daikon can be unspecific, in that they define acceptable ranges for values or what a value should not be, rather than finding an expression of what the value should be. This is because Daikon invariants attempt to capture behavior for all executions through the function. In contrast, since PRECIS invariants are path-specific, they can be simultaneously more specific and still cover a large proportion of function executions. In addition, Daikon often suffers from generating too many invariants – a typical function can have 20 or more generated invariants, some of which may be spurious and others which check trivial properties. This makes automatically checking the invariants infeasible, and requires manual effort from the developer to browse through the generated invariants and decide which are of interest.

DIDUCE is another dynamic approach that is instead designed for *online* anomaly detection [15]. For each variable of interest, it generates an invariant in the form of a bit-string mask representing the valid values for each bit in the binary representation of the variable. This mask is gradually loosened during the training phase to be as restrictive as possible while still allowing all observed values. During online execution, this mask invariant will trigger

when a bit-string not covered by the mask is observed. DIDUCE has been shown effective at detecting latent bugs. A downside is the invariants it generates are not very intuitive for humans, as they describe a valid bit-string for each variable.

We find that PRECIS is the invariant generation technique best suited for bug localization for a number of reasons. It uses dynamic analysis, and so scales better for large applications or those which use binary libraries. It generates a relatively small number of high-quality invariants, which makes automatic integration into code more feasible. These invariants are intuitive for humans to understand, so they are likely to help developers diagnose bugs. Furthermore, since these invariants are path-specific, they can be combined with a path-based bug localization technique to capture behavior specifically on problematic program paths. To our knowledge, no other invariant generation techniques can claim all of these advantages.

Early statistical bug localization techniques analyzed the correlation between *predicates* and failing runs [19, 41]. In these works predicates refer a Boolean expression that is inserted into program code, rather than the specific control-flow oriented meaning used by PRECIS. In particular, [19] is closely related to PREAMBL. Both tools use the same scoring metrics, but in PREAMBL it is applied to path profiles, while CBI to predicate profiles.

A more recent work by Chilimbi et al., `HOLMES`, proposed the use of path profiling to capture more information about failing runs while using less overhead [21]. `HOLMES` targets single-procedure, acyclic paths. For each execution, it identifies whether or not a given path was observed or executed. It then scores and ranks paths based on the same Sensitivity, Increase, and Importance metrics as CBI.

PREAMBL shares many similarities with `HOLMES`. Both localize bugs to specific path profiles, and both use the same scoring metrics proposed by [19]. However, PREAMBL localizes to variable-length, interprocedural paths. This allows for improved quality of localization: since PREAMBL selects the paths with maximal Increase scores, predicates that are not relevant to the bug are naturally not included in the localized path. Further, multi-procedure paths lead to better detection of bugs which result from interactions between procedures.

5.4 Regression Testing

The vast majority of both research and practice in detecting regressions is through the technique of *regression testing*. Regression testing involves maintaining a set of program test input and expected output values such that regressions are detected by a discrepancy in the output value. The problem of selecting, prioritizing, and minimizing regression tests has been approached in various contexts [42, 43, 44, 45, 46]. The work presented in this thesis is orthogonal to this body of work in that it captures relationships (invariants) between intermediate variables in the program in addition to the program input and output relationships captured by a typical regression test.

Regression testing based on value spectra [47] proposes capturing relationships between program states and entry and exit of functions (value spectra) in addition to the relationship between program inputs and outputs. Domain specific invariants to create regression tests have been proposed for web applications using Javascript and Ajax [48, 49].

CHAPTER 6

PRECIS IMPLEMENTATION

The goal of PRECIS implementation was to create a fully automated system to perform all steps of the PRECIS methodology, from program instrumentation, to data collection, to invariant generation, and to integration of those invariants into code for runtime checking. The following sections describe the implementation work that went into creating the PRECIS tool.

