Error Explanation and Fault Localization with Distance Metrics

Alex David Groce

March 2005

CMU-CS-05-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Edmund Clarke, Chair
Reid Simmons
David Garlan
Willem Visser, NASA Ames Research Center

This research was sponsored by the National Science Foundation (NSF) under grant nos. CCR-0098072, CCR-0121547, CCR-9803774 and through an NSF Graduate Fellowship. The Siemens Industrial Affiliates Program also provided funding related to this research. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government or any other entity.

Keywords: Formal methods, model checking, fault localization, automated debugging, distance metrics, bounded model checking

For my parents.

Abstract

When a program's correctness cannot be verified, a model checker produces a counterexample that shows a specific instance of undesirable behavior. Given this counterexample, it is up to the user to understand and correct the problem. This can be a very difficult task. The error may be in the specification, the environment or modeling assumptions, or the program itself. If the error is determined to be real, the *fault localization* problem remains: before the problem can be corrected, the faulty portion of the code must be identified. Industrial experience and research show that debugging is a time-consuming and difficult step of development even for expert programmers. The counterexample provided by a model checker does not provide sufficient information to ease this task. Counterexample traces can be very long and difficult to read, and often include hundreds or potentially even thousands of lines of code unrelated to the error.

Error explanation is the effort to provide automated assistance in moving from a counterexample to a correction for an error. Explanation provides information as to the *cause* of an error and includes *fault localization* by indicating likely problem areas in the source code or specification.

This work presents a novel and successful approach to error explanation. The approach is based on distance metrics for program executions. The use of distance metrics is inspired by the *counterfactual* theory of causality proposed by philosopher David Lewis, and the insights gained from previous work on providing practical error explanation.

Acknowledgments

It is traditional at this point to note that while a tremendous amount of gratitude is due to the folks mentioned here, any flaws in this work are the unaided product of the author. Far be it from me to deviate from a wise and correct tradition.

My advisor, Edmund Clarke, has provided support, encouragement, and (of course) advice throughout my graduate school career. I thank him for all of his efforts and ideas, and for making the Clarke group such a consistently wonderful place to do model checking research that I almost wish I didn't have to graduate. I should also at this point thank Martha for taking care of all of us in the group, including Ed.

I would also like to thank the other members of my thesis committee – David Garlan, Reid Simmons, and Willem Visser – for their feedback and support. In particular, I would like to thank Willem for the many brainstorming sessions at Ames during the first summer we looked into the idea of explaining counterexamples, and for encouraging me to see how far error explanation could be taken.

Robert O'Callahan first directed my attention in a serious way to Andreas Zeller's delta debugging work; if not for that, it is very possible that this thesis would concern something other than error explanation[1].

On that note, I would like to thank Andreas Zeller very much for encouraging this work, which began as an attempt to produce a model-checking based fault localization technique half as nifty as delta debugging (and its many variations). In particular, Andreas and the other participants at the December 2003 Dagstuhl on Understanding Program Dynamics provided many valuable insights into how to improve (and present) this work.

[1]To determine if that's true, we might use a distance metric on possible worlds and Lewis' counterfactual theory: in the closest possible world to ours in which Rob never mentioned Zeller's work to me, what is my thesis topic?

The `explain` tool would not exist if not for the patience, intelligence, and programming wizardry of Daniel Kroening. There really is no `explain` tool, only a series of rickety hacks built on the solid foundation of CBMC. The `explain` GUI is thanks to Flavio Lerda. Sagar Chaki provided a similar foundation in the MAGIC tool for abstract error explanation.

Ofer Strichman, Daniel, Sagar, and Flavio are responsible for many essential suggestions and insights. Helmut Veith, Prasana Thati, Natasha Sharygina, and Anubhav Gupta also offered useful comments and critiques.

Manos Renieris' ideas, code, and expert assistance were indispensable in the empirical evaluation portion of this work. Fadi Aloul tuned, debugged, and added features to the pseudo-Boolean solver PBS at my request in a truly obliging manner. The tools in this thesis wouldn't work at all without Fadi's help, and it would have been very difficult to make the case that they work without Manos' insights.

Dimitra Giannakopoulou, Jamie Cobleigh, Corina Păsăreanu, Charles Pecheur, and Sarfraz Khurshid all provided helpful comments and suggestions while I was at NASA Ames Research Center (and afterward). Dimitra provided most helpful comments on an early draft of the TACAS 2004 paper. Discussions with Thomas Ball, Mayur Naik, Fabio Somenzi, Kavita Ravi, and Michael Ernst contributed in important ways to the development of the ideas that appear in this thesis. I would like to thank GrammaTech, Inc. for help in using CodeSurfer to generate the PDGs for evaluation, and Thomas Reps and Tim Tietelbaum for useful suggestions. Roderick Bloem, Stefan Staber, and ShengYu Shen also provided useful insight into related ideas. Audiences at SPIN, TACAS, FSE, JPL, Microsoft Research, and IBM's T. J. Watson Research Center provided valuable comments and questions about this work — in particular, Gerard Holzmann, Rajeev Joshi, Sriram Rajamani, Manuvir Das, Mark Wegman, and Eran Yahav inspired fruitful ideas. I owe Rance Cleaveland a great debt for having first interested me in model checking, and for having advised me to attend graduate school at Carnegie Mellon.

Being a part of CMU SCS has been a lot of fun. I would like to thank all of the folks here for making life in Pittsburgh enjoyable, and Sharon Burks in particular for reminding me of all the things I kept forgetting I needed to do in order to graduate!

If I begin naming names, I'm sure to leave someone important out, so I'll just say that this bit thanks everyone who deserves thanks: all my great friends, from Starmount High School in good old Yadkin County, from NC State, from CMU, from zephyr (though the last group is generally disordered), and elsewhere. Without y'all, my sanity would have failed long before this thing was finished.

That said, I will explicitly mention those fine souls who loaned me books, critiqued slides, asked me "just what is it you research, anyway?" (and listened to the answer, then asked good questions) or read some portion of the thesis: thanks Josie, Kevin, James, Bruce, Francisco, Pete, Chris, Jeffrey, Benoît, Tom, Pat, William, Andrzej, Darrell, David, Jennifer, Phil, Karl, Neel, and Lauren! Special thanks to Mahim for loaning me his laptop at my thesis defense. If I've forgotten anyone here, I apologize from the bottom of my heart[2].

I must, naturally, thank my family: without the love and support of my mom Carole, my dad Leonard, and my sister Andrea I doubt I'd have ever managed to be accepted in polite society, much less complete a thesis.

Finally, thanks be to God for dappled things, er, for, quite literally, everything.

[2]Listing *everyone* on zephyr who helped out with LaTeX would require more space than I think I have.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Overview of the Approach 4

1.3 Explanation and Causality 6

1.4 Lewis' Counterfactual Theory of Causality 8

1.4.1 Causal Dependence 9

1.5 Error Explanation with Distance Metrics 11

1.5.1 Limitations of the Approach 13

1.6 Narrative Table of Contents 14

2 Background and Related Work **17**

2.1 Philosophical Background 17

2.2 Related Work 19

2.2.1 Error Explanation and Fault Localization in Model Checking . 19

2.2.2 Error Explanation and Fault Localization in Testing 25

2.2.3 Slicing and Counterexample Minimization 27

2.2.4 Error Explanation and Fault Localization in Artificial Intelligence (Model-Based Diagnosis) 29

2.2.5 String and Sequence Comparison 30

2.3 Original Contributions 31

3 Error Explanation with Distance Metrics **33**

3.1 Distance Metrics for Program Executions 33

3.1.1 Representing Program Executions 34

3.1.2 The Distance Metric d 42

3.1.3 Choosing an Unwinding Depth 48

3.2 Producing an Explanation 49

3.2.1 Finding the Closest Successful Execution 50

3.3 Closest Successful Execution Δs and Causal Dependence 56

4 Δ-Slicing **59**

4.1 Motivation 59

4.2 Computing a Δ-Slice 63

4.3 Explaining and Slicing in One Step 69

4.3.1 Motivation 69

4.3.2 Naïve Approach 70

4.3.3 Shadow Variables 71

4.3.4 Disadvantages of One-Step Slicing: The Relativity of Relevance 74

5 Case Studies and Evaluation for Concrete Explanation **77**

5.1 Case Studies 77

5.1.1 TCAS Case Study 78

5.1.2 μC/OS-II Case Study 89

5.2 Evaluation of Fault Localization 92

5.3 Evaluation of Modifications to the Distance Metric 98

5.3.1 Measuring Distance Over Input Changes Only 98

5.3.2 Increasing the Weight for Input Changes 100

5.3.3 Increasing the Weight for Control Flow Changes 101

5.3.4 Allowing Arbitrary Value Changes (Interventions) 102

5.4 Evaluation of One-Step Slicing 103

5.4.1 One-Step Slicing in Action 104

6 Causal Dependence and Explanation 109

6.1 Hypothesizing and Checking Causal Dependence 109

6.1.1 Motivation 110

6.1.2 Algorithm for Checking Causal Dependence 111

6.1.3 Checking Causal Dependence in Practice 116

6.1.4 Alternative Approaches for Hypothesis Selection 117

7 Explaining Abstract Counterexamples 119

7.1 Motivation 119

7.1.1 Predicate Abstraction 121

7.1.2 Motivating Example 125

7.2 Abstract Error Explanation 135

7.3 A Distance Metric for Abstract Executions 138

7.3.1 Alignment 140

7.3.2 The Distance Metric d 143

7.4 Finding a Successful Execution 145

8 Explaining LTL Property Failures 149

8.1 Successful Executions for LTL Properties 149

8.2 Example of LTL Explanation 151

9 Case Studies and Evaluation for Abstract Explanation 165

9.1 Experimental Results 165

9.1.1 Benchmarks 166

9.2 Evaluation of Fault Localization 168

9.3 Benchmark: Mutex Explanation 168

9.4 SSL Explanation 172

9.5 Comparing Concrete and Abstract Explanation 177

9.5.1 Is Abstract Superior to Concrete? . . . 177
9.5.2 Is Concrete Superior to Abstract? . . . 179
9.5.3 Choosing a Distance Metric . . . 181

