Automatic Debugging Using Potential Invariants

by
Brock Pytlik

A Thesis submitted in partial fulfillment of the requirements for Honors
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2003

© Copyright 2003 by Brock Pytlik

This thesis by Brock Pytlik is accepted in its present form by
the Department of Computer Science as satisfying the research requirement
for the awardment of Honors.

Date ________________ ________________________________

Shriram Krishnamurthi, Reader

Date ________________ ________________________________

Steve Reiss, Reader

Acknowledgements

I would like to thank Manos Renieris for his guidance on this project. I also want to thank my advisors and readers, Shriram Krishnamurthi and Steve Reiss who oversaw this project and reviewed my thesis.

Contents

List of Tables vi

List of Figures vii

1 Introduction 1

2 Methodology 3

2.1 Carrot's Process 4

2.1.1 Instrumenter 4

2.1.2 Oracle 5

2.1.3 The Invariant Generator 5

2.1.4 The Invariant Differencer 6

2.2 A Possible Alternative Methodology 6

3 Related Work 7

4 Potential Invariants 9

4.1 Value Sets 9

4.2 Relational Potential Invariants 10

5 Level of Instrumentation 12

6 Experiments and Results 13

6.1 Value Sets 14

6.1.1 Is adding another good run to the model unlikely to cause extensions? 14

6.2 Relational Potential Invariants 17

6.2.1 Is adding another good run to the model unlikely to cause invalidations? . . . 17

6.2.2 Is adding a bad run likely to cause invalidations? 21

6.3 Analysis 23

7 Nascent Work 24

7.1 Using Temporal Information:The Map Back Program 24

7.1.1 The Map Back Program Details . . . 24
7.1.2 The Map Back Program Results . . . 25
7.2 More Analysis using Temporal Information . . . 25
7.2.1 The Map Back and Remove Duplications Program Details . . . 25
7.2.2 Results for The Map Back and Remove Duplications Program . . . 25

8 Future Work **27**

9 Conclusion **28**

A Function Relationships **29**

B Recognize **31**
B.1 Correct Recognize with Faults of Versions Commented . . . 31
B.2 Faulty Recognize Version 3 . . . 33
B.3 Faulty Recognize Version 7 . . . 35
B.4 Faulty Recognize Version 8 . . . 37

C Recognize2 **39**
C.1 Correct Recognize2 with Faults of Versions Commented . . . 39

D Examples of Bugs in tcas in the Siemens Suite **43**
D.1 tcas Faulty Version 1 . . . 43
D.2 tcas Faulty Version 2 . . . 43
D.3 tcas Faulty Version 13 . . . 43

Bibliography **45**

List of Tables

6.1 The second column shows the range in the number of invalidations that occur. For example, 0-4 in *print_tokens* means that at least one run causes 0 invalidations and at least one run causes 4 invalidations. The third column shows the number of traces, across all versions that of a program, that cause at least one invalidation. The fourth column shows the total number of traces that exist across all versions. 21
6.2 The invalidations for *recognize* for each faulty version. 21
6.3 The invalidations for *print_tokens* for each faulty version. 22
6.4 The code for `keyword` in the correct version of *print_tokens* and the faulty version 8 22

A.1 The call structure of the code on the right is pictured in the graph on the left. The labels for each node are for when the current function is `C`. 29
A.2 The call structure of the code on the right is pictured in the graph on the left. The labels for each node are for when the current function is `D`. 30

List of Figures

6.1 Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *tcas* . . . 14

6.2 Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *print_tokens* . . . 15

6.3 Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *replace* . . . 15

6.4 Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *recognize* . . . 16

6.5 Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *recognize2* . . . 16

6.6 Number of extensions (on the y axis) made by each run (1200 - 2400) as it was added to the existing model for the correct version of *print_tokens* . . . 17

6.7 Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *tcas* . . . 18

6.8 Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *print_tokens* 18

6.9 Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *replace* . . . 19

6.10 Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *recognize* . . 19

6.11 Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *recognize2* . 20

6.12 A closer look at the first 200 runs of *print_tokens* . . . 20

Chapter 1

Introduction

Bugs often are caused by erroneous assumptions about a program and its intended use. One reason debugging is difficult is that it often involves challenging or correcting these bad assumptions. Finding and examining those assumptions can be an arduous process because there is a large amount of code and many assumptions that must be examined. Also, examining objectively what we think to be true is difficult. Tools which check these assumptions are valuable because they are less likely to share the programmer's bias.

A second reason debugging is difficult is that it often involves understanding and processing a large amount of information simultaneously. A bug is a global property of the program. Ideally, programmers would look at a problematic run and analyze it effectively in the context of the entire code base to determine where the bug lies. Few programs are small enough, or few programmers are talented enough, to be able to understand and process the entire code base of a large project. The solution most programmers use is to examine pieces of the program individually, assuring the correctness of that piece. Until a debugging program is created which can understand the purpose of the program and identify the problematic point, or all code comes with detailed formal specifications, the best that can be done is to assist the programmer to eliminate parts of the program that do not need to be examined and suggest pieces of code to examine first.

One way to facilitate debugging is to provide a tool to examine the differences between correct runs and faulty ones. In this paper we present a method for comparing runs by contrasting program traces. This method does not depend on any knowledge of the program's input structure. First, we find predicates that are true for all correct runs. Then we find places in a faulty run which contradict those predicates. We believe that potential invariants[1] can serve as the predicates which we compare.

A potential invariant is a relationship of one, two, or more variables, to which a counter example has not been found. Potential invariants offer abstraction from the written code of the program. Potential invariants can allow a programmer to chunk information into larger structures, reducing the effort needed to understand a program in its entirety. We conjecture that comparing potential invariants of working and non-working runs of a program will offer useful information to find bugs

in the non-working programs. To test these ideas, we built *Carrot* which finds and compares these potential invariants.

The rest of this paper is structured as follows: Chapter 2 describes our methodology; Chapter 3 discusses related work; Chapter 4 describes the potential invariants *Carrot* uses; Chapter 5 discusses the level of instrumentation *Carrot uses*; Chapter 6 describes experiments and results; Chapter 7 outlines two approaches to improve performance that we explored but for which we do not have formal results; Chapter 8 describes other ideas for future work; Chapter 9 outlines the conclusions we have drawn from the project.

Chapter 2

Methodology

Ideally, fault localization tools do not require a specification of the subject program's intended behavior or the structure of its inputs and outputs. Requiring any of these pieces of information creates more work for the programmer and limits the domain of programs which the tool can analyze. For example, creating specifications for legacy code is a difficult and time-consuming process. One piece of information remaining is that a run has failed, which often is not inherently informative. This knowledge becomes valuable when situated in a larger context of information. The context that *Carrot* uses is a (large) number of runs that do not fail. This context does not provide much information that would be useful for localizing a bug unless it is augmented with information about what happens during runs, i.e. program traces.

Our method contrasts the traces of working runs with the traces of the runs that did not work. To incorporate information from many successful runs, we construct a model representative of successful runs and contrast that model with a unsuccessful run. Rather than viewing a run as a sequence of events, *Carrot* views a run as a set of predicates that held during the run. We call these predicates **potential invariants**. A potential invariant is a relationship, of one, two, or more variables, to which a counter example has not been found. Our method uses two types of potential invariants, value sets, which track individual values of variables, and relational potential invariants, which track relations between values of variables. Value sets are extended to accommodate new values while relational potential invariants are invalidated when a counter example is found. (See chapter 4 for more details about potential invariants.)

Carrot can compute the set of potential invariants that are true for every run in a set of runs. As the size of the set of runs grows, the number of relational potential invariants should drop since more counter examples will be seen. The potential invariants a run invalidates or extends show situations which have not occurred in the runs in the previously examined set. Those potential invariants which remain after a set of runs have been examined form a model of those runs. Given a sufficiently large test suite, the model will stabilize as the rate of novel situations seen drops.

To compare the working and non-working runs, *Carrot* finds the potential invariants which hold for all working runs. Next *Carrot* identifies the potential invariants in the set that are broken by a

single non-working run. These differences demonstrate what makes this run different from all the working runs and, hopefully, what makes it fail. To call *Carrot* successful, there must be a correlation between many of the differences reported and the bug in the program. We would like for a causal relation to be apparent between the differences reported and the bug, but a correlation is easier to establish and serves as a reasonable initial goal.

This work focuses on examining a single non-working run. For example, imagine a company has released a program which has passed a large test suite. A customer discovers a bug and sends them a report. By finding the differences of the run which caused this bug and the runs over the rest of the test suite, *Carrot* should be able to locate the bug.

A single faulty run has the advantage that it likely stems from a single bug. Combining multiple non-working runs might be more complicated because each of the non-working runs might be caused by a separate bug.

2.1 Carrot's Process

Carrot needs four inputs. First, it needs the program to be debugged. Second, it needs a test suite of inputs for that program and the outputs that a correct program would produce. Third, it needs an oracle for the correctness of a run. Fourth, it needs a faulty run.

Carrot consists of 4 pieces: the instrumenter, the oracle, the invariant generator, and the invariant differencer. The instrumenter takes the code of the subject program and produces modified code and a file. That file lists the program locations, the function entry locations or exit locations at which information is taken, and the variables at each program location about which the information is taken. This file is called the **declaration file**. After the instrumented code is compiled, it is run using each entry in the test suite as input. Each run produces a trace of the program execution and the output of the program, if one exists. An oracle determines if the run is correct or faulty. Next, all of the correct traces and the declaration file are given to the invariant generator. The invariant generator produces a list of potential invariants for each program location. For each faulty trace, *Carrot* gives the bad trace and the potential invariants that hold for all good traces to the invariant generator which removes any of the potential invariants which are not true for the bad trace. *Carrot* then feeds the potential invariants from all good traces and the potential invariants from all good traces and one bad trace to the invariant differencing program which finds the potential invariants that are modified or invalidated by the bad run.