6.1 Program Instrumentation

The program instrumentation phase involves capturing all the inputs, predicates, and outputs relevant to PRECIS. This is nontrivial, for a number of reasons:

Prior work in instrumenting Daikon indicated that a static, source-to-source transformation for the instrumentation of the desired variables would be problematic for larger programs. Their tool, `dfec`, was developed as an extension to a C/C++ front end [50]. It works by instrumenting the entry and exit points of functions with statements which evaluate input and output values, and print them to a trace data file. `dfec` worked for smaller, simpler programs but quickly ran into issues when dealing with complex program constructs. Specifically, the instrumentation code needed to be added at every possible function exit, and for each input or output null-checking statements needed to be added, possibly multiple in the case of multi-level pointers or `struct`s.

Ultimately, these problems spurred Daikon to move to a dynamic instrumentation framework, Fjalar [32]. Fjalar is based on Valgrind [33], a well-known tool for the dynamic inspection of program state. Valgrind runs in a context enclosing the executing program while remaining in the same memory space. This allows for easy programmatic access to program state, in

Daikon's case the input and output variables. For example, hooks can be made such that Fjalar code is called whenever the executing program enters or exits a function. However, as Daikon is a C/C++ source level tool, it needs more information to identify the names and locations of source-level variables. Thus, the program is compiled with the GDWARF-2 standard debugging format. This information is read by Fjalar, and parsed to identify the stack memory offsets of the variables of interest. These offsets can then be queried during runtime to capture the desired variable values.

As Fjalar is a dynamic instrumentation approach, it proved to be more versatile than `dfec`. Much larger programs could be handled, including those with irregular control flow structure such as `goto` and `longjmp` statements.

To make use of the existing work that has been done in dynamic instrumentation of program state for the purpose of invariant generation, we build the PRECIS instrumentation tool on top of Fjalar as a plug-in. However, the program state information provided by Fjalar proved less than required for PRECIS for the following reasons:

First, each run of a function may have a different number of inputs and outputs, and so the instrumentation process must be able to dynamically adjust for that fact. Inputs and outputs can vary between runs for a number of reasons: input declarations might not be reached in some executions, certain inputs may be `NULL` during executions, and certain heap address locations may or may not be written during other executions. Thus, null-checking had to be integrated into the variable capture code.

Second, the value of predicates (control flow branches) is not easily available in a binary, even when compiled with debugging flags. This makes counting and instrumenting a function's predicates difficult.

Third, the presence of loops and structure of loops is also not readily available for analysis in a binary. This makes handling loops in the manner described by the PRECIS methodology difficult.

```
#ifndef PRECIS_LOOP_FUNC_HEADER
#define PRECIS_LOOP_FUNC_HEADER 1
void precis_loop_test(int io, int loop_num){}
#endif

#ifndef PRECIS_PREDICATE_FUNC_HEADER
#define PRECIS_PREDICATE_FUNC_HEADER 1
int precis_pred_test(int var, int p_num){return var;}
#endif

...

if (precis_pred_test(if_cond, 0)){
   ...
}

while (loop_cond) {
   precis_loop_test(0,0);
   ...
   precis_loop_test(1,0);
}
```

Figure 6.1: Example source file `test.c` after initial source-to-source transformation.

To address the second and third concerns, an initial source-to-source transformation had to be performed to in effect trigger the existing hooks available in Fjalar. This transformation does not intrinsically add any instrumentation. However, a dummy function is defined for predicates and for each loop. The dummy predicate function is called with the evaluated result of that predicate, and the unique predicate number corresponding to the predicate. A loop is instrumented with at least two dummy loop function calls: one at the start of the loop and one at each possible exit. Each such call passes 0 or 1 depending on whether the loop is being entered or exited, and the unique number of the loop in the program.

Figure 6.1 shows an example transformed source file. When the program is executing, `precis_pred_test` will be called whenever a predicate is evaluated. PRECIS is called during each function entrance; in this case it reads

and saves the formal parameters as the value and number of the predicate, respectively. Similarly, when `precis_loop_test` is called with `io=0`, the context of the calling function (which is the next function on the stack) is noted. Loop inputs are captured. Then, when it is called at the end of a loop, the loop outputs are captured. Any predicates encountered in between are saved in the context of the loop, not the surrounding function.