10 Conclusions **183**
10.1 Conclusions . . . 183
10.2 Future Work . . . 186
10.2.1 SSA and Abstract Explanation . . . 186
10.2.2 Slicing . . . 187
10.2.3 Concurrency . . . 188
10.2.4 Explicit-State Approaches . . . 189
10.2.5 Explanation and Symbolic Execution . . . 190
10.2.6 Metrics for More Complex Counterexample Forms . . . 191
10.2.7 Further Empirical Evaluation and User Studies . . . 191
10.2.8 Automated Program Correction . . . 192
10.3 Summary . . . 194

Bibliography **195**

A Command Line Options for the `explain` Tool **207**

B TCAS Version #1 Counterexample **209**

C TCAS Version #1 First Explanation **217**

D TCAS Version #1 Counterexample (Post-Assumption) **225**

E TCAS Version #1 Second Explanation **233**

F μC/OS-II Counterexample **243**

G μC/OS-II Explanation **253**

List of Figures

1.1 Explaining an error using distance metrics 4
1.2 Causal dependence 10

2.1 A counterexample, a negative, and a positive 21

3.1 minmax.c 36
3.2 Constraints generated for minmax.c 38
3.3 Counterexample for minmax.c 39
3.4 Counterexample values for minmax.ce 40
3.5 Δs for minmax.c and the counterexample in Figure 3.3 52
3.6 Closest successful execution for minmax.c 54
3.7 Closest successful execution values for minmax.c 55
3.8 Δ values ($\Delta = 1$) for execution in Figure 3.6 55

4.1 slice.c 60
4.2 Δ values for slice.c 61
4.3 Partial constraints for slice.c 66
4.4 Δ-slicing constraints for slice.c 67
4.5 Δ-slice for slice.c 68
4.6 Δ-slice for minmax.c 69
4.7 One-step Δ-slicing constraints for slice.c 73

5.1 diff of correct TCAS code and variation #1 79

5.2 First explanation for variation #1 (after Δ-slicing) 79
5.3 Code for violated assertion 80
5.4 Explaining tcasv1.c 81
5.5 Second explanation for variation #1 (after Δ-slicing) 85
5.6 Values removed by Δ-slicing from report 86
5.7 Correctly locating the error in tcasv1.c 87
5.8 Code structure for μC/OS-II error 90
5.9 Explanation for μC/OS-II error 91
5.10 One-step slicing report for TCAS variation #1 105
5.11 Code for determining if RA is computed 106

6.1 sort.c 113
6.2 Counterexample for sort.c 114
6.3 Explanation for sort.c 115
6.4 Causes for sort.c 115
6.5 Causes for TCAS error #1 116

7.1 Error explanation with distance metrics 120
7.2 Counterexample-guided abstraction refinement (CEGAR) 121
7.3 minmax.c 124
7.4 Concrete Δ values for minmax.c 125
7.5 Abstract Δ values for minmax.c 126
7.6 Abstract state space for minmax.c 130
7.7 Abstract counterexample graph for minmax.c 131
7.8 Abstract counterexample for minmax.c. 132
7.9 Abstract successful execution graph for minmax.c 133
7.10 Abstract successful execution for minmax.c 134
7.11 Alignments for executions 141

8.1 Successful executions for LTL properties 152

8.2 locks.c 155
8.3 Counterexample for locks.c 157
8.4 Abstract Δ values for locks.c 160
8.5 Successful execution for locks.c 161

9.1 Incorrect version of mutex code fragment 169
9.2 Correct version of mutex code fragment 169
9.3 Abstract Δ values for mutex lock failure 170
9.4 Mutex locking property 171
9.5 SSL-1 code fragment 172
9.6 Abstract Δ values for SSL-1 173
9.7 Counterexample for SSL-1 174

List of Tables

5.1 Scores for localization techniques 94
5.2 Scores for metric over inputs-only 99
5.3 Scores when interventions are allowed 102
5.4 Scores with one-step slicing 104

9.1 Experimental results for MAGIC examples 167

Chapter 1

Introduction

"Did the Nightmare kill the Venetian painter known as Giancristoforo Doria?"

"The painter you name died at the hands of his callous fellow conspirators. The conditions of his imprisonment in the Arqana destroyed him. He died from a madness inherent within him. He committed suicide. He was killed by sorcery. The Arabian Nightmare took him. His death was determined and more than determined. There are always more causes than events. . ."

- Robert Irwin, The Arabian Nightmare

1.1 Motivation

In an ideal world, given a trace demonstrating that a system violates a specification, a programmer or designer would always be able in short order to identify and correct the faulty portion of the code, design, or specification. In the real world, dealing with an error is often an onerous task, even with a detailed failing run in hand. Debugging is one of the most time consuming tasks in the effort to improve software quality [Ball and Eick, 1996], and locating an error is the most difficult aspect of the debugging process [Vesey, 1985]. This work describes the application of a technology traditionally used for *discovering* errors to the problem of *understanding and isolating* errors.

The goal of this thesis research can be broken down into two related (but not equivalent) tasks, *error explanation* and *fault localization*:

- **Definition 1 (error explanation)**

  Error explanation *describes automated approaches that aid users in moving from a trace of a failure to an understanding of the essence of the failure and, perhaps, to a correction for the problem.*

Error explanation is to some extent a fundamentally psychological problem, and it is unlikely that completely formal proof of the superiority of any approach is possible. By this we do not mean that objective evaluation of explanation methods is impossible: user studies and other measures of increased programmer productivity may well serve to demonstrate the superiority of inferiority of explanation approaches. By "psychological" we rather mean that a logically defensible explanation that does not aid actual programmers in correcting errors is, in some sense, missing the point. The best proof of an explanation is more likely to lie in whether it aids in correcting an error, rather than relying on a formally definable logical evaluation.

- **Definition 2 (fault localization)**

  Fault localization *is the more specific task of identifying the faulty core of a system: which components are responsible for an error. Alternatively,* fault localization *can be thought of as reporting which portions of a system should be* modified *in order to correct an error.*

Fault localization *is* suitable for quantitative evaluation, and we present comparisons of quantitative results to establish the effectiveness of the distance metric approach, based on an evaluation method recently adopted by the fault localization community [Cleve and Zeller, 2005; Renieris and Reiss, 2003]. To some extent, given the very reasonable assumption that determining the location of a fault is a key task in understanding and correcting an error [Vesey, 1985], demonstrating that a method is effective for fault localization provides a strong argument that the method will be useful for the more general task of error explanation. This assumption suggests that on the one hand, error explanation subsumes fault localization, and, on the other hand, that fault localization is the most critical aspect of error explanation. This seems natural, given that an ideal explanation would presumably be a description of the best (from the point of view of correctness and understandability of resulting code) *fix* for an error, and that this would necessarily entail a localization of the code which is to be altered (and thus a very precise localization).

A more concrete way to think about the difference between explanation and localization is that an explanation will likely involve a *story* about causality: "A happens, which causes B to happen, which causes C to happen, which results in

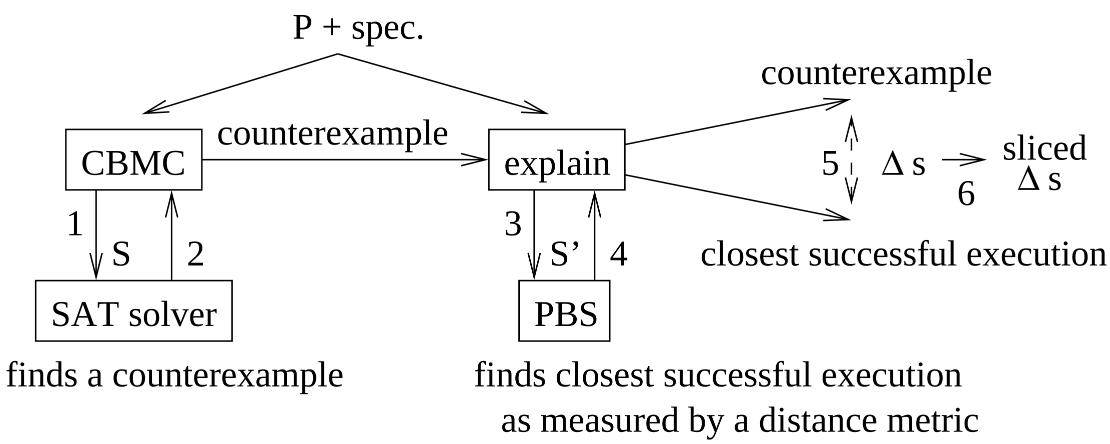


Figure 1.1: Explaining an error using distance metrics

an error.” A localization will have a simpler structure, simply indicating program modules, functions, lines, or expressions as likely to be faulty.