2.1.1 Instrumenter

To instrument programs, we use *dfec*[1], *daikon*'s front end for C. *Dfec* instruments programs at function entries and exits. The instrumented code reports the program point and the values for all function parameters, global variables, object fields that are in scope, and, at function exits, the return value. Each call or return is recorded individually in the order it happens.

2.1.2 Oracle

The oracle is specific to each program being debugged. It must be able to distinguish between a correct run and a incorrect run.

2.1.3 The Invariant Generator

The invariant generator takes a declaration file and a set of trace files for which potential invariants are to be inferred. The invariant generator has a predefined set of potential invariant schemas to use. Potential invariant schemas abstractly represent the different relationships which may occur between any variables at any program point. Tying potential invariant schemas to specific program locations or trace points and variables creates potential invariants. For each program location in the declaration file, the invariant generator instantiates the potential invariant schemas over each permutation of variables at that program point. For example, suppose a function has three integer parameters, x, y, z and that one of the potential invariant schemas represents a test for one variable always being less than another variable. The list of potential invariants would include

- $x < y$
- $y < x$
- $x < z$
- $z < x$
- $y < z$
- $z < y$

Some relational potential invariants are instantiated only during the run. For example, the *sum* potential invariant ($x + y = c$ where x and y are variables and c is some constant) is complete once the invariant generator can compute c, which occurs the first time that x and y are bound to specific values.

Once all potential invariants are instantiated, the invariant generator checks them against the set of trace files. The invariant generator begins by assuming that all potential invariants are true, including those that are mutually contradictory. At every trace point in every trace file, the remaining potential invariants are checked against the values for variables at that runtime point. If the values present a counter example to the potential invariant, the potential invariant is either invalidated and removed from the list of potential invariants, or it is modified and left in the list of potential invariants. Which action is taken depends on the type of the potential invariant. (See chapter 4 for more detail.) For example, suppose the value set for x at some program location is $\{3,4\}$. Next, suppose that while examining the trace, x is 3 at some point, then nothing happens to the value set. Finally, suppose that after that, x is 5, then the value set is marked as changed and is expanded to $\{3,4,5\}$. Similarly, imagine that $x < y$ has always been true at some program location.

If at some trace point, $x = 3$ and $y = 4$, then nothing changes. If at another trace point, $x = 5$ and $y = 4$, then the potential invariant that $x < y$ is invalidated.

2.1.4 The Invariant Differencer

The invariant differencer takes two inputs, each containing sets of potential invariants. One input contains all of the potential invariants that were still true after all good traces had been examined. These potential invariants form the **basis set**. The other input contains all of the potential invariants left after all good traces and one bad trace were examined. These invariants are the **remaining potential invariants**. For each program location, the invariant differencer examines each potential invariant in the basis set of potential invariants and determines if the same potential invariant exists at the same program location in the set of remaining potential invariants. If the potential invariant has been deleted in the set of remaining potential invariants, the invariant differencer reports that the potential invariant has been invalidated. If the potential invariant has been modified when the bad run is included, the invariant differencer reports the differences between the basis set and the remaining set. The invariant differencer reports each program location and the differences that occurred at that program location.

2.2 A Possible Alternative Methodology

An alternative comparison method would be to find the potential invariants of a single non-working run, and find which of those potential invariants did not hold for all working runs. If our system of potential invariants is used, this approach would not work. For a single run, the set of potential invariants remaining is typically very large and reflects the eccentricities of the single run rather than being indicative of the typical behavior of the program. Also, the set of differences would be enormous and would not indicate any specific bug. For example, if a function which was called once took two integers as arguments, a and b, the values of these arguments would obey one of three relations:

- $a < b$
- $a = b$
- $a > b$

This would be viewed as a potential invariant although it was a chance result of a single run.

Chapter 3

Related Work

Michael Ernst developed the notion of dynamically discovering invariants for a program[1]. His system, *daikon*, begins by instrumenting the program at function entries and exits. The programs are run multiple times, producing traces of the runs. The traces contain information about the values of function parameters, global variables, object fields that are in scope, and the return value at function exits. The traces are analyzed by *daikon* to determine which potential invariants hold for the runs examined. A potential invariant is a relationship among some variables which holds for all examples of values for these variables seen. *Daikon* looks for a predetermined set of invariants. *Daikon*'s invariants include value sets, which track the values variables have taken, and comparison invariants, which track the ordering relation between the values of two variables. Other examples of *daikon*'s invariants include the values of variables x and y summing to a constant, checking the ordering relationship of array elements, and using a pairwise comparison of vectors to determine which is less than the other. *Daikon* reports a confidence level for each invariant based on the number of examples seen. Ernst's thesis is that examining the invariants for a program provides interesting insight into how a program behaves.

Carrot builds off of the ideas behind *daikon*. It uses the idea of potential invariants from *daikon* as the unit for comparing good and bad runs. The major difference between *daikon* and *Carrot* is that *Carrot* is designed to identify bugs in a program using potential invariants while *daikon* is designed to notice interesting behavior of a program.

Sudheendra Hangal and Monica S. Lam developed *Diduce*[2], a system for detecting changes in established patterns of behavior for a program. Their approach begins by instrumenting program points, which are all of the read and write locations for objects and static variables as well as all of the procedure call sites. At each program point, *Diduce* tracks three types of values, "the value being read or written", "the parent object", and the difference between the old and new values.[2] All invariants are represented by bit strings. *Diduce* tracks the starting values of each bit in the bit string and whether each bit has taken a different value. As new values for bits are encountered, an invariant violation is reported and the invariant is changed to note that more bits have been seen with different values. *Diduce* detects invariant violations during a program run and can store the

invariants after a run to be used as the initial invariants for another run. *Diduce* also reports a confidence level associated with invariant violations which is a function of the number of examples which have confirmed the invariant.

Like *diduce*, *Carrot* uses the values of variables to examine the behavior of programs. While *diduce* examines the values on a bit level, *Carrot* records values separately and examines potential invariants in all combinations of variables. The largest difference between *Carrot* and *diduce* is that *Carrot* examines various relationships among variables which are not captured by examining bit differences.

Ralf Hildebrandt and Andreas Zeller created a system, *delta debugging*[4] for locating the pieces of input which caused problems for the program. *Delta debugging* selects pieces of the input to the program and determines if they produce the same bug as the larger input. If a piece of input does preserve the bug in the execution of the program, then that piece is divided into more pieces which are examined. If the piece in question does not preserve the bug in the execution of the program, other pieces and other combinations of pieces are examined to test their preservation of the bug. By iterating this processes, it becomes possible to locate the minimal input which reproduces the bug in question. It is also possible to determine the smallest difference in the input which causes the bug[6]. After establishing the smallest difference in the input, *delta debugging* builds a cause effect chain which states what the differences between the two runs are and what the causal relation is between the differences. *Delta debugging* is able to create the cause-effect chains by examining the program at the instruction level and finding the differences in instructions when the smallest difference which causes the bug is applied.

Carrot works at a higher level of abstraction than *delta debugging* does when looking for the bug location. *Carrot* locates the function which contains the bug while *delta debugging* works on an instruction level.

Chapter 4

Potential Invariants

Carrot currently uses six potential invariants. Those six potential invariants can be divided into two groups, value sets and relational potential invariants. Relational potential invariants describe relations among variables. The less than potential invariant in 2.1.3 is an example of a relational potential invariant. Typically, relational potential invariants can only be invalidated, not modified. The other group of potential invariants are value sets. This kind of potential invariant tracks the individual values that a variable or group of variables has taken. Value sets are never invalidated. Instead, as new values are encountered, the value set expands to include the new values seen.

The potential invariants in *Carrot* do not have confidence levels, unlike the potential invariants in *daikon*[1]. In *daikon*, confidence levels convey how much faith the program had in its potential invariants. The confidence levels were roughly based on the numbers of examples seen confirming the potential invariant. For the purposes of *Carrot*, confidence levels only complicate the comparison of potential invariant sets. Choosing an arbitrary minimum confidence level needed for a potential invariant to be reported does not seem a useful route to pursue. If a minimum level is chosen, then potential invariants can disappear because their confidence level drops, then reappear once more supportive examples are seen and their confidence level grows again. The effect of a non-working run on the confidence levels does not appear to be a useful measure for comparing the potential invariant sets.

4.1 Value Sets

Value sets record sets of values for variables. If a new value is encountered while checking a run, that value is added to the value set for the variable(s) associated with that value. Value sets are initially empty and are extended each time a new value is seen. Value sets provide a high degree of detail about a specific run of a program.

In this work, we use value sets of single variables and pairs of variables. In principle, it is possible to extend the number of value sets to a large, finite number. In practice, two problems occur as the number of variables is increased. First, the number of value set potential invariants for a particular

size at a program location is equal to p choose k where p is the number of variables and k is the number of variables the value set tracks. If p is large and $k \approx \frac{1}{2}p$ then constructing all possible instances of those value sets could be prohibitively expensive.

The second problem is sparse data. Assuming a finite test suite, as the dimensions of the value set space increases, the probability of selecting the same point twice drops. For example, if an integer variable can take on values between 1 and $20,000$, then a value set tracking that integer might have to cover a space of size 20,000. Suppose that there are now two integer variables that span 1 to $20,000$. A value set which tracks pairs of values now has to track a space of size is now 400,000,000. Clearly, the chances of encountering a value previously seen is much greater when the space has a size of 20,000 instead of 400,000,000. Tracking a large value space increases the number of differences which appear and increases the likelihood of the differences which are not related to the bug. The differences instead reflect the limited nature of the test suite.