6.2 Trace Data Format

Figure 6.2 shows the CSV trace data format outputted by Fjalar and used by the PredicateClustering tool. It encodes the variable names of each input, predicate, and output, along with all observed values. Each line represents a single execution of that function. Values that were not observed during a given execution are encoded with an `INVALID` value.

```
<function_name>:
i,i,...,i,p,p,...,p,o,o,...,o
<i0_name>,..,<iN_name>, p<p_N1>,..,p<p_NN>,<o0_name>,..,<oN_name>

<i0_val>,..,<iN_val>,<p_V1>,..,p<p_VN>,<o0_val>,..,<oN_val>
<i0_val>,..,<iN_val>,<p_V1>,..,p<p_VN>,<o0_val>,..,<oN_val>
[Repeated for all lines of trace data]
```

Figure 6.2: Generated trace data CSV format.

6.3 Predicate Clustering

Predicate clustering is performed by the `PredicateClustering` tool, a program written in Java. `PredicateClustering` reads in the CSV trace data file shown in Figure 6.2. It then performs the steps of predicate grouping, seed cluster generation, and invariant generation discussed in Chapter 2. The end result is a set of generated invariants, which can be exported in a human-readable format for programmer review, or a machine-readable format as shown in Figure 6.3.

6.4 Invariant Format

Figure 6.3 shows the format of machine-readable invariants generated by the `PredicateClustering` tool. The header of the file contains information about the function name, the input variable names, and the predicate numbers. Following this is a set of outputs. Each output contains a list of predicate word to output function mappings. The predicate word is simply a list of values (`1`, `0`, or `X`), depending of the observed value of the predicate in that position. The mapping from predicate positions to predicate numbers is indicated in the header: the first predicate number on the third line of the header corresponds to the first shown value in the predicate word, and so on. The input vector is a list of coefficients corresponding to the linear output function. The mapping between coefficients and variable names is the same as the second line of the header, except that the input vector has one more coefficient, corresponding to a constant offset.

```
<function_name>:                                          <--
<input0_name>,<input1_name>,...,<inputN_name>             <-- Header
p<p_Num1>,p<p_Num2>,...,p<p_NumN>                         <--

<output0_name>{
   <pWord0>:[input vector]
   .
   .
   .
}

<output1_name>{
   ...
}

...
```

Figure 6.3: Generated invariant machine-readable format.

6.5 Automated Assertion Integration and Checking

Automatic invariant integration is done through the PRECIS plug-in to Fjalar. Once the machine-readable invariant format shown in Figure 6.3 is produced, it can be passed into Fjalar with the `--check-invariants` command-line argument. This will in effect use the same instrumentation process as was used in generating the trace data file, except instead of generating the file will check the observed output values against the encoded invariants. This is done by gathering the inputs listed in the invariant file and computing an expected output based on the output function for the predicate word observed. If the observed output and expected output match, the assertion passes. Else, the assertion fails and the program exits.

6.6 Fault Injection Framework

The PRECIS plug-in to Fjalar has also been extended to inject faults into running programs, specifically for the work discussed in Chapter 3. Fjalar is a naturally intuitive place to put fault injection for several reasons. First, it provides easy access to program state. Given that this is done dynamically, it is not very difficult to simply change the value stored at a given address, rather than capture it as is done in the PRECIS flow.

However, instead of dividing observed variables into inputs and outputs, all program state is considered as a candidate for injection. This includes local variables, heap variables, and other variables not typically captured for PRECIS. A minor extension to Fjalar was required to be able to access and modify this program state.

CHAPTER 7

CONCLUSION

7.1 Lessons Learned

The following is a list of some lessons learned in the time spent developing the techniques discussed in this thesis:

- **Path information is an intuitive way to divide a program's behavior.** Isolating the behavior of a single program path is in effect separating the "when" (the path condition) from the "what" (the end result) in the behavior of the program. The forward direction – *given a path, what should the behavior be?* – is intuitive as a way to structure invariants as generated by PRECIS. The backward direction also proved useful – *given some behavior, what is the program path associated with it?* This reverse relationship is used in PREAMBL to localize program misbehavior to certain program paths.