Model checking [Clarke and Emerson, 1981; Clarke et al., 2000b; Queille and Sifakis, 1982] tools explore the state-space of a system to determine if it satisfies a specification. When the system disagrees with the specification, a counterexample trace [Clarke et al., 1995] is produced. This work explains how a model checker can provide error explanation and fault localization information in addition to a counterexample witness, in order to ease the debugging process.

1.2 Overview of the Approach

For a program P, the process (Figure 1.1) is as follows:

1. The bounded model checker CBMC uses loop unrolling and static single assignment to produce from P and its specification a SAT problem, S. The satisfying assignments of S are bounded *executions* of P that violate the specification (counterexamples).

2. CBMC uses a SAT solver to find a counterexample.

3. The `explain` tool produces a propositional formula, S'. The satisfying assignments of S' are executions of P that do *not* violate the specification. `explain` extends S' with constraints representing an optimization problem: find a satisfying assignment that is as similar as possible to the counterexample, as measured by a *distance metric* on executions of P.

4. `explain` uses the PBS solver to find a successful execution that is as close as possible to the counterexample.

5. The differences (Δs) between the successful execution and the counterexample are computed.

6. A slicing step is applied to reduce the number of Δs the user must examine. The Δs are then presented to the user as *explanation* and *localization*.

If the explanation is unsatisfactory at this point, the user may need to add assumptions and return to step 1 (see Section 5). The most important novel contributions of this work are the third, fourth, and sixth steps of this process: previous approaches to error explanation did not provide a means for producing a successful execution guaranteed to be as similar as possible to a counterexample, and lacked the notion of causal slicing.

In Chapter 7, this basic outline will be revisited and generalized, but the essential idea of combining bounded model checking [Biere et al., 1999] with an optimization problem [Aloul et al., 2002] to generate a successful execution that is maximally similar to a given counterexample will remain unchanged.

1.3 Explanation and Causality

There are many possible approaches to error explanation. A basic notion shared by many researchers in this area [Ball et al., 2003; Groce and Visser, 2003; Zeller, 2002] and many philosophers [Sosa and Tooley, 1993] is that to explain an event (e.g. an error trace in a program) is to identify its causes. A second common intuition is that successful executions that closely resemble a faulty run can shed considerable light on the sources of the error (by an examination of the differences in the successful and faulty runs) [Groce and Visser, 2003; Renieris and Reiss, 2003; Zeller and Hildebrandt, 2002].

The sources of the second intuition are probably complex (and, naturally, related to intuitions about causality), though it seems reasonable to expect that the experience of many researchers (and most programmers) in debugging code that is poorly understood is involved: before an error can be understood, some notion of *correct* behavior must be available for comparison. A similar successful execution provides a basis for forming expectations about correct behavior — not only *some* correct behavior of the program, but a correct behavior *relevant* to the erroneous behavior in question.

The idea that explanation is "about" causality, on the other hand, is a fundamental matter of definition: whatever notion of explanation or localization is used appears to relate to the causal structure of the system. Given this basic assumption, a natural approach to error explanation and fault localization is to search for a theory of causality that satisfies certain criteria:

1. The theory should provide a computable definition of causality. Automated localization and explanation cannot rely on a purely psychological notion that cannot be translated into algorithmic terms.

2. In addition to determining whether events are causally related, the theory should be applicable to *finding* causes: given an event, explanation and localization rely on producing hypotheses about causality, rather than simply determining if a given candidate cause agrees with the theory.

3. The theory should also be agreeable to basic intuitions: the results are finally intended to be used by a programmer/designer or verification expert, and should be, at some level, based on principles that the user can understand. Given that explanation and localization is an interactive process, some degree of understanding to allow the user to effectively guide the process is desirable.

The second requirement also makes quantitative evaluation of a method derived from a theory possible: given a fault localization, it is possible to determine its quality and compare to competing methods [Renieris and Reiss, 2003].

1.4 Lewis' Counterfactual Theory of Causality

David Lewis [Lewis, 1973a] has proposed a theory of causality that provides a more formal justification for the second intuition if we assume explanation is the analysis of causal relationships. If explanation is, at heart, about causality, and, as Lewis proposes, causality can be understood using a notion of similarity (that is, a distance metric), it is reasonable to expect that successful executions resembling a counterexample can be used to explain an error.

Following Hume [Hume, 1739, 1748; Sosa and Tooley, 1993] and others, Lewis holds that a cause is something that *makes a difference*: if the cause c had not been, the effect e would not have been. Lewis equates causality to an evaluation based on distance metrics between possible worlds (*counterfactual dependence*) [Lewis, 1973b]. This provides a philosophical link between causality and distance metrics for program executions.

For Lewis, an effect e is dependent on a cause c at a world w iff at all worlds *most similar* to w in which $\neg c$, it is also the case that $\neg e$. Causality does not depend on the impossibility of $\neg c$ and e being simultaneously true of any possible world, but on what happens when we alter w *as little as possible*, other than to remove the possible cause c. This seems reasonable: when considering the question "Was Larry slipping on the banana peel causally dependent on Curly dropping it?" we do not, intuitively, take into account worlds in which another alteration (such as Moe dropping a banana peel) is introduced. This intuition also holds for causality in programs, despite the more restricted context of possible causes: when determining if a variable's value is a cause for a failed assertion, we wish to consider whether changing that value results in satisfying the assertion without considering that there may be some other (unrelated) way to cause the assertion to fail. Distance metrics between possible worlds are problematic, and Lewis' proposed criteria for such metrics have been criticized on various grounds [Horwich, 1987; Kim, 1973; Sosa and Tooley, 1993].

Program executions are much more amenable to measurement and predication than possible worlds. The problems introduced by the very notion of counterfactuality are also avoided: a counterfactual is a scenario *contrary to what actually happened*. Understanding causality by considering events that are, by nature, only hypothetical may make theoretical sense, but imposes certain methodological difficulties. On the other hand, when explaining features of program executions, this aspect of counterfactuality is usually meaningless: any execution we wish to consider is just as real, and just as easily investigated, as any other. A counterexample is in no way privileged by *actuality*.

1.4.1 Causal Dependence

If we accept Lewis' underlying notions, but replace possible worlds with program executions and events with propositions about those executions, a practically applicable definition of causal dependence emerges[1]:

[1] Our causal dependence is actually Lewis' counterfactual dependence.

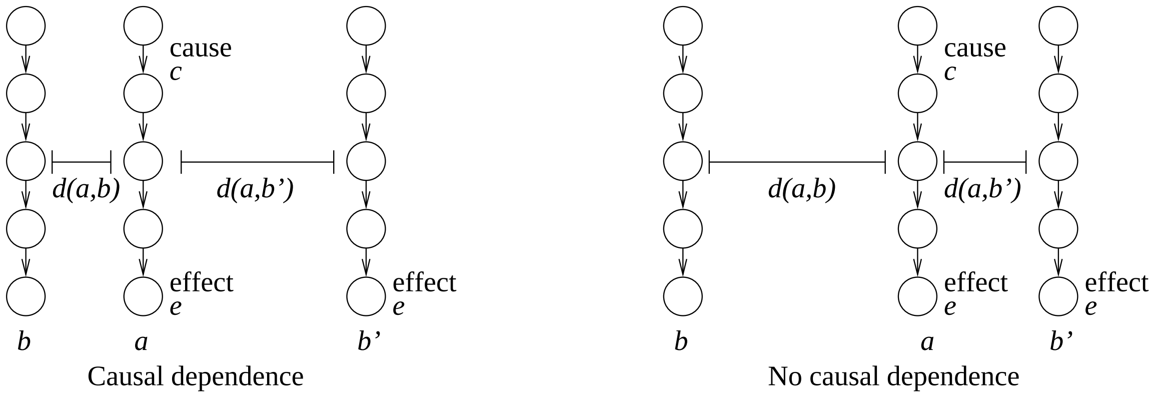


Figure 1.2: Causal dependence

Definition 3 (causal dependence)

A predicate e is causally dependent *on a predicate c in an execution a iff:*

1. *c and e are both true for a (we abbreviate this as $c(a) \wedge e(a)$)*

2. *There exists an execution b such that: $\neg c(b) \wedge \neg e(b) \wedge$*

$$(\forall b' \; . \; (\neg c(b') \wedge e(b')) \Rightarrow (d(a, b) < d(a, b')))$$

where d is a *distance metric* for program executions (defined in Section 3.1). In other words, e is causally dependent on c in an execution a iff executions in which the removal of the cause also removes the effect are more like a than executions in which the effect is present without the cause.

Figure 1.2 shows two sets of executions. In each set, an execution a, featuring both a potential cause c and an effect e, is shown. Also shown in each set is an execution b, such that (1) neither the cause c nor the effect e is present in b and (2) that is as similar as possible to a. That is, no execution which does not feature either

c or e is closer to a than b. Execution b' in each group is, in like manner, as close as possible to a, and features the effect e but not the potential cause c. If b is closer to a than b' is (that is, $d(a, b) < d(a, b')$, as in the first set of executions), we say that e is causally dependent on c. If b' is at least as close to a as b (as in the second set of executions), we say that e is not causally dependent on c.