Carrot uses two kinds of value sets.

- OneOf: A OneOf value set stores the value of a single variable at a single program location. Each time a new value is seen, the value set is augmented to include the new value.
- OnePairOf: A OnePairOf value set stores unique pairs of values. Suppose a function receives the values 4 and 5 for x and y, and that x and y have taken those values previously, but never at the same time. The value 4 would already be in the OneOf set for x and the value 5 would be in the OneOf set for y. These sets would not recognize any novel event. The OnePairOf would recognize that a new pair of values has been encountered.

4.2 Relational Potential Invariants

Relational potential invariants express a relationship among variables. The relationship can be equality between two variables at a program point, or it may be more complex. For example, the values of two variables always sum to the same value. Before any runs are examined, all relational potential invariants are assumed to be true. There are a pre-defined, finite, set of relational potential invariant schemas from which the relational potential invariants are created. Relational potential invariant schemas represent the relation among variables but are not attached to a specific program location or variable. A relational potential invariant is instantiated by applying a relational potential invariant schemata to a specific variable or variables at a specific program location. All possible relational potential invariants are created by applying them to all variables in scope at all program locations. A relational potential invariant is assumed true until shown to be false, at which point it is discarded. It is shown to be false when a counter-example is found. For example, if the relational potential invariant is that "at the entry to function `foo`, variable `x` is always less then variable `y`," but a run shows `y` is greater than `x`, the potential invariant is discarded.

Carrot current uses four kinds of relational potential invariants.

- Equality: The *equality* relational potential invariant compares if two variables were always equal at a program location.
- Sum: The *sum* relational potential invariant notes if two variables always summed to the same value. For example `x+y` always equaled 5.
- Less Than: The *less than* relational potential invariant checks if one variable was always less than another variable at a program location.
- Constant Equality: The *constant equality* relational potential invariant checks if a variable is always equal to a constant that was declared someplace in the code.

Chapter 5

Level of Instrumentation

Carrot gathers information at the function level by using *dfec*[1] to instrument function entries and exits. Programs instrumented by *dfec* record the values for function parameters, global variables, object fields in scope, and, at function exits, the return value of the function.

Programs instrumented by *dfec* do not track the their call structure. We believe that some rudimentary information about the calling contexts of functions might provide valuable information about program behavior. By examining the ordering of function entries and exits, the invariant generator gathers three pieces of information: which function was the last to be entered before this program point; which function was the last to exit before this program point; which function invoked the function at this program point. This information was treated the same as data provided by programs instrumented using *dfec*; variables were created and the variables' values were the names of the functions for the three contexts. For examples and pictures of these relationships see Appendix A.

Chapter 6

Experiments and Results

Does Carrot work? We must show the following things to be true for value sets or relational potential invariants to be useful.

- Value Sets
  1. Adding another good run to the model is unlikely to cause extensions
  2. Adding a bad run to a model is likely to cause extensions
  3. The extensions are related to the bug in the code
- Relational Potential Invariants
  1. Adding another good run to the model is unlikely to cause invalidations
  2. Adding a bad run is likely to cause invalidations
  3. The invalidations are related to the bug in the code

We test *Carrot* on three programs in the Siemens suite[3, 5], *tcas*, *print_tokens*, and *replace*. *tcas* is a large predicate that handles collision avoidance between two airplanes. It takes thirteen integers as inputs and compares these values to determine the output. The second program from the Siemens suite is *print_tokens*. It tokenizes its input file and outputs the set of tokens. The other program from the Siemens suite is *replace*. *Replace* takes a string and a specifications for substitutions, written as regular expressions, and performs the substitutions.

In addition to those programs, we created two small programs *recognize* (see Appendix B) and *recognize2* (see Appendix C) that determine whether an input string matches the regular expression $(a(bb+cc))^*)$.

Each program has a correct version, a number of versions with a single inserted fault, and a set of inputs. Each faulty version has at least one input that causes the faulty version to produce an incorrect output. We use the correct version as an oracle to provide the correct answer for each input.

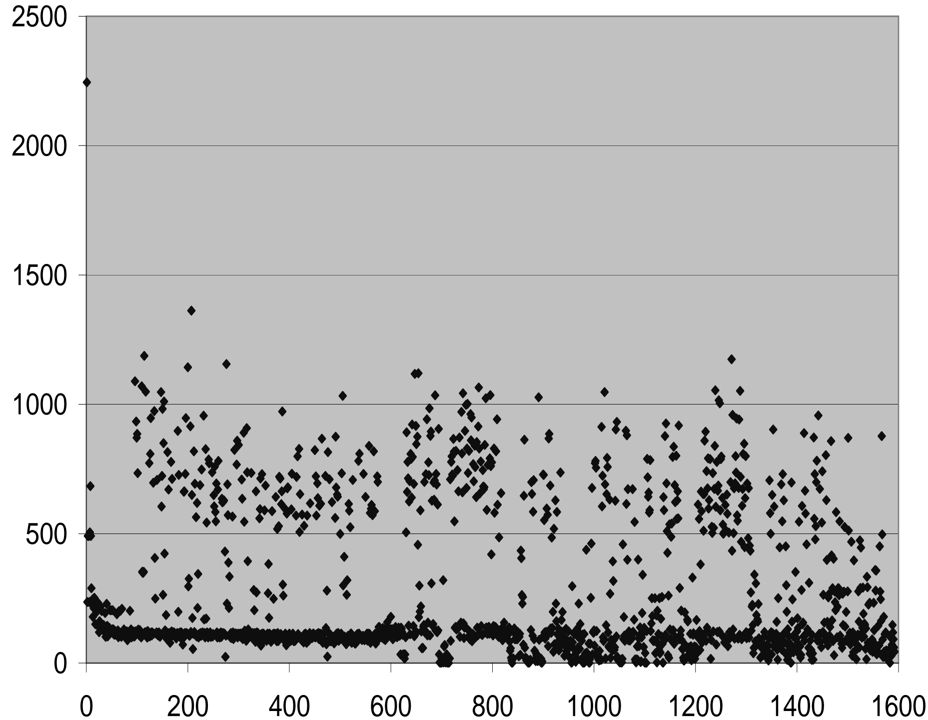

Figure 6.1: Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *tcas*

To check the first requirements for value sets and relational potential invariants, we gather information about how the correct versions of all 4 programs perform. To check the second and third requirements, we examine all faulty versions available for *tcas* (41 faulty versions), *print_tokens* (8), *recognize* (10) and *recognize2* (9).

6.1 Value Sets

6.1.1 Is adding another good run to the model unlikely to cause extensions?

We examine the number of extensions made to the model after each run was added. We test the correct versions of *tcas*, *print_tokens*, *replace*, *recognize*, and *recognize2*. Figures 6.1, 6.2, 6.3, 6.4 and 6.5 show the pattern of extensions that occur as more runs are added to the existing model. The graphs for *tcas*, *print_tokens*, *recognize* and *recognize2* clearly show that adding more good runs does cause many extensions to value sets. Because so many extensions happen when any run is added, any interesting extensions a faulty run causes will be lost in the noise. *Print_tokens* (figure 6.2) displays a unique pattern. The runs where the lull of extensions occurs cause a designed error to happen. These traces tend to be very short and simple. While the number of invalidations appears small in comparison to the other runs, the number of extensions still varies between 0 and 50 (see