- **Building a tool to automate PRECIS is nontrivial.** A compiled C/C++ program does not lend itself easily to source-level analysis. Once a program is compiled, most to all source level information such as variable names, if statement conditions, and loop structure is lost. Programs compiled with debugging flags retain some of that information, but not all. Thus, a source-to-source transformation must be performed first to trigger events that could be observed by a dynamic instrumentation tool. In this, Valgrind and Fjalar proved invaluable as they provided a way to examine the contents of program state from an enclosing state. This increased overhead, but provided a way for much simpler, relatively speaking, instrumentation.

7.2 Future Work

There is a great deal of future work possible to extend the techniques discussed in this thesis.

An important step in linking the regression testing and bug localization phases of software development is an automatic way to associate localized paths derived by PREAMBL with invariants generated by PRECIS. At present, paths are associated with invariants by human inspection. However, given large programs this would be impractical. A relatively simple addition to PREAMBL would be a way of reading in the invariant format generated by PRECIS and reporting the associated invariants in user-readable terms. This will greatly improve the relevance of the localized paths by associating them with some behavior.

7.3 Final Thoughts

This thesis explored the idea of automatically generating program invariants and applying them to research areas across the field of software engineering.

PRECIS is an invariant generation technique that uses predicate data in combination with a linear regression combination to derive function-level invariants. The three-step process of data generation, predicate clustering, and invariant generation generates high-specificity, high-coverage, and high-quality invariants.

The invariants generated by PRECIS can be used to detect hardware faults. The invariants can be integrated into program text for software-level checking, or used on specialized hardware for low overhead. Fault injection experiments show that these invariants have high-coverage for a several types of manifested errors.

The same predicate information gathered by PRECIS can also be used for bug localization. PREAMBL localizes bugs to variable-length interprocedural paths through a statistical path-scoring algorithm. This process is optimized through the use of a path-tree data structure. These localized paths can then be associated with invariants to improve the software debugging process.

A major theme of this thesis is the use of existing code to improve the

software development process. Past approaches to static invariant generation have shortcomings due to the path-explosion problem, while dynamic approaches have remained too general and unspecific. This thesis explores an answer to these problems by using path information (in the form of predicates). Program path predicates provide a unique "fingerprint" for each execution, which "filters" the specific behavior for that particular path. Since each individual path tends to have relatively simple behavior, this allows for the piecemeal analysis of each path individually. These analyses can then be recombined to form a model for the entire program.

This thesis has shown the promise of predicate-based automatically generated invariants for improvements in regression testing, bug localization, and fault detection. Continuing with a similar approach holds promise for even greater improvements in those areas, as well as new applications in other challenging areas of software development.

APPENDIX A

GETTING PRECIS AND PREAMBL

PRECIS and PREAMBL are available on an University of Illinois at Urbana-Champaign wiki page. Documentation on how to use the tools is included.

For more details, go to:

`https://wiki.engr.illinois.edu/display/precis/`.

REFERENCES

[1] J. Misra, "Prospects and limitations of automatic assertion generation for loop programs," *SIAM Journal on Computing*, vol. 6, no. 4, pp. 718–729, 1977.

[2] N. Bjorner, A. Browne, and Z. Manna, "Automatic generation of invariants and intermediate assertions," in *Theoretical Computer Science*. Springer-Verlag, 1997, pp. 589–623.

[3] S. Bensalem and H. Saidi, "Powerful techniques for the automatic generation of invariants," in *CAV*. Springer-Verlag, 1996, pp. 323–335.

[4] A. Almossawi, K. Lim, and T. Sinha, "Analysis tool evaluation: Coverity prevent," Ph.D. dissertation, Carnegie Mellon University, 2006.

[5] Hewlett-Packard, "HP Fortify static code analyzer (SCA)," 2012. [Online]. Available: https://www.fortify.com/products/hpfssc/source-code-analyzer.html

[6] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)*, Sept 2005.

[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," in *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, 2004, pp. 289–302.

[8] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proc. of POPL '02*, ser. POPL '02. New York, NY, USA: ACM, 2002. [Online]. Available: http://doi.acm.org/10.1145/503272.503275 pp. 4–16.

[9] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for java," in *Proceedings of the First NASA Formal Methods Symposium*, 2009, pp. 121–125.

[10] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[11] K. Sen, D. Marinov, and G. Agha, *CUTE: A concolic unit testing engine for C*. ACM, 2005, vol. 30, no. 5.