1.5 Error Explanation with Distance Metrics

This work presents a distance metric that allows determination of causal dependencies and the implementation of that metric in a tool called `explain` [Groce et al., 2004] that extends CBMC [CBMC Website], a model checker for programs written in ANSI C. The focus of the work, however, is not on computing causal dependence, which is only useful *after* forming a hypothesis about a possible cause c, but on helping a user find likely candidates for c. Given a good candidate for c, it is likely that code inspection and experimentation are at least as useful as a check for causal dependence. Lewis' theory provides only a method for determining *if* a candidate is really a cause, not a method for generating candidates in the first place. The philosophical discussion of causality is more relevant to settling disputes about proposed causes than it is to the most important task for error explanation, coming up with likely candidate causes.

The basic approach, presented in Chapter 3 (and outlined in Figure 1.1), is to explain an error by finding an answer to an apparently different question about an execution a: "How much of a must be changed in order for the error e *not* to occur?"

— `explain` answers this question by searching for an execution, b, that is as similar as possible to a, except that e is not true for b. Typically, a will be a counterexample produced by model checking, and e will be the negation of the specification. Section 3.3 provides a proof of a link between the answer to this question about changes to a and the definition of causal dependence. The guiding principle in both cases is to explore the implications of a change (in a cause or an effect) by altering as little else as possible: differences will be relevant if irrelevant differences are suppressed.

It is *these differences* that will give the user a set of *candidate causes* for an error. The key notion is that a cause is *something that makes a difference.* Some counterexample inputs must change their values in order to avoid an error: what happens as a result of these inputs changes which results in the error failing to manifest itself? These behavioral changes should provide a user with insight into why the error appears in the failing execution. Minimizing the distance (i.e., the number of changes with respect to the counterexample) avoids the introduction of irrelevant changes — things that *don't make a difference* are *not* **causes** — and minimizes the amount of information that a user must read in order to start hypothesizing causes. We can expect that if these differences are, in fact, closely associated with the real causes of error, high quality fault localization can be provided by indicating the program source locations of the changes in computed values. Experimental results bear out this indirect indication that changes with respect to a most similar successful execution are valuable in establishing the causes of an error.

1.5.1 Limitations of the Approach

The approach presented is automated in that the generation of a closest successful execution requires no intervention by the user; however, it may be necessary in some cases for a user to add simple assumptions to improve the results produced by the tool. For most of the instances seen in the case studies, this need for intervention is a result of the structure of the property, and the introduction of assumptions to improve the result can, in principle, be fully automated. More generally, however, it is not possible to make use of a fully automated refinement of assumptions, as an explanation can only be evaluated by a human user: there is no independent *objective* standard by which the tool might determine if it has captured the right notion of the incorrectness of an execution, in a sense useful for debugging purposes. In particular, while the specification may correctly capture the full notion of correct and incorrect behavior of the program, it will not always establish sufficient guidance to determine the correct executions that are relevant to a particular failing execution. Assumptions are used, in a sense, to refine the *distance metric* (instead of the specification) by removing some program behaviors from consideration. The frequency of this need is unknown: only one example required the addition of a non-automatable assumption. See Section 5.1.1 for the details of this occasional need for additional guidance.

A more fundamental limitation is that reporting changes to a counterexample with respect to a minimally distant successful execution does not work when a program has *no successful executions*. In this case, the `explain` tool can still be used to check candidate causes generated by some other method for causal dependence (see Chapter 6), but the primary technique presented in this work will not provide explanation or localization. In our experience, the programs to which model checkers are applied typically do not present this problem, as model checking is most useful for finding difficult-to-discover failing executions of programs that behave correctly for most inputs.

1.6 Narrative Table of Contents

You've just finished reading Chapter 1, which presents a high level overview of the goals, philosophical underpinnings, and central approach of the thesis. This chapter also includes the narrative table of contents you are now reading.

Chapter 2 briefly addresses the subject of the philosophy of causality, and describes the large body of more practical work on error explanation and fault localization. Related work in model checking and testing is presented in some detail; model-based diagnosis and program slicing are presented in less detail, as the methods used in these approaches are less directly related to our distance metric based technique.

The heart of the thesis is Chapter 3: if you have time to read only one chapter, make it this one, which presents the basic technique for error explanation and fault localization with distance metrics. In this critical chapter you will read about (1) a distance metric for program executions and (2) an algorithm for finding a successful execution that is as similar as possible to a given counterexample. The ideas presented in Chapter 3 are crucial to the understanding of the subsequent chapters.

Chapter 4 introduces Δ-slicing, an algorithm for removing irrelevant information from an error explanation. This material is the most technically involved section of the thesis. Complete understanding of causal slicing is not essential for a basic grasp of the distance metric based approach, but is likely to reward the reader in pursuit of deeper insights into the relationship between explanation and causality.

The central experimental results of the thesis are presented in Chapter 5, which is therefore essential reading. In addition to comparison with other explanation and localization techniques and discussion of how to quantitatively evaluate fault localization, the chapter provides a look at the practicalities of explanation for real programs.

Chapter 6 is something of a digression; it shows how the `explain` tool can be used to check causal dependence (as defined above) and automatically hypothesize causes for an error. The reader in a hurry is advised to skip this portion of the thesis.

Abstract explanation is the subject of Chapters 7-9. Abstract explanation is a generalization of the technique presented in Chapter 3 to executions in an abstract state space, where each "execution" may represent many possible concrete execution. Abstraction provides an automatic generalization of the differences in executions to *logical* changes — e.g.., "`x <= y` to `x > y`" in place of "`x = 10` to `x = 20`".

Chapter 7 describes how (and why) to compute abstract explanations, following a brief introduction to predicate abstraction and counterexample guided abstraction refinement in software model checking.

Chapter 8 shows how abstract explanation can be applied to explain violations of Linear Temporal Logic specifications, and is not essential reading.

Chapter 9 presents experimental results demonstrating the utility of abstract explanation and compares and contrasts abstract explanation to the “concrete” explanation technique presented in Chapter 3.

Chapter 10 summarizes major conclusions and proposes a number of possible directions for future work.

Chapter 2

Background and Related Work

If it rained knowledge I'd hold out my hand; but I would not give myself the trouble to go in quest of it.

- Samuel Johnson, as quoted in Boswell's Life of Johnson

2.1 Philosophical Background

In this work, the assumption that explanation and localization are rooted in the idea of causality is taken as a given. The problem of causality is one of the oldest issues in philosophy [Sosa and Tooley, 1993]. A survey of the philosophical treatments, dating at least to Aristotle (and arguably to the pre-Socratics) is beyond the scope of this work. The modern development of the philosophy of causality can be considered to begin with Hume [Hume, 1739, 1748]. Stalnaker [Stalnaker, 1968] and Lewis [Lewis, 1973a] propose counterfactual [Lewis, 1973b] theories of causality, but a number of competing notions (e. g. that of Mackie [Mackie, 1965]) are often defended in the philosophical community. Even without considering probabilistic causality [Salmon, 1980] and its related problems, it is safe to say that there is no philosophical consensus on what it means for an event c to *cause* an event e.

Nonetheless, this work builds on the counterfactual framework for causality proposed by David Lewis [Lewis, 1973a]. While Lewis' theory is informative, it is *not* essential that it be accepted to justify the work presented here. The empirical results (Chapters 5 and 9) demonstrate the effectiveness for localization of the distance metric-based methods, and the works of Renieris and Reiss, Zeller, and others rely on the basic assumption that similar executions provide information about causality without making reference to Lewis' work.

Objections to Lewis' theory are often based on the arbitrary nature of distance metrics on possible worlds or the problematic nature of events in his theory [Bennett, 1987]. Both objections are substantially weakened with respect to program executions: in particular, the concept of events need not be considered at all, only observable predicates on the executions. Similarly, issues concerning the temporal direction of causality [Bennett, 1984] are less problematic in a practically oriented task such as automated debugging: whether it makes logical sense to ascribe a predicate that holds after an error occurs as a cause of that error, it makes little sense for purposes of assisting in debugging: the error cannot be "prevented" by code executed after it occurs. Model checkers often consider an execution to terminate once an error state is reached, which further reduces the importance of this consideration. As noted in the introduction, the issues raised by the essential non-observability of counterfactuals in the real world are irrelevant when considering program executions in a model checker, as no particular execution or counterexample is privileged by actuality.

2.2 Related Work

2.2.1 Error Explanation and Fault Localization in Model Checking

The most basic level of explanation in model checking is the production of a counterexample [Clarke et al., 1995; Clarke and Veith, 2003] demonstrating that a program does not satisfy a specification. A counterexample serves as a rudimentary error explanation in that it presents enough detail, in theory, to reconstruct the causality of a failure; additionally, it serves as localization: the portion of the system exercised in the counterexample provides a region in which to search for a fault. Unfortunately, counterexamples typically present too much information: most of the detail provided in any given counterexample is likely to be *irrelevant* to the error. Even finding the shortest counterexample does not remove this problem: for many programs, even the shortest path to a failure will contain much more non-faulty code than faulty code. In cases where the faulty code induces an immediate failure, reading backwards from the end of a counterexample may be sufficient, but faulty code may only manifest as a failure after a large amount of additional code has been executed. In the kinds of subtle hard-to-test-for errors that model checking is particularly suited for detecting, this may be especially likely to happen.