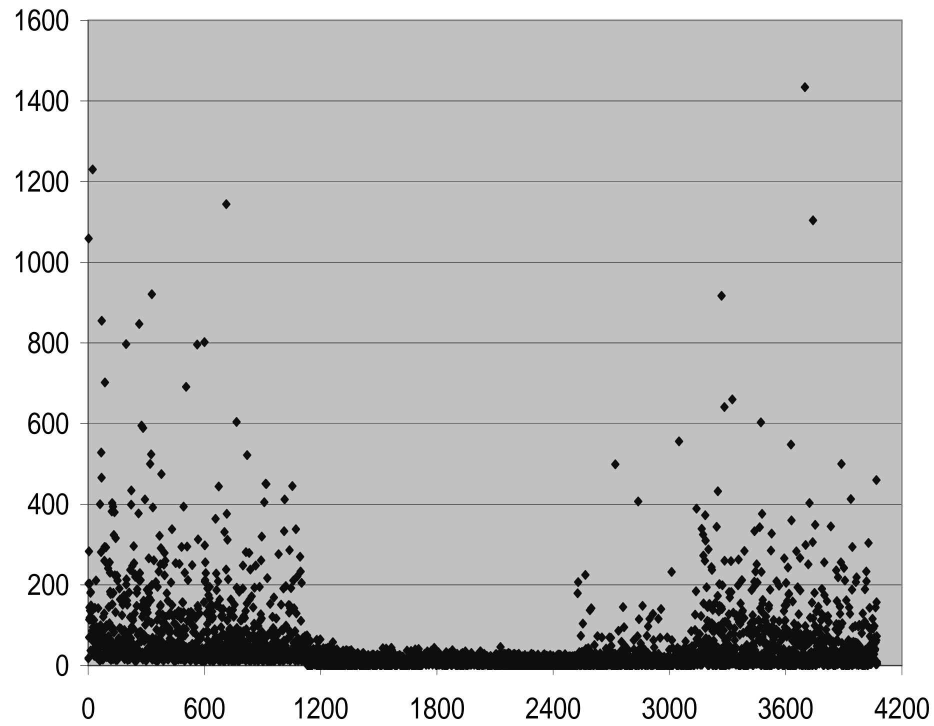

Figure 6.2: Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *print_tokens*

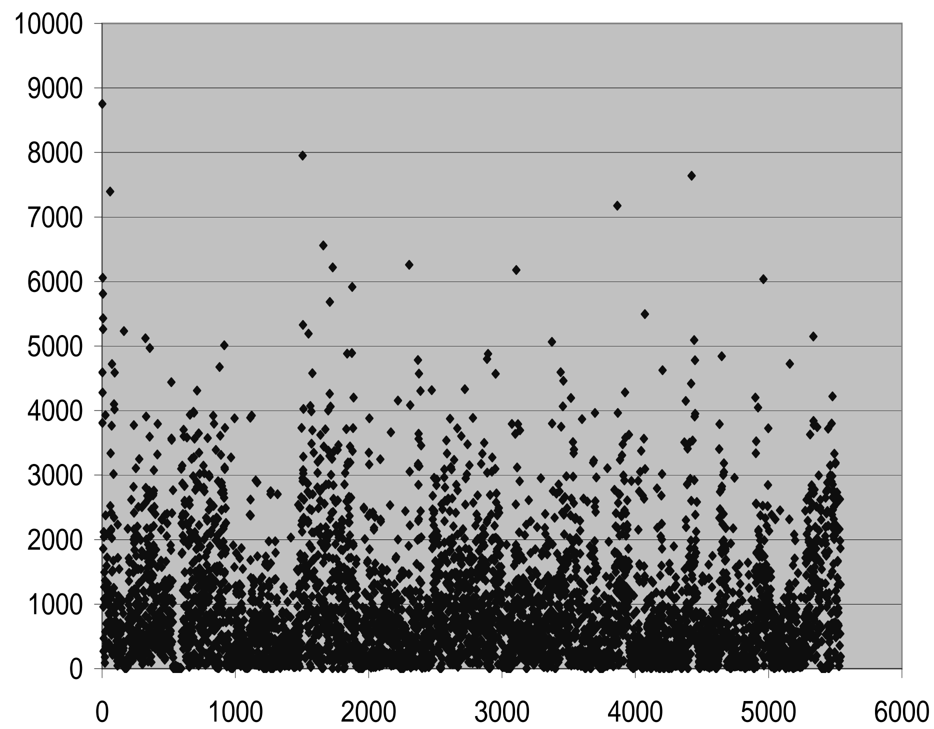

Figure 6.3: Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *replace*

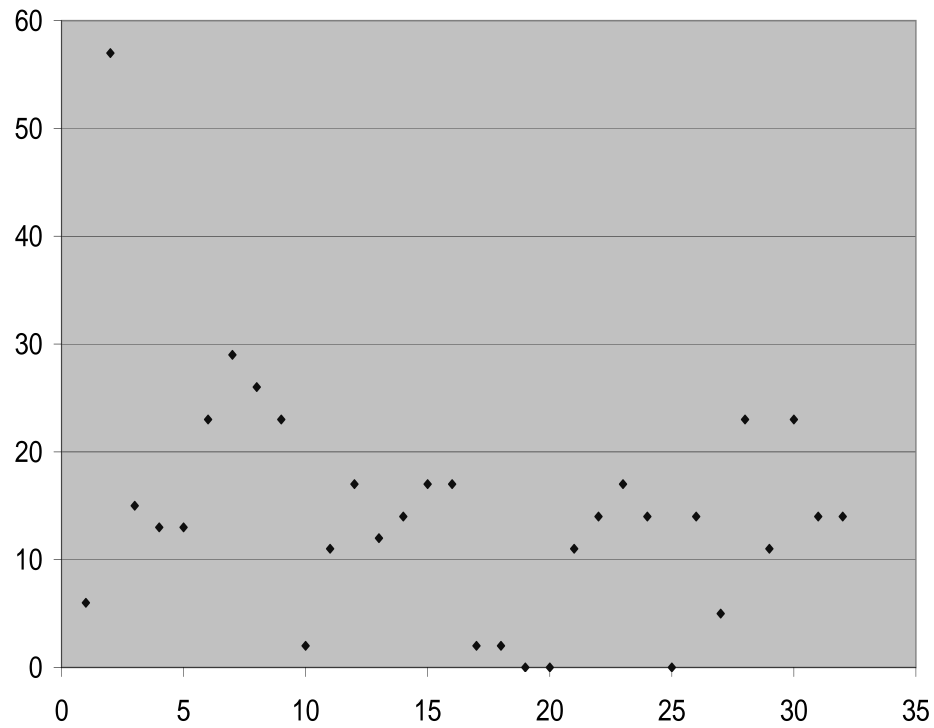

Figure 6.4: Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *recognize*

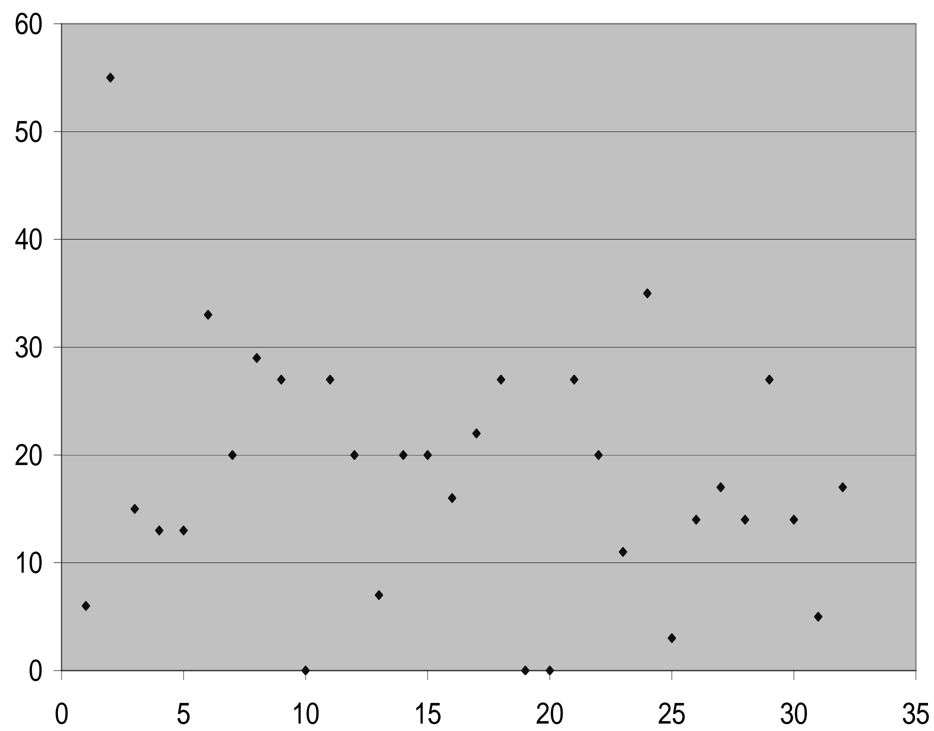

Figure 6.5: Number of extensions (on the y axis) made by each run as it was added to the existing model for the correct version of *recognize2*

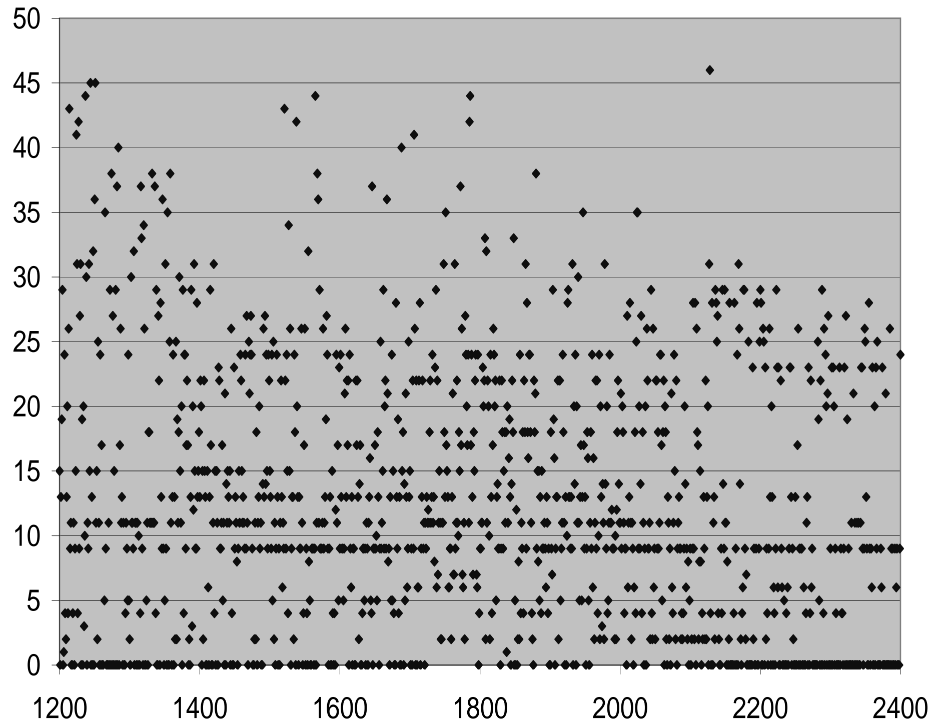

Figure 6.6: Number of extensions (on the y axis) made by each run (1200 - 2400) as it was added to the existing model for the correct version of *print_tokens*

figure 6.6). Twenty or thirty extensions that are not clustered around one or two program locations in a program with 18 functions (36 program locations) is too many to be useful for a user.