[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on operating systems design and implementation*. USENIX Association, 2008, pp. 209–224.

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

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. of ICSE '99*, ser. ICSE '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/302405.302467 pp. 213–224.

[15] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. of ICSE '02*, ser. ICSE '02. New York, NY, USA: ACM, 2002. [Online]. Available: http://doi.acm.org/10.1145/581339.581377 pp. 291–301.

[16] J. Henkel and A. Diwan, "A tool for writing and debugging algebraic specifications," in *Proc. of ICSE '04*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=998675.999449 pp. 449–458.

[17] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *Proc. of ICSE '08*, ser. ICSE '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368127 pp. 281–290.

[18] Y. Kannan and K. Sen, "Universal symbolic execution and its application to likely data structure invariant generation," in *Proc. of ISSTA '08*, ser. ISSTA '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390665 pp. 283–294.

[19] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[20] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound Boolean predicates," in *International Symposium on Software Testing and Analysis*, S. Elbaum, Ed., London, United Kingdom, July 9–12 2007.

[21] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *31st International Conference on Software Engineering (ICSE 2009)*, J. Atlee and P. Inverardi, Eds. Vancouver, Canada: ACM SIGSOFT and IEEE, May 2009.

[22] Wikipedia, "Windows error reporting — wikipedia, the free encyclopedia," 2012, [Online; accessed 12-April-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Windows_Error_Reporting

[23] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 2nd ed. Morgan Kaufmann, Jan. 2006.

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

[25] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of ICSE '94*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=257734.257766 pp. 191–200.

[26] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[27] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63 –75, Mar. 2002.

[28] M. Rela, H. Madeira, and J. Silva, "Experimental evaluation of the fail-silent behaviour in programs with consistency checks," in *Proceedings of Annual Symposium on Fault Tolerant Computing*, Jun. 1996, pp. 394–403.

[29] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-specific error detectors using dynamic analysis," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 5, pp. 640–655, Sept.-Oct. 2011.

[30] P. Sagdeo, V. Athavale, S. Kowshik, and S. Vasudevan, "Precis: Inferring invariants using program path guided clustering," Nov. 2011, pp. 532–535.

[31] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu, "An architectural framework for providing reliability and security support," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN '04. Washington, DC, USA: IEEE Computer Society, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1009382.1009774 pp. 585–594.

[32] P. J. Guo, "A scalable mixed-level approach to dynamic analysis of C and C++ programs," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 5, 2006.

[33] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[34] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[35] H. Leung and L. White, "Insights into regression testing [software testing]," in *Proceedings of the Conference on Software Maintenance*. IEEE, 1989, pp. 60–69.

[36] B. Beizer, *Software Testing Techniques*. Dreamtech Press, 2002.

[37] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[38] F. Vokolos and P. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 1998, pp. 44–53.

[39] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065014 pp. 15–26.

[40] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the 23rd international conference on Machine learning*, ser. ICML '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1143844.1143983 pp. 1105–1112.

[41] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081753 pp. 286–295.

[42] S. Yoo and M. Harman, "Tr-09-09: Regression testing minimisation, selection and prioritisation – A survey," 2009.

[43] H. K. N. Leung and L. White, "A cost model to compare regression test strategies," in *Proceedings of the Conference on Software Maintenance*, 1991, pp. 201–208.

[44] T. Ball, "On the limit of control flow analysis for regression test selection," in *Proceedings of the 1998 ACM SIGSOFT ISSTA*, 1998, pp. 134–142.

[45] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering*, vol. 23, pp. 498–516, 1997.

[46] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, 2004, pp. 241–252.

[47] T. Xie and D. Notkin, "Checking inside the black box: Regression testing based on value spectra differences," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 28–37.

[48] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38, pp. 35–53, 2012.

[49] S. Mirshokraie and A. Mesbah, "JSART: JavaScript assertion-based regression testing," in *ICWE*, vol. 7387, 2012, pp. 238–252.

[50] B. Morse, "A C/C++ front end for the daikon dynamic invariant detection system," Ph.D. dissertation, Massachusetts Institute of Technology, 2002.