A quite considerable body of work has described proof-like and evidence-based counterexamples [Chechik and Gurfinkel, 2003; Namjoshi, 2001; Peled et al., 2001; Stevens and Stirling, 1998; Tan and Cleaveland, 2002]. Automatically generating assumptions for verification [Cobleigh et al., 2003] can also be seen as a kind of error explanation: an assumption describes the conditions under which a system avoids error. However, all of these approaches appear to be unlikely to result in *succinct* explanations for errors, as they may encode the full complexity of the transition system; one measure of a useful explanation lies in how much it *reduces* the information the user must consider (this is the underlying rationale behind the evaluation technique used in this thesis and in other studies of fault localization).

Error explanation facilities are now featured in Microsoft's SLAM [Ball and Rajamani, 2001] model checker [Ball et al., 2003] and NASA's Java PathFinder 2 (JPF) [Visser et al., 2003] model checker [Groce and Visser, 2003]. Jin, Ravi, and Somenzi proposed a game-like explanation (directed more at hardware than software systems) in which an adversary tries to force the system into error [Jin et al., 2002]. Of these, only JPF uses a (weak) notion of distance between traces, and it cannot solve for nearest successful executions.

The SLAM implementation of error explanation [Ball et al., 2003] first generates all possible successful paths through the system being model checked. A projection to control locations is then made to determine locations that appear in counterexample paths vs. successful paths. In the event that only data, rather than control

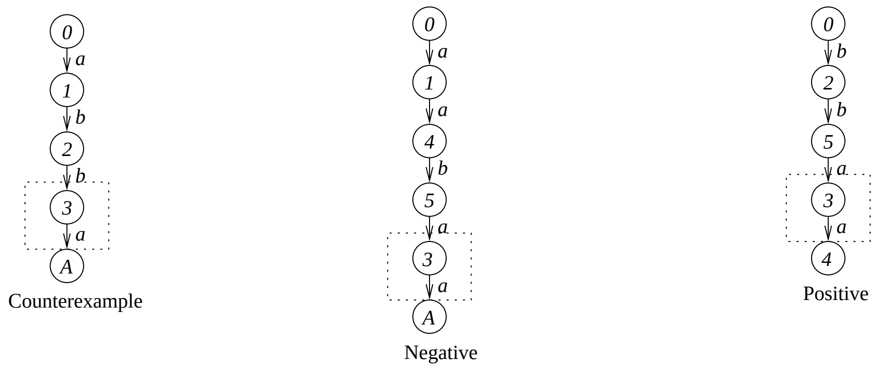


Figure 2.1: A counterexample, a negative, and a positive

flow, distinguishes paths, data values can be taken into account. The published experiments show good results; unfortunately, the current version of SLAM does not support these features, so obtaining up-to-date experimental results or comparisons across examples with this method has been difficult.

The JPF approach [Groce and Visser, 2003] is similar in spirit, but only collects a subset of the successful and unsuccessful paths. The model checker explores the "neighborhood" of a counterexample (which can be defined in part by the search strategy used, including heuristic searches [Groce and Visser, 2002, 2004]) and divides paths into positives and negatives, based on their relationship to the original counterexample (Figure 2.1). In particular, positives and negatives are executions which reach the same control location and take the same action (the JPF explanation method considers Labeled Transition Systems) as the counterexample: a positive proceeds to a non-error state, and a negative proceeds to an error state.

JPF provides a number of analyses over these paths (which can be found in a way that approximates a weak distance metric, so that good and bad paths "closer" to a given counterexample are discovered first), including control location and nondeterministic choice analyses and examination of thread interleavings. Section 5.2 presents results from applying the JPF facilities to a case study, and comparing to the techniques reported here. The JPF implementation has the important feature of providing explanations for concurrent software, demonstrating that thread interleaving changes often lead to a quick understanding of how to fix an error. This "transform analysis" (which applies to other nondeterministic behavior, but is most powerful, perhaps, in the case of thread interleavings) is a direct inspiration for the distance metric approach presented in this thesis.

The most important differences between the SLAM and JPF approaches can be summarized by noting that SLAM achieves completeness in exploration, which can be both beneficial and harmful. JPF's use of a limited notion of distance metrics can produce explanations in cases where complete exploration makes this difficult, but the ad hoc nature of the metrics and exploration reduces the effectiveness and efficiency, as compared to the techniques presented in this thesis (see Chapter 5 for details).

The method of Jin, Ravi, and Somenzi (the "Fate and Free Will" approach) divides the variable space of a system into values controlled by a user and values controlled by an adversary. Using the onion-ring expansion of a symbolic model checker, this approach computes a strategy for forcing the system into error or avoiding error. The explanation is in terms of which variables are crucial for driving the system toward or away from the error states. It is difficult to compare results from this method to the focus of the work in this thesis, as the aim is to explain hardware errors, and the explanations are of a different structure.

Shen et al. [Shen et al., 2004c] propose an algorithm quite similar in spirit to the basic approach taken in our work: finding a closest execution by a distance metrics. They address the problem of multiple nearest witnesses (MNW) by use of an iteration over control flow predicates (which might be thought of as a kind of secondary distance metric that refines the original metric). In all instances other than the "toy" example used to show the virtues of abstract explanation, we have not observed MNW to be a problem — picking an arbitrary successful run appears to work quite well. It may be that the distance metric used by Shen et al. [Shen et al., 2004a,b] makes this more of an issue, as it appears to be more coarse and purely dependent on control-flow than the one presented in Chapter 3.

Sharygina and Peled [Sharygina and Peled, 2001] propose the notion of the *neighborhood* of a counterexample and suggest that an exploration of this region may be useful in understanding an error. However, the exploration, while aided by a testing tool, is essentially manual and offers no automatic analysis.

Temporal queries [Chan, 2000] use a model checker to fill in a hole in a temporal logic formula with the strongest formula that holds for a model. Chan and others [Chan, 2000; Gurfinkel et al., 2002] have proposed using these queries to provide feedback in the event that a property does not hold on a model.

Hopper, Seshia, and Wing combine security protocol model checking [Clarke et al., 2000c] with Kindred's theory-generation [Kindred and Wing, 1996] to improve analysis of attacks on security protocols [Hopper et al., 2000]. In particular, they (1) use theory-generation to produce dubious assumptions and search for examples of violations of these assumptions using the model checker and (2) use theory-generation to determine which assumptions have failed in a counterexample generated by the model checker.

Simmons and Pecheur noted in 2000 that explanation of counterexamples was important for incorporating formal verification into the design cycle for autonomous systems, and suggested the use of truth maintenance systems (TMS) [Nayak and Williams, 1997] for explanation [Simmons and Pecheur, 2000].

Leino et al. have proposed a method for generating counterexamples from refutation-based theorem prover results, and have suggested that error explanation and fault localization techniques similar to those for model checking could be used in this framework [Leino et al., 2004].

Jobstmann, Griesmayer, and Bloem have investigated program repair in a game theoretic formulation in which they concentrate on *memoryless* strategies in order to avoid adding new program state [Jobstmann et al., 2005]. While no evidence of successful automatic software correction for realistic programs is shown, the algorithm has complexity comparable to that of model checking and the authors demonstrate that given a localization, it may be practical to compute actual repairs for assignments in some faulty programs. Staber, Jobstmann, and Bloem propose that diagnosing a fault in software coincides with the problem of finding a fix, and apply their technique to the minmax example from our TACAS 2004 paper [Groce, 2004], a locking example, and a sequential multiplier [Staber et al., 2005].

Finally, Kumar, Kumar, and Viswanathan have investigated the fundamental complexity of the error explanation problem [Kumar et al., 2005]. The first class of explanation they consider is a determination of the smallest number of changes to a system that will ensure that a given counterexample is no longer exhibited. For three models (Mealy machines, extended finite state machines, and pushdown automata) the authors show that this problem is NP-complete, and that no polynomial approximation algorithm can exist unless P = NP. The second explanation class considered is that featured in this thesis work: finding a non-counterexample execution of a system that most closely resembles a given counterexample. For this class, the authors present a polynomial time dynamic programming algorithm for Mealy machine and pushdown automata representations. For extended finite state machines, this problem is as difficult as finding an edit-distance to correct the system. While the theoretical complexity of the dynamic programming algorithm improves on the NP-complete (because SAT-based) techniques presented in subsequent chapters, the time complexity was not generally an issue in our experimental results (Chapters 5 and 9), given that model checking must be performed first.

2.2.2 Error Explanation and Fault Localization in Testing

Fault localization and visualization techniques based on testing, rather than verification, differ from the verification or model-based approaches in that they rely on (and exploit) the availability of a good test suite. When an error discovered by a model checker is not covered by a test suite, these techniques may be of little use.

Dodoo, Donovan, Lin and Ernst [Dodoo et al., 2000] use the Daikon invariant detector [Ernst et al., 1999] to discover differences in invariants between passing and failing test cases, but propose no means to restrict the cases to similar executions relevant for analysis or to generate them from a counterexample. Pytlik et al. report on a (largely unsuccessful) attempt to use potential invariants discovered by Daikon to localize faults [Pytlik et al., 2003]. Hangal and Lam's DIDUCE tool also makes use of invariants (and violations of hypothesized invariants) to isolate errors in Java programs [Hangal and Lam, 2002]. The JPF implementation of error explanation also computes differences in invariants between sets of successful executions and counterexamples using Daikon [Groce and Visser, 2003]. Program spectra [Harrold et al., 2000; Reps et al., 1997] and profiles provide the basis for a number of testing based approaches, which rely on the presence of anomalies in summaries of test executions. The Tarantula tool [Jones et al., 2002] uses a visualization technique to illuminate (likely) faulty statements in programs, as does χSlice [Agrawal et al., 1995].