These results show that value sets are not useful for debugging in *Carrot's* methodology. The amount of noise produced by adding a good run to an existing model shows that any interesting differences a faulty run produces will be lost in the noise that adding any run produces.

6.2 Relational Potential Invariants

6.2.1 Is adding another good run to the model unlikely to cause invalidations?

To determine if adding good runs to the model would cause invalidations, we again examine the correct versions of *tcas*, *print_tokens*, *replace*, *recognize*, and *recognize2*. We count the number of remaining relational potential invariants after each good trace has been added to the model. The graphs of these counts appear in figures 6.7, 6.8, 6.9, 6.10, and 6.11. Figure 6.12 offers more detail about the first 200 runs for *print_tokens*.

The graphs show that the number of remaining relational potential invariants drops quickly as the first runs were added. The rate at which the relational potential invariants are invalidated also drops quickly. All programs examined reach a stable state. Once that state was reached, adding a

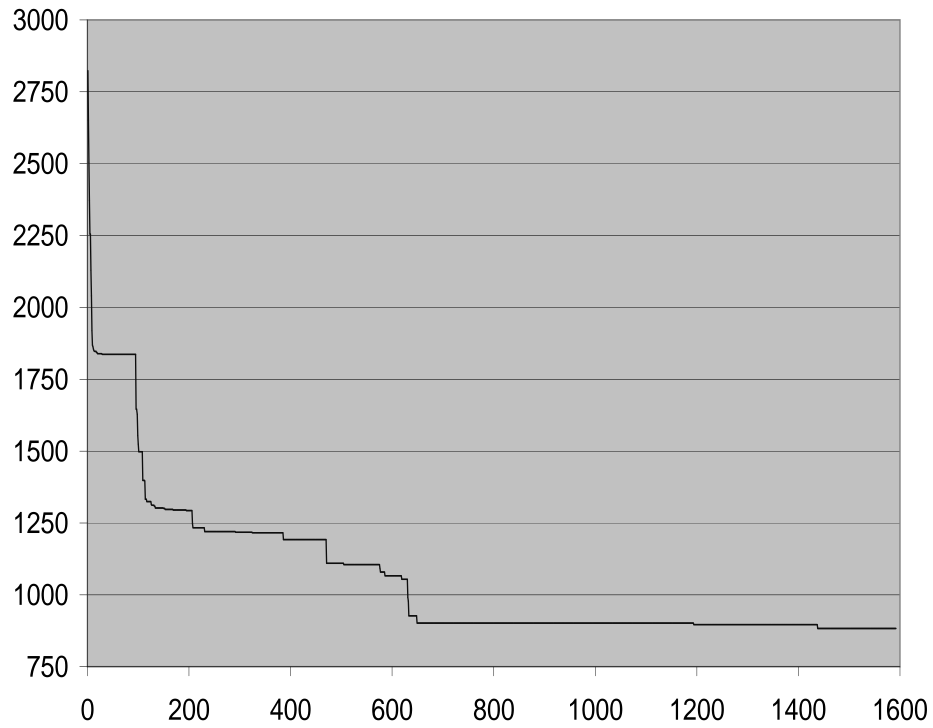

Figure 6.7: Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *tcas*

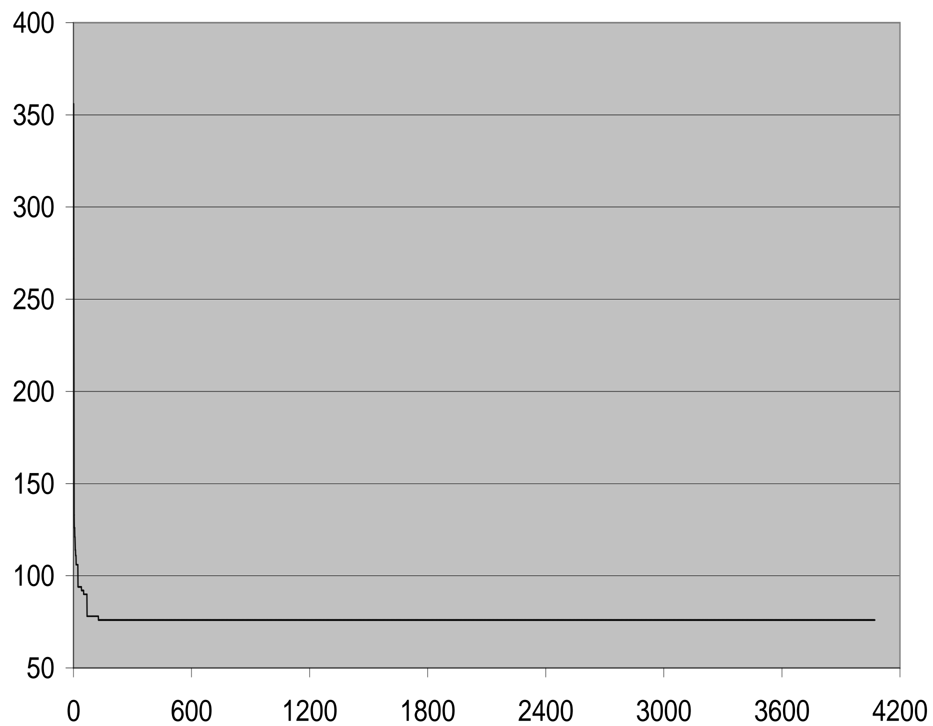

Figure 6.8: Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *print_tokens*

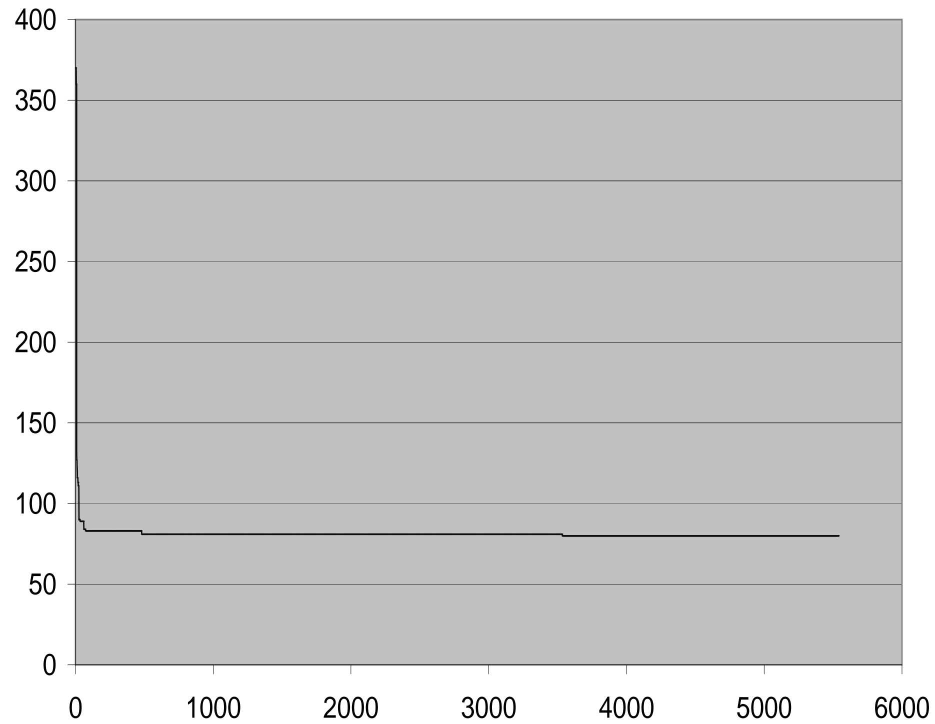

Figure 6.9: Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *replace*

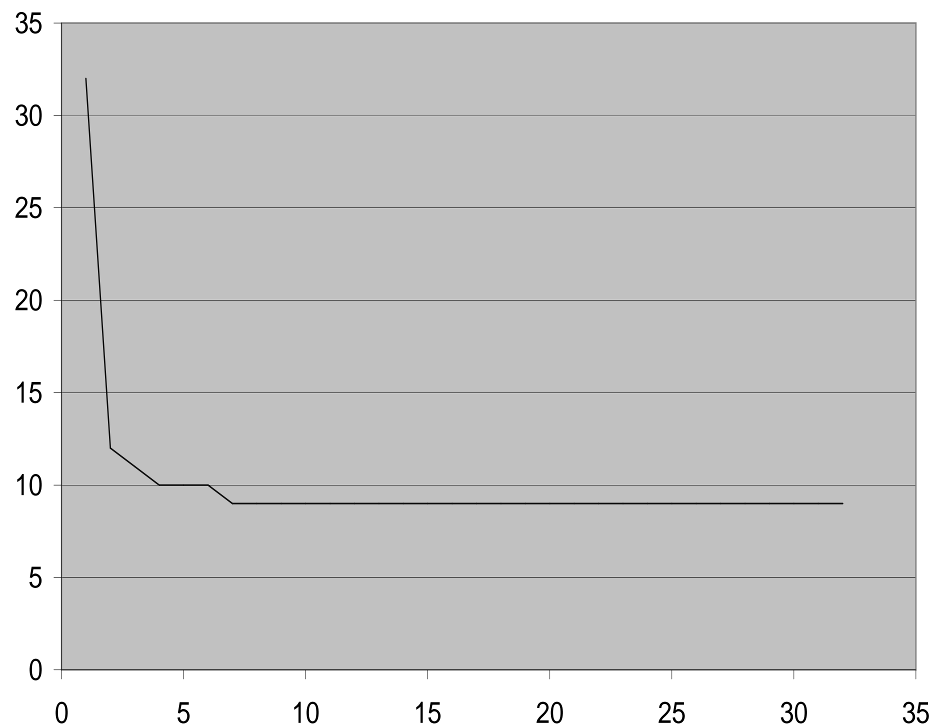

Figure 6.10: Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *recognize*

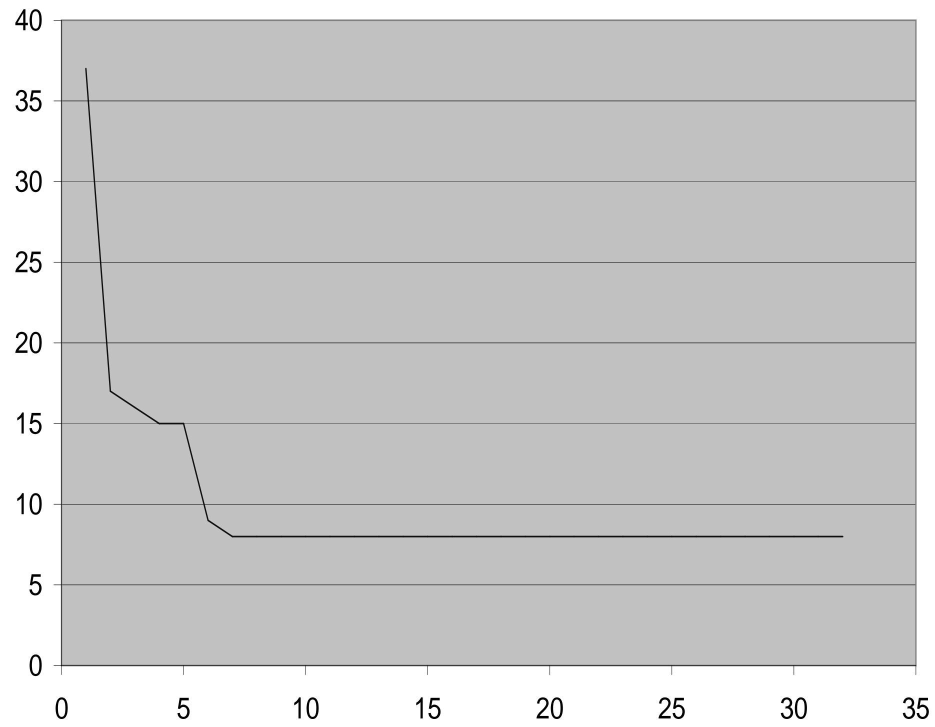

Figure 6.11: Number of potential invariants remaining (on the y axis) as the number of runs included in the model increases (on the x axis) for the correct version of *recognize2*

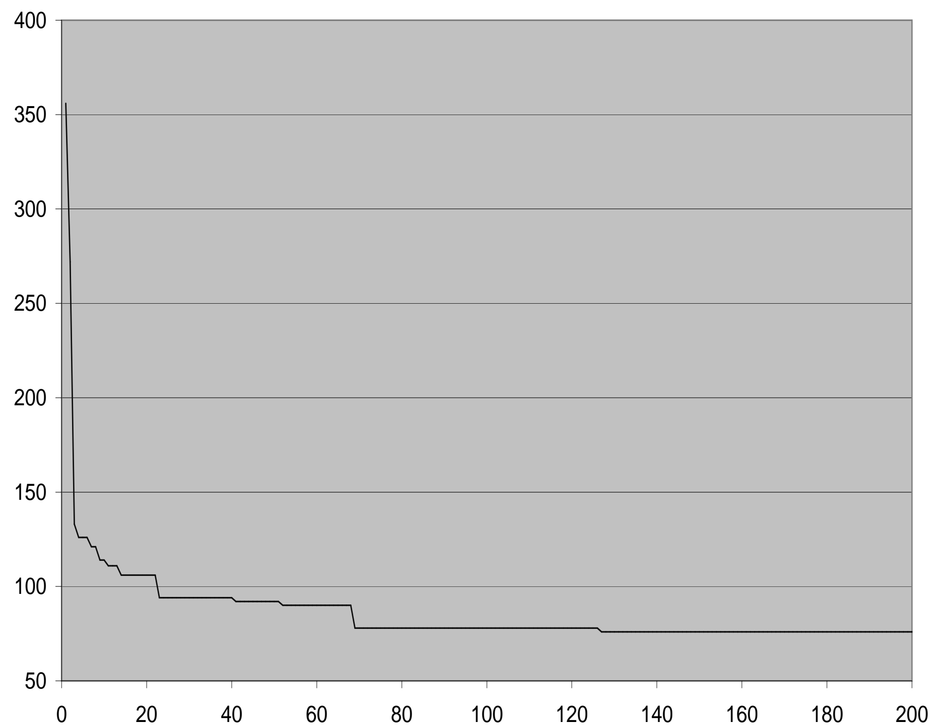

Figure 6.12: A closer look at the first 200 runs of *print_tokens*

Program	Range	Traces with invalidations	Total Traces
tcas	0	0	568
recognize	0-1	17	89
recognize2	0	0	27
print_tokens	0-4	50	532

Table 6.1: The second column shows the range in the number of invalidations that occur. For example, 0-4 in *print_tokens* means that at least one run causes 0 invalidations and at least one run causes 4 invalidations. The third column shows the number of traces, across all versions that of a program, that cause at least one invalidation. The fourth column shows the total number of traces that exist across all versions.

Version	Traces with invalidations	Total Traces
1	0	10
2	0	6
3	6	25
4	0	6
5	0	1
6	0	18
7	5	7
8	6	6
9	0	6
10	0	4

Table 6.2: The invalidations for *recognize* for each faulty version.

good run to the model is unlikely to cause any invalidations.

6.2.2 Is adding a bad run likely to cause invalidations?

Since adding good runs to a stable model is unlikely to cause new invalidations, the next question to answer is whether a bad run will violate the model and cause invalidations. Relational potential invariants are useless for trying to debug *tcas* and *recognize2* (see Table 6.1). No faulty run for either of these programs causes a single invalidation.

Recognize Results

Table 6.2 shows that some of the faulty runs for *recognize* cause invalidations. Versions 1, 2, 4, 5, 6, 9, and 10 do not have any faulty runs which cause invalidations. The relational potential invariants that *Carrot* uses cannot help debug these versions of *recognize*.

Version 3 has 6 runs that cause the same invalidation which is related to the bug in the code. The invalidation reported is that, at the exit of `isDone`, `c+return` no longer equals 98 (see Appendix B.2 for the faulty version). The trouble is that only 6 of 25 runs triggered this invalidation. The

Versions	Traces with invalidations	Total traces
1-7	2	484
8	48	48

Table 6.3: The invalidations for *print_tokens* for each faulty version.

Correct

```
static int keyword(int state)
{
      switch(state)
      {
          case 6 : return(LAMBDA);
          case 9 : return(AND);
          case 11: return(OR);
          case 13: return(IF);
          case 16: return(XOR);

          default: fprintf(stdout, "error\n");
          break;
      }
      exit(0);
}
```

Faulty Version 8

```
static int keyword(int state)
{
      switch(state)
      {
          case 6 : return(LAMBDA);
          case 9 : return(AND);
          case 11: return(OR);
          case 13: return(IF);
          default: return(XOR);

      }
      exit(0);
}
```

Table 6.4: The code for `keyword` in the correct version of *print_tokens* and the faulty version 8

goal of *Carrot* is to detect the bug given any single faulty run for a program, not a subset of the possible faulty runs.

Version 7 has 5 faulty runs which cause invalidations. The invalidation is the same as is triggered by the faulty runs in version 3. In this case, the invalidations are only distantly related to the bug in the code and would not quickly direct the user to the correct program location (see Appendix B.3 for the faulty version).

Version 8 has 6 faulty runs, all of which cause the same invalidation. The counter example is that `return<c` is not true. The invalidation again is distantly related to the bug in the code, but would not guide the user toward finding the bug (see Appendix B.4 for the faulty version).

Print_tokens Results

Table 6.3 shows that invalidations do not offer any help for versions 1 through 7 of *print_tokens*. Only two faulty runs produced any invalidations and neither of the invalidations caused were related to the bug in the program. Version 8 is a unique case. All 48 faulty runs had invalidations and all of the invalidations were related to the bug. The important function is shown in table 6.4. At the program location for `default: return(XOR)`, the model for all good runs states that `state = return` and `state+XOR=32` (`XOR` is defined as 16). Both of these relational potential invariants are invalidated when other values are passed to keyword. This was the one instance where relational potential invariants were invalidated and the invalidations pointed to the bug in the program.

6.3 Analysis

Value sets are useless in our framework because adding any run, buggy or not, causes too many extensions making it difficult for the user to find the extensions that are related to the bug. For the most part, relational potential invariants are also not useful in our framework. One version of one program out of 68 versions examined had a bug that the relational potential invariants found.

A reasonable objection to these results is that *Carrot* does not have the correct relational potential invariants to identify the problems in the buggy versions of programs.We used *tcas* as a case study to determine if there were other simple relational potential invariants that we could create to catch bugs in those versions. Relational potential invariants could be written to capture the bug, but the only by first identifying the bug and then writing the potential invariant schemata to catch the bug. The potential invariant schemas needed to catch the *tcas* bugs would have to be very large, on the order of seven variables, and would essentially replicate the code. See appendix D for examples of the complexity of the bugs involved. Further, *daikon* continues to allow users to create domain specific potential invariant schemas. This suggests that if a simple, common, interesting set of potential invariants exists, they have not yet been found.