This work was partly inspired by the success of Andreas Zeller's delta debugging technique [Zeller and Hildebrandt, 2002], which extrapolates between failing and successful test cases to find similar executions. The original delta-debugging work applied to test inputs only, but was later extended to minimize differences in thread interleavings [Choi and Zeller, 2002]. Delta-debugging for deriving cause-effect chains [Zeller, 2002] takes state variables into account, but requires user choice of instrumentation points and does not provide true minimality or always preserve validity of execution traces. Cleve and Zeller extended the cause-effect chain approach to consider relations in *time* as well as *space*, by discovering the point where *cause transitions* occur, and new variables become failure causes [Cleve and Zeller, 2005]. The AskIgor project [AskIgor Website] makes cause-effect chain debugging available via the web.

Renieris and Reiss [Renieris and Reiss, 2003] describe an approach that is quite similar in spirit to the one described here, with the advantages and limitations of a testing rather than model checking basis. They use a distance metric to *select* a successful test run from among a given set rather than, as in this paper, to automatically *generate* a successful run that resembles a given failing run as much as is possible. Experimental results show that this makes their fault localization highly dependent on test case quality. Section 5.2 makes use of a quantitative method for evaluating fault localization approaches proposed by Renieris and Reiss.

2.2.3 Slicing and Counterexample Minimization

The "slicing" technique presented in Section 4 should be understood in the context of both work on program slicing [Agrawal et al., 1995; Tip, 1995; Weiser, 1979; Zhang et al., 2003] and some work on counterexample minimization [Groce and Kroening, 2004; Ravi and Somenzi, 2004; Shen et al., 2005]. The technique presented here can be distinguished from these approaches in that it is not a "true" slice, but the result of a causal analysis that can only be performed between two executions which differ on a predicate (in this application, the presence of an error).

In general, program slicing is another approach to the problem of determining which portions of a program might contain an error: a static slice backwards from the point at which an error is detectable should contain the faulty code. Dynamic slicing provides the same localization property, for a specific execution of a program. Agrawal [Agrawal et al., 1995] treats slicing very much as a debugging technique, and makes use of intersections and other slice operations for debugging purposes.

Counterexample minimization also attempts to reduce the amount of irrelevant information contained in a counterexample. Some aspects of this work resemble slicing approaches, in that they make use of the irrelevance of certain variables to the SAT results for a bounded model checking problem [Ravi and Somenzi, 2004]. However, attempts to minimize the length of an error trace or to "semantically" reduce the program variable values in a counterexample [Groce and Kroening, 2004] are essentially unrelated to slicing.

Both slicing and counterexample minimization are, in some sense, orthogonal to fault localization. In particular, fault localization and our style of explanation are not suited to the task of grasping all "important" behavior in a trace. When using model checking to produce "counterexamples" that are best seen as solutions to a problem (i.e., model checking for planning), minimization is likely to be more useful than explanation.

2.2.4 Error Explanation and Fault Localization in Artificial Intelligence (Model-Based Diagnosis)

Analyses of causality from the field of artificial intelligence usually rely on causal theories or more precise logical models of relationships between components than are available in model checking of software systems [Galles and Pearl, 1997; Lucas, 1998; Reiter, 1987], but is applicable in some cases to software systems with specifications on the level considered in in this work. The JADE system for diagnosing errors in Java programs makes use of model-based techniques [Mateis et al., 2000]. The program model is extracted automatically, but requires a programmer to answer queries to manually identify whether variables have correct values at points that are candidates for diagnosis. Mayer and Stumptner present a more automated system based on multiple abstract models and a conflict detection mechanism [Mayer and Stumptner, 2003]. Wotawa has discussed the relationship between model-based debugging and program slicing [Wotawa, 2002] and program mutation [Wotawa, 2001].

Shapiro [Shapiro, 1983] introduced a technique for debugging logic programs that also relies on interaction with a user as an oracle. Further developments based on this technique have reduced the (potentially very large) number of user queries (in part by use of slicing) [Kókai et al., 1997]. Related techniques for debugging of programs in functional languages, such as Haskell, rely on similar models or queries and a semantics of the consequences of computations [Alpuente et al., 2002].

We have worked with Mota, Oliviera, et al. on preliminary efforts to integrate our error explanation techniques into an agent-based approach to software development, and explain errors in UML models [Mota et al., 2003].

The idea of automatically correcting programs suggested in Section 10.2.8 bears more obvious connections to diagnosis work than the techniques presented in the bulk of this thesis. Given the state of the art in model-based diagnosis, it appears that successful program correction would probably require either a user to answer queries as an oracle or a higher degree of specification than is generally provided in software model checking. Pursuit of automatic correction would also introduce a connection to literature on program mutation and mutation testing [Budd, 1980].

2.2.5 String and Sequence Comparison

The distance metrics used for concrete explanation (Chapter 3) are based on the static single assignment (SSA) form [Alpern et al., 1988] and loop unrolling techniques often used in static analysis and compiler optimization. The metrics for abstract explanation presented in Chapter 7 are more similar to those used in traditional string or biological sequence analysis [Durbin et al., 1998; Gusfield, 1997; Sankoff and Kruskal, 1983]. In either case, the metric is a Levenshtein distance [Sankoff and Kruskal, 1983], a count of atomic operations needed to transform one string (or execution) into another (similar to Zeller's Δs [Zeller and Hildebrandt, 2002]). For solving the distance metric constraints produced in either case, we rely on an encoding as a pseudo-Boolean problem [Aloul et al., 2002].

2.3 Original Contributions

This thesis presents a new distance metric for program executions, and uses this metric to provide error explanations based on David Lewis' counterfactual analysis of causality. While previous approaches have taken into account the similarity of executions, our approach is the first to automatically *generate* a successful execution that is *maximally* similar to a counterexample. Solving this optimization problem produces a set of differences that is as succinct as possible. Our novel slicing algorithm then makes use of the program semantics and the fact that we are interested only in causal differences to further reduce the amount of information that must be understood by a user. By extending the original idea to distance metrics and explanations over *abstract executions* of a program, we provide for automatic generalization to the *logical* causes of an error. The idea of abstract error explanation also introduces the notion that automatically generated program abstractions can be used for program understanding as well as verification.

Chapter 3

Error Explanation with Distance Metrics

". . . I'm glad I was able to give a scientific explanation to it, or it would have worried me."

- R. A. Lafferty, "Narrow Valley"

3.1 Distance Metrics for Program Executions

A distance metric [Sankoff and Kruskal, 1983] for program executions is a function $d(a, b)$ (where a and b are executions of the same program) that satisfies the following properties:

1. *Nonnegative property:* $\forall a \ . \ \forall b \ . \ d(a, b) \geq 0$

2. *Zero property:* $\forall a \ . \ \forall b \ . \ d(a, b) = 0 \Leftrightarrow a = b$

3. *Symmetry:* $\forall a \,.\, \forall b \,.\, d(a,b) = d(b,a)$

4. *Triangle inequality:* $\forall a \,.\, \forall b \,.\, \forall c \,.\, d(a,b) + d(b,c) \geq d(a,c)$

In order to compute distances between program executions, we need a single, well-defined representation for those executions.

3.1.1 Representing Program Executions

Bounded model checking (BMC) [Biere et al., 1999] also relies on a representation for executions: in BMC, the model checking problem is translated into a SAT formula whose satisfying assignments represent counterexamples of a certain length.

CBMC [Kroening et al., 2004] is a BMC tool for ANSI C programs. Given an ANSI C program and a set of *unwinding depths* U (the maximum number of times each loop may be executed), CBMC produces a set of constraints that encode all executions of the program in which loops have finite unwindings. CBMC uses unwinding assertions to notify the user if counterexamples with more loop executions are possible. The representation used is based on loop unrolling and a transformation very much like static single assignment (SSA) form [Alpern et al., 1988]. CBMC first unrolls all loops (and recursion) in a program to some bounded depth. The SSA-like transformation then produces a new program in which each variable is assigned exactly once and control flow is represented by conditional expressions (like the ϕ functions used in traditional SSA). This differs from more traditional SSA in that CBMC carries through the use of ϕ functions to produce a purely equational form in which control flow has been completely removed from the program. The \guard functions used below are therefore not quite the same as the traditional "magic" n-ary ϕ functions found in SSA, but are a true replacement for the original program's control flow. This transformation allows execution of the program (to a bounded depth) to be expressed completely by a series of equations. CBMC maintains a mapping from SSA(-like) form[1] variables to the pre-transformation source code, ensuring that counterexamples and explanation results can be given in terms of the original program.

CBMC and `explain` handle the full set of ANSI C types, structures, and pointer operations including pointer arithmetic. CBMC checks only safety properties, although in principle BMC (and the `explain` approach) can handle full LTL [Biere et al., 2002][2].

Given the example program minmax.c (Figure 3.1), which contains an intentionally introduced fault, CBMC produces the constraints shown in Figure 3.2 (U is not needed, as minmax.c is loop-free)[3]. The renamed variables describe unique assignment points: `most#1` denotes the second possible assignment to `most`, `least#2` denotes the third possible assignment to `least`, and so forth. CBMC assigns uninitialized (`#0`) values nondeterministically — thus `input1, input2,` and `input3` will be unconstrained 32 bit integer values. The \\`guard` variables encode the control flow

[1]In subsequent discussion, we will refer to this SSA-like form as "SSA form" for the sake of convenience.