Chapter 7

Nascent Work

This section describes some of the approaches currently being examined.

7.1 Using Temporal Information:The Map Back Program

The number of extensions to value sets that *Carrot* reports appears overwhelming. Manos Renieris and I hypothesized that combining temporally distinct differences for a program location was causing problems. To solve this problem, we wrote the map back program. The differences produced by the invariant differencer and the bad trace which produced them are given to the map back program which aligns the differences to the runtime points in the trace file at which the differences could have occurred. We hypothesized that by organizing the differences temporally, a cascade of differences would appear. A single difference would appear at some function entry or exit and all other differences would appear after and be caused by the initial difference.

7.1.1 The Map Back Program Details

The map back program takes the output produced by the invariant differencer and the trace file which produced the differences reported and produces an output listing the differences at the trace points at which they could have occurred. The map back program steps through each of the trace points listed in the trace file it was given. At each runtime point, it checks if there are any differences at the program location corresponding to that trace point. It performs this check by determining if any of the values of the variables in scope at that trace point would have produced the observed difference. For example, if a difference was that for function `f`, $x < y$ was invalidated and this instance of a call to `foo` had values for x and y of 5 and 4, then this call to `f` could have invalidated $x < y$.

7.1.2 The Map Back Program Results

We hypothesized that the map back program would demonstrate a cascade of differences between the good runs and the bad run. We were not able to find any instances of a cascade of differences by manual examination. The flood of information simply increased. A single difference at a program location may appear at each trace point associated with that static program point.

7.2 More Analysis using Temporal Information

The differences, as organized by the map back program, are still too numerous to be of use to a programmer or to find the beginning of a cascade of differences. The map back and remove duplications program was created to attempt to simplify the output of the map back program. The temporally aligned differences from the map back program are given to the map back and remove duplications program which reports only the temporally first instance of each difference.

7.2.1 The Map Back and Remove Duplications Program Details

The map back and remove duplications program takes the output of the map back program and removes all but the temporally first instance of a difference. The map back and remove duplications program examines each trace point listed in the output of the map back program. For each trace point, it checks whether the difference listed has occurred previously. If it has, it is removed from the report. If it is the first instance of this difference, it is left in the report and is added to a list of previously reported differences. The map back and remove duplications program reduced the number of differences reported to the user.

One impact of the map back and remove duplications program is that *Carrot* is no longer conservative. Before removing duplications, if the buggy function had any difference detectable by *Carrot*, it would be tagged as potentially buggy. The possibility now exists that *Carrot* will fail to tag the buggy function because the difference that marked the buggy function had occurred in an earlier function.

7.2.2 Results for The Map Back and Remove Duplications Program

The removal of duplications helps to reduce the information load on the user. Fewer differences allows the user to focus more attention on the differences reported. The cascade of differences still does not appear, but in some cases, the number of differences is small enough to perhaps be useful to the user. Even with the reduced number of differences, a problem still exists: the differences often are not clearly tied to the bug in the code.

There are three sources of the problem of linking differences reported by the map back and remove duplications program to locations of bugs in code. The value sets are affected differently by the removal of duplicate differences than relational potential invariants were. In a program like *tcas*, the value set differences all appear as arguments to main. While the bug may be several function

calls deep, the value set differences are reported at the top level function. In programs which use data sources other than program arguments as their inputs, the differences may be reported closer to the buggy function. However, the difference probably will appear for the function which makes the system call to read the next item in the data source. Unless the buggy function happens to be the function making the system call, the difference will appear under a different function. For all potential invariants, the other source of difficulty is that the differences provide no information about whether the marked function contains a bug or is being used incorrectly.

Chapter 8

Future Work

The current results for *Carrot* are not promising. There are several avenues to pursue that may improve *Carrot*'s performance.

The first avenue is to improve the resolution of the instrumentation. The function level information does not give enough detail about how the program runs to be effective. By instrumenting the code at a basic block level, the information might be detailed enough to make *Carrot* useful.

A second approach to improving the map back program approaches is to organize the differences using a dynamic control flow graph. Structuring the differences this way may show the cascade of differences for which we are looking. Mapping the invalidations on a dynamic control flow graph may illuminate paths along which there are many invalidations.

A third approach is to use multiple faulty runs. The amount of noise presented to the user may be reduced by first finding the differences that each faulty run causes and then intersecting those differences. Manually examining differences of many bad runs for a single version of a program suggests that the interesting differences often are those that are caused by each faulty run.

An approach that could be used with the previous idea is to reintroduce confidence levels for the differences to aid the programmer in prioritizing the differences to examine. Both *diduce*[2] and *daikon*[1] use confidence levels to prioritize reports to the users. A similar scheme might be useful for *Carrot*.

Another approach is to automatically build more complex relational potential invariants out of simple relational potential invariants as they become invalidated. For example, suppose at a program location that $x < y$ and $z + w = 10$ are invalidated by the same run. Two new relational potential invariants could be created, $(x < y) \rightarrow (z + w = 10)$ and $(z + w = 10) \rightarrow (x < y)$. Using a process like this might allow *Carrot* to automatically create the complex relational potential invariants that some programs may need to be debugged.

Lastly, one promising approach is to contrast the faulty runs not with the full set of correct runs, but with some runs which we have established are similar to the faulty run by some orthogonal means. In particular, it would be useful to contrast a faulty run with successful runs that have similar control flow.

Chapter 9

Conclusion

I presented a methodology for automatically debugging programs. Using many correct runs, *Carrot* builds a model of all of the correct runs. *Carrot* then compares a single faulty run against the model and locates places where the run violates the model.

Carrot builds the model out of potential invariants. Potential invariants can be divided into two types, value sets and relational potential invariants. Value sets offer a very precise measure of what occurs in a run but notice too many violations of the model to be useful. Relational potential invariants are a more abstract representation of what runs do. When a faulty run violates the model by invalidating a relational potential invariant, the invalidation often illustrates the bug. Unfortunately, it is extremely rare for a relational potential invariant to be invalidated by a faulty run.

Carrot is currently fails to be a useful debugging tool. It identifies one bug out of sixty-eight versions examined. Value sets are not useful as long as only a single faulty run is considered because any run, correct or faulty, produces too much noise. Value sets may be useful if information from multiple faulty runs is used. Relational potential invariants can be useful for debugging, though only one example of their usefulness was found. More complex relational potential invariants are needed to debug most programs.

Appendix A

Function Relationships

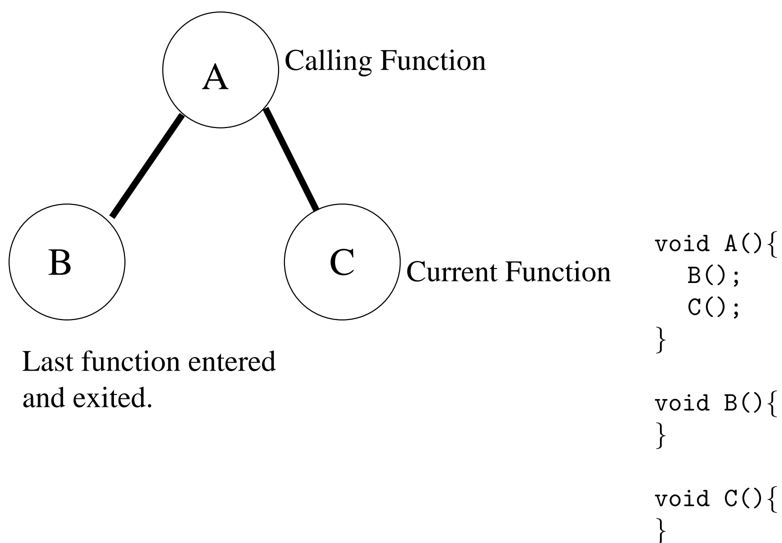

Table A.1: The call structure of the code on the right is pictured in the graph on the left. The labels for each node are for when the current function is `C`.

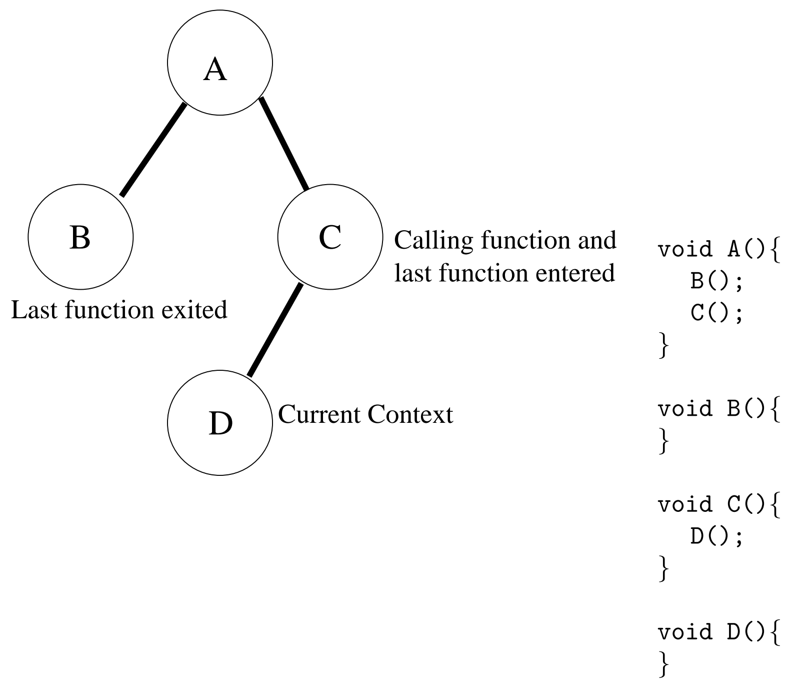

Table A.2: The call structure of the code on the right is pictured in the graph on the left. The labels for each node are for when the current function is `D`.