[2]Explanation for LTL properties has been implemented for error explanation in MAGIC [Chaki et al., 2004c], as described in Chapter 8.

[3]Output is slightly simplified for readability.

```
int main () {
  int input1, input2, input3;       //input values
  int least = input1;               //least#0
  int most = input1;                //most#0
  if (most < input2)                //guard#1
    most = input2;                  //most#1,2
  if (most < input3)                //guard#2
    most = input3;                  //most#3,4
  if (least > input2)               //guard#3
    most = input2;                  //most#5,6 (ERROR!)
  if (least > input3)               //guard#4
    least = input3;                 //least#1,2
  assert (least <= most);           //specification
}
```

Figure 3.1: minmax.c

of the program (\`guard#1` is the value of the conditional on line 5, etc.), and are used when presenting the counterexample to the user (and in the distance metric). Control flow is handled by conditional choice functions, as usual in SSA form: the constraint {-10}, for instance, assigns `most#2` to either `most#1` or `most#0`. The value assigned depends on the value of the conditional (\`guard#1`, from source line 5) for the assignment to `most#1`. The syntax is that of the C conditional expression: if \`guard#1` is true (i.e., `most#0 < input2#0`), `most#2` is assigned the value of `most#1`, otherwise it gets the value for `most#0`. Thus `most#2` is the value assigned to `most` at the point before the execution of line 7 of minmax.c. The property/specification is represented by the *claim*, {1}, which appears below the line, indicating that the conjunction of these constraints should imply the truth of the claim(s). A solution to the set of constraints {-1}-{-14} is an execution of minmax.c. If the solution satisfies the claim, {1} (`least#2 <= most#6`), it is a successful execution of minmax.c; if it satisfies the negation of the claim, ¬{1} (`least#2 > most#6`), it is a *c*ounterexample.

CBMC generates CNF clauses representing the conjunction of ({-1}∧{-2}∧ . . . {-14}) with the negation of the claim (¬{1}). CBMC calls zChaff [Moskewicz et al., 2001], which produces a satisfying assignment in less than a second. The satisfying assignment encodes an execution of minmax.c in which the assertion is violated (Figure 3.3).

Figure 3.4 shows the counterexample from Figure 3.3 in terms of the SSA form assignments (the internal representation used by CBMC for an execution).

```
{-14} least#0 == input1#0
{-13} most#0 == input1#0
{-12} \guard#1 == (most#0 < input2#0)
{-11} most#1 == input2#0
{-10} most#2 == (\guard#1 ? most#1 : most#0)
{-9}  \guard#2 == (most#2 < input3#0)
{-8}  most#3 == input3#0
{-7}  most#4 == (\guard#2 ? most#4 : most#3)
{-6}  \guard#3 == (least#0 > input2#0)
{-5}  most#5 == input2#0
{-4}  most#6 == (\guard#3 ? most#5 : most#4)
{-3}  \guard#4 == (least#0 > input3#0)
{-2}  least#1 == input3#0
{-1}  least#2 == (\guard#4 ? least#1 : least#0)
|--------------------------
{1}   least#2 <= most#6
```

Figure 3.2: Constraints generated for minmax.c

```
Initial State
----------------------------------------------------

State 1 line 2 function c::main
------------------------------------------(input1#0)
  input1 = 1

State 2 line 2 function c::main
------------------------------------------(input2#0)
  input2 = 0

State 3 line 2 function c::main
------------------------------------------(input3#0)
  input3 = 1

State 4 line 3 function c::main
-------------------------------------------(least#0)
  least = 1
```

Figure 3.3: Counterexample for minmax.c

```
State 5 line 4 function c::main
---------------------------------------------(most#0)
  most = 1

State 12 line 10 function c::main
---------------------------------------------(most#6)
  most = 0

Failed assertion: assertion line 13 function c::main
```

Figure 3.3 (continued)

input1#0 = 1	most#3 = 1
input2#0 = 0	most#4 = 1
input3#0 = 1	\guard#3 = TRUE
least#0 = 1	most#5 = 0
most#0 = 0	most#6 = 0
\guard#1 = FALSE	\guard#4 = FALSE
most#1 = 0	least#1 = 1
most#2 = 1	least#2 = 1
\guard#2 = FALSE	

Figure 3.4: Counterexample values for minmax.ce

In the counterexample, the three inputs have values of `1`, `0`, and `1`, respectively. The initial values of `least` and `most` (`least#0 and most#0`) are both `1`, as a result of the assignments at lines 3 and 4. Execution then proceeds through the various comparisons: at line 5, `most#0` is compared to `input2#0` (this is `\guard#1`). The guard is not satisfied, and so line 6 is not executed. Lines 8 and 12 are also not executed because the conditions of the `if` statements (`\guard#2` and `\guard#4` respectively) are not satisfied. The only conditional that is satisfied is at line 9, where `least#0 > input2#0`. Line 10 is executed, assigning `input2` to `most` rather than `least`.

In this simple case, understanding the error in the code is not difficult (especially as the comments to the code indicate the location of the error). Line 10 should be an assignment to `least` rather than to `most`. A good explanation for this faulty program should isolate the error to line 10. We follow Renieris and Reiss in considering the fault to be the program point at which a change should be made to correct the error. Of course, there may be many ways to fix an error. The preference of one way to satisfy a specification rather than another is to some extent subjective. In this case, however, other possible candidates for the "fault" do seem less natural: e.g., changing the guard on line 9 alone cannot result in a guarantee of satisfaction for the assertion. Given that `most` can end up holding the value of any of the inputs (as a result of the comparisons on lines 5-8), including `input2`, a small modification to the program ensuring satisfaction of the assertion on line 14 probably requires comparing least and `input2` — in order to avoid the case where `least` is greater than `most` because `most` is `input2`. For the larger programs examined in Chapter 5, the "best fixes" and thus fault locations are both difficult to reasonably dispute and established beforehand by a third party.

For given loop bounds (irrelevant in this case), all executions of a program can be represented as sets of assignments to the variables appearing in the constraints. Moreover, all executions (for fixed U) are represented as assignments to the same variables. Different flow of control will simply result in differing `\guard` values (taking the place of the traditional ϕ functions) assignments.

3.1.2 The Distance Metric d

The distance metric d will be defined only between two executions of the same program with the same maximum bound on loop unwindings[4]. This guarantees that any two executions will be represented by constraints on the same variables. The distance, $d(a, b)$, is equal to the number of variables to which a and b assign different values. Formally:

Definition 4 (distance, $d(a, b)$) *Let a and b be executions of a program P, represented as sets of assignments, $a = \{v_0 = val_0^a, v_1 = val_1^a, \ldots, v_n = val_n^a\}$ and $b = \{v_0 = val_0^b, v_1 = val_1^b, \ldots, v_n = val_n^b\}$.*

$$d(a, b) = \sum_{i=0}^{n} \Delta(i)$$

[4]Counterexamples can be extended to allow for more unwindings in the explanation.

where

$$\Delta(i) = \begin{cases} 0 \texttt{ if } & val_i^a = val_i^b \\ 1 \texttt{ if } & val_i^a \neq val_i^b \end{cases}$$

Here v_0, v_1, etc. do not indicate the first, second, third, and so forth assignments in a considered as an execution trace, but uniquely named SSA form assignments. The pairing indicates that the value for each assignment in execution a is compared to the assignment with the same unique name in execution b. SSA form guarantees that for the same loop unwindings, there will be a matching assignment in b for each assignment in a. In the running example $\{v_0, v_1, v_2, v_3, v_4 \ldots\}$ are {`input1#0`, `input2#0`, `input3#0`, `least#0`, `most#0`, ...}, execution a could be taken to be the counterexample (Figures 3.3 and 3.4), and execution b might be the most similar successful execution (see Figures 3.6 and 3.7).

This definition is equivalent to the Levenshtein distance [Sankoff and Kruskal, 1983] if we consider executions as strings where the alphabet elements are assignments and substitution is the only allowed operation[5]. The properties of inequality guarantee that d satisfies the four metric properties.

The metric d is measuring *all* variable value and control flow changes needed to "transform" execution a into execution b or vice versa. This includes changes in input variables *a*nd "internal" variables that are fully determined by the inputs. Of course, it would be possible to only measure changes over input variables (as these are the only variables a user has full control over), or only over input variables

[5]A Levenshtein distance is one based on a composition of atomic operations by which one sequence or string may be transformed into another.

and control flow changes, and so forth. The purpose of the metric, however, is not to find a sequence of inputs that is "close" to the original set — it is to find an entire *execution* that is similar. Changing inputs is not a very useful "solution" to an error, it is simply a means to the end of fault localization and error explanation. A small change in input might result in a drastic change in behavior. Consider the case of a reactive system accepting commands from a user that allows for an "abort" sequence, after which all other inputs are ignored and no the system takes no actions. Changing the first command sequence sent to such a system to "abort" will result in an execution with very similar inputs to a failure. The resulting behavior is also (presumably) error free. However, it is not very useful for purpose of localizing a fault induced by improper response to an input later in the command sequence. Even if the distance is measured over inputs, changes over intermediate variables need to be tracked in order to compute a localization (unless the localization is restricted to input points, which greatly reduces the chances of pinpointing a fault). Experimental results (see Section 5.3.1) confirm the hypothesis that measuring changes over inputs results in less effective fault localization than measuring distance over all aspects of executions.

Another notion of "closest execution" would allow changes to values at any point in program execution, even if the change (called an *intervention*) results in a violation of the program semantics/transition relation. It should be clear that this is problematic for fault localization: typically a "symptomatic" value close to an assertion will be arbitrarily changed to produce "correct" (but impossible) program behavior, giving no information about the real location of the fault. In the event that such a change *is* coincident with a fault, it is likely to be close enough to the detected error that a simple reverse-reading of the counterexample trace would also quickly discover the error. As expected, experimental results (Section 5.3.4) show that this kind of metric[6] unlikely to result in a fruitful approach to explanation.

The metric d differs from the metrics often used in sequence comparison in that it does not need to make use of a notion of *alignment*. In the distance metrics traditionally used to compare strings sequences [Sankoff and Kruskal, 1983], an *alignment* determines which alphabet symbols or sequence elements should be compared in computing a distance. The need for alignments arises from the possibility that strings being compared may have different numbers of characters, for example. If program executions are represented by sequences of states and actions, the same issue of alignment naturally arises: should state #2 of execution a be compared to state #2 of execution b, or to some other state? If control flow is such that the respective second states of the executions are in different control locations, this may not be the best possible choice. Consider the case in which execution a of some program P takes a branch at line 1 and thus has control flow passing through lines 1, 2, 3, 4, and 5 of the program, while execution b does not take the branch, and so passes through lines 1, 4, and 5. If a and b are represented not in SSA form, but as sequences of states (5 states in the case of a, and 3 in the case of b), comparing the executions without alignment (or, rather, with a naïve fixed alignment) will "pair up" the control locations for states as follows: $\{(1,1),(2,4),(3,5)\}$, with the states at locations 4 and 5 of a not matched with any states from b. However, the same variables may not even

[6]Strictly speaking, this is a change in the set of "executions" allowed, not a change in the metric.

be in scope at different control locations: it seems more reasonable to compute the distance by comparing the states at the same control locations and leave the states at locations 2 and 3 unmatched. In the presence of loops, even the principle of comparing states with matching control locations does not establish a unique matching, and so an alignment must be chosen in order to determine the distance between two sequences: the true distance is determined by choosing an alignment that produces a minimal distance, where distance is a function not only of the sequences but of alignment.

The SSA form based representation encodes executions as assignments to variables, not as state/action sequences: while control flow can be extracted from this representation, it is not necessary to take any measures to handle cases in which two executions have different control flow. In contrast, the MAGIC [Chaki et al., 2003a, 2004a] implementation of error explanation [Chaki et al., 2004c] (see Chapter 7) does represent an execution as a series of states and actions, including a program counter to represent control flow. Although viewing executions as sequences of states is a natural result of the usual Kripke structure approach to verification, the need to compute an alignment and compare *all* data elements when two states are aligned can impose a serious overhead on explanation [Chaki et al., 2004c] (this issue is revisited in Chapter 9).

In the CBMC/`explain` representation, however, the issue of alignments does not arise. Executions a and b will both be represented as assignments to `input1`, `input2`, `input3`, `\guard#0`-`\guard#4`, `least#0`-`least#2`, and `most#0`-`most#6`. The distance between the executions, again, is simply a *count* of the assignments for which they do not agree. This does result in certain counter-intuitive behavior: for instance, although neither execution a nor execution b executes the code on line 12 (`\guard#4` is FALSE in both cases), the values of `least#1` will be compared. Therefore, if the values for `input3` differ, this will be counted twice: once as a difference in `input3`, and once as a difference in `least#1`, despite the second value not being used in either execution. In general, a metric based on SSA form unwindings may be heavily influenced by results from code that *is not executed*, in one or both of the executions being compared. Any differences in such code can eventually be traced to differences in input values, but the *weighting* of differences may not match user intuitions. It is not that information is *lost* in the SSA form encoding: it is, as shown in the counterexamples, possible to determine the control flow of an execution from the `\guard` (or ϕ function) values; however, to take this into account complicates the metric definition and introduces a potentially expensive degree of complexity into the optimization problem of finding a maximally similar execution[7].

A state and alignment based metric avoids this peculiarity, at a potentially high computational cost. Experimental results [Chaki et al., 2004c] show that in some cases the "counterintuitive" SSA form based metric may produce better explanations — perhaps because it takes all potential paths into account. In all cases, we are comparing two executions, on of which contains a fault. This means that we cannot be certain that code that is not executed is in fact irrelevant to the fundamental problem: one possible error is that a condition that should have been satisfied in the

[7]Each Δ, as shown below, would potentially introduce a case split based on whether the code was executed in one, both, or neither of the executions being compared.

counterexample a was not satisfied. If this condition *is* satisfied in the successful execution b it obviously might be beneficial to take into account that a's execution over the incorrectly omitted control flow would have been very similar to b's execution of the same code. The same reasoning applies, in a less compelling manner, to the case in which b also fails to execute the omitted code, but fails to execute it because a change elsewhere means the omission is correct.

In summary, the representation for executions presented here has the advantage of combining precision and relative simplicity, and results in a very clean (and easy to compute) distance metric. The pitfalls involved in trying to align executions with different control flow for purposes of comparison are completely avoided by the use of SSA form. Obviously, the details of the SSA form encoding may need to be hidden from non-expert users (the CBMC GUI provides this service) — a good *presentation* of a trace may hide information that is useful at the level of *representation*. Any gains in the direct presentability of the representation itself (such as removing values for code that is not executed) are likely to be purchased with a loss of simplicity in the distance metric d, as seen in the metric used by MAGIC.

3.1.3 Choosing an Unwinding Depth

The definition of d presented above applies to executions with the same unwinding depth and therefore (due to SSA form) the same variable assignment. However, it is possible to extend the metric to any unwinding depth by simply considering there to be a difference for each variable present in the successful execution but not in the counterexample. Using this extension of d, a search for a successful execution can be carried out for any unwinding depth. It is, of course, impossible to bound in general the length of the closest successful execution. In fact, no successful execution of a particular program may exist. However, given a closest successful execution within some unwinding bounds, it is possible to determine a maximum possible bound within which a closer execution may be found. For a program P, each unwinding depth determines the number of variables in the SSA form unwinding of the program. If the counterexample is represented by i variables, and the successful execution's unwinding requires $j > i$ variables, then the minimum possible distance between the counterexample and any successful execution at that unwinding depth is $j - i$. Given a successful execution with distance d from a counterexample, it is impossible for a successful execution with unwinding depth such that $j - i \geq d$ to be closer to the counterexample.

3.2 Producing an Explanation

Generating an explanation for an error requires two phases:

- First, `explain` produces a successful execution that is as similar as possible to the counterexample. Section 3.2.1 describes how to set up and solve this optimization problem.
- The second phase produces a subset of the changes between this execution and the counterexample which are *causally necessary* in order to avoid the error.

The subset is determined by means of the Δ-slicing algorithm described in Chapter 4.

3.2.1 Finding the Closest Successful Execution

The next step is to consider the optimization problem of finding an execution that satisfies a constraint and is as close as possible to a given execution. The constraint is that the execution *not* be a counterexample. The original BMC problem is formed by negating the verification claim V, where V is the conjunction of all assertions, bounds checks, overflow checks, unwinding assertions, and other conditions for correctness, conditioned by any assumptions. For minmax.c, V is:

```
{1}:   least#2 <= most#6
```

and the SAT instance S to find a counterexample is formed by negating V:

```
¬{1}:  least#2 > most#6.
```

In order to find a successful execution it is sufficient to use the original, unnegated, claim V.

The changes that, when counted, give the distance to a given fixed execution (e.g., a counterexample) can be easily added to the encoding of the constraints that define the transition relation for a program. The values for the Δ functions necessary to compute the distance are added as new constraints (Figure 3.5) by the `explain` tool. For the SSA form based metric, this (rather simple) set of Boolean variables conditioned on whether the value from the fixed execution has been changed is *fully sufficient to allow computation of the distance to that fixed execution.* These Δ constraints are the same as the $Delta(i)$ from Definition 4. *The distance $d(a, b)$ (where a is the fixed execution) is just the sum of these Δ values, considered as 1 where there is a change and 0 where there is no change.*

These constraints *do not* affect satisfiability; correct values can always be assigned for the Δs. The Δ values are used to encode the optimization problem. For a fixed a, $d(a, b) = n$ can directly be encoded as a constraint by requiring that exactly n of the Δs be set to 1 in the solution. However, it is more efficient (as the structure and optimization aspect of the problem can then be incorporated into the SAT solver's algorithm [Aloul et al., 2002]) to use pseudo-Boolean (0-1) constraints[8] [Barth, 1995] and use the pseudo-Boolean solver PBS [Aloul et al., 2002] in place of zChaff. A pseudo-Boolean formula has the form:

$$(\Sigma_{i=1}^{n} c_i \cdot b_i) \bowtie k$$

where for $1 \leq i \leq n$, each b_i is a Boolean variable, c_i is a rational constant, k is a rational constant, and $\bowtie$ is one of $\{<, \leq, >, \geq, =\}$. For our purposes, each c_i is 1, and each b_i is one of the Δ variables introduced above[9]. PBS accepts a SAT

[8] So called because they use Boolean values to represent not logical constraints alone but summations in the sense of 0-1 ILP.

[9] In practice, several Δ variables (for example, changes in guards) may be equivalent to