Appendix B

Recognize

B.1 Correct Recognize with Faults of Versions Commented

```
char isDone(char c);
char checkA(char c);
char checkTwo(char, char* str);

main(int argc, char **argv)
{
  int i = 0;
  char * str;
  char subs[2];

  /* Version 5 checks for (argc < 1). */
  if(argc < 2){
    printf("1\n");
    exit(0);
  }

  str = argv[1];

  while(!isDone(str[i])){
    if(checkA(str[i])){
      i+=1;
      subs[0] = str[i];
      subs[1] = str[i+1];
      /* Version 9 replaces || with &&. */
```

```
        if((checkTwo('b',subs))||(checkTwo('c',subs))){
          /* Version 4 increments i by 1 instead of 2. */
          i+=2;
        } else {
          /* Version 6 prints 1 instead of 0. */
          /* Version 10 removes the following two lines. */
          printf("0\n");
          exit(1);
        }
      } else {
        /* Version 7 prints 1 instead of 0. */
        printf("0\n");
        exit(1);
      }
  }
  /* Version 8 prints 0 instead of 1. */
  printf("1\n");
  exit(0);
}

char isDone(char c){
  /* Version 3 replaces == with !=. */
  return (c == '\0');
}

char checkA(char c){
  /* Version 2 replaces == with !=. */
  return (c == 'a');
}

char checkTwo(char c, char* str){
  /* Version 1 replaces && with ||. */
  return ((str[0]==c) && (str[1]==c));
}
```

B.2 Faulty Recognize Version 3

The bug is that != in `isDone` should be ==.

```
char isDone(char c);
char checkA(char c);
char checkTwo(char, char* str);

main(int argc, char **argv)
{
  int i = 0;
  char * str;
  char subs[2];

  if(argc < 2){
    printf("1\n");
    exit(0);
  }

  str = argv[1];

  while(!isDone(str[i])){
    if(checkA(str[i])){
      i+=1;
      subs[0] = str[i];
      subs[1] = str[i+1];
      if((checkTwo('b',subs))||(checkTwo('c',subs))){
        i+=2;
      } else {
        printf("0\n");
        exit(1);
      }
    } else {
      printf("0\n");
      exit(1);
    }
  }
  printf("1\n");
  exit(0);
}
```

```
char isDone(char c){
  /*!= should be ==*/
  return (c != '\0');
}

char checkA(char c){
  return (c == 'a');
}

char checkTwo(char c, char* str){
  return ((str[0]==c) && (str[1]==c));
}
```

B.3 Faulty Recognize Version 7

The bug is that next to last `printf` in main should print 0, not 1.

```
char isDone(char c);
char checkA(char c);
char checkTwo(char, char* str);

main(int argc, char **argv)
{
  int i = 0;
  char * str;
  char subs[2];

  if(argc < 2){
    printf("1\n");
    exit(0);
  }

  str = argv[1];

  while(!isDone(str[i])){
    if(checkA(str[i])){
      i+=1;
      subs[0] = str[i];
      subs[1] = str[i+1];
      if((checkTwo('b',subs))||(checkTwo('c',subs))){
        i+=2;
      } else {
        printf("0\n");
        exit(1);
      }
    } else {
      /*Should print 0 instead of 1*/
      printf("1\n");
      exit(1);
    }
  }
  printf("1\n");
  exit(0);
```

```
}

char isDone(char c){
  return (c == '\0');
}

char checkA(char c){
  return (c == 'a');
}

char checkTwo(char c, char* str){
  return ((str[0]==c) && (str[1]==c));
}
```

B.4 Faulty Recognize Version 8

The bug is that the last `printf` in main should print 1, not 0.

```
char isDone(char c);
char checkA(char c);
char checkTwo(char, char* str);

main(int argc, char **argv)
{
  int i = 0;
  char * str;
  char subs[2];

  if(argc < 2){
    printf("1\n");
    exit(0);
  }

  str = argv[1];

  while(!isDone(str[i])){
    if(checkA(str[i])){
      i+=1;
      subs[0] = str[i];
      subs[1] = str[i+1];
      if((checkTwo('b',subs))||(checkTwo('c',subs))){
        i+=2;
      } else {
        printf("0\n");
        exit(1);
      }
    } else {
      printf("0\n");
      exit(1);
    }
  }
  /*This should print 1, not 0*/
  printf("0\n");
  exit(0);
```

```
}

char isDone(char c){
  return (c == '\0');
}

char checkA(char c){
  return (c == 'a');
}

char checkTwo(char c, char* str){
  return ((str[0]==c) && (str[1]==c));
}
```

Appendix C

Recognize2

C.1 Correct Recognize2 with Faults of Versions Commented

```
bool isDone(char str);
bool checkA(char str);
bool checkBB(char* str);
bool checkCC(char* str);

main(int argc, char **argv)
{
  int i = 0;
  char * str;

  char subs[2];

  if(argc < 2){
    printf("1\n");
    exit(0);
  }

  str = argv[1];

  while(!isDone(str[i])){
    if(checkA(str[i])){
      i+=1;
      /* Version 4 sets subs[0] equal to str[i+1].*/
      subs[0] = str[i];
```

```
      /* Versions 2 sets subs[1] equal to str[i].*/
      subs[1] = str[i+1];
      if((checkBB(subs))||(checkCC(subs))){
        i+=2;
      } else {
        printf("0\n");
        exit(1);
      }
    } else {
      printf("0\n");
      exit(1);
    }
  }
  printf("1\n");
  exit(0);
}

bool isDone(char c){
  return (c == '\0');
}

bool checkA(char str){
  return (str == 'a');
}

bool checkBB(char* str){
  bool ans = true;
  if(str[0]=='\0'){
    return false;
  }
  switch(str[0]){
  case 'a':
    ans = false;
    break;
  case 'b':
    ans = true;
    break;
  case 'c':
    ans = false;
```

```
    break;
  default:
    ans = false;
    break;
  }

  switch(str[1]){
  case 'a':
    ans = false;
    break;
  case 'b':
    ans = ans;
    break;
  case 'c':
    ans = false;
    break;
  default:
    ans = false;
    break;
  }
  return ans;
}

bool checkCC(char* str){
  bool ans = true;
  if(str[0]=='\0'){
    /* Version 8 returns true instead of false. */
    return false;
  }
  switch(str[0]){
  case 'a':
    ans = false;
    break;
  case 'b':
    ans = false;
    break;
  /* Version 1 removes the c case and has the default case set ans to true. */
  /* Version 1 also changes the switch statement below.*/
  case 'c':
```

```
    ans = true;
    break;
  default:
    /* Version 6 sets the value of ans to true. */
    /* Version 7 sets the value of ans to true. */
    /* Version 7 also makes a change to the default case below. */
    ans = false;
    break;
  }

  switch(str[1]){
  case 'a':
    ans = false;
    break;
  case 'b':
    /* Version 9 sets ans equal to ans. */
    ans = false;
    break;
  /* Version 1 removes the c case and has the default case set ans to ans. */
  /* Version 1 also changes the switch statement above.*/
  case 'c':
    /* Version 3 sets ans to true. */
    ans = ans;
    break;
  default:
    /* Version 5 sets ans to ans. */
    /* Version 7 sets ans to ans. */
    /* Version 7 also makes a change to the default case above. */
    ans = false;
    break;
  }
  return ans;
}
```

Appendix D

Examples of Bugs in tcas in the Siemens Suite

D.1 tcas Faulty Version 1

To catch the bug in version 1 of *tcas* the relational potential invariant would have to keep track when `Inhibit_Biased_Climb` was larger than `Down_Separation`. `Inhibit_Biased_Climb` depends on the values of `Up_Separation` and `Climb_Inhibit`. When `Inhibit_Biased_Climb` is larger than `Down_Separation`, the relational potential invariant must also track when `Own_Below_Threat` is true and `Down_Separation` equals `ALIM`. `Own_Below_Threat` depends on the relationship between `Own_Tracked_Alt` and `Other_Tracked_Alt` while `ALIM` depends upon `Alt_Layer_Value`.

D.2 tcas Faulty Version 2

A relational potential invariant to find this bug would have to know that `Climb_Inhibit` had never been true when the difference between `NOZCROSS` and `MINSEP` is large enough enough so that adding each to `Up_Separation` produces one sum which is less than `Down_Separation` and the other which is larger than `Down_Separation` at a time when the result of `Inhibit_Biased_Climb` had an impact on the result of the function, which is itself a complex predicate. The result of `Inhibit_Biased_Climb` matters when the result of `Non_Crossing_Biased_Climb` matters and `!(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())))` has a different truth value than `Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM())`.

D.3 tcas Faulty Version 13

The relational potential invariant would need to track that the value of `Own_Tracked_Alt_Rate` is never between 600 and 700 at the same time that `High_Confidence` is true and `Cur_Vertical_Sep` is

greater than 600 and that `enabled` (the result of conjoining the three previous predicates together) matters in the result of the program. `Enabled` matters when `tcas_equipped` and `intent_not_known` are true or `tcas_equipped` is false.

Bibliography

[1] M. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, 2000.

[2] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. May 2002.

[3] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 83–90, 1998.

[4] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *International Symposium on Software Testing and Analysis*, pages 135–145, 2000.

[5] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200. Sorento, Italy, may 1994.

[6] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, november 2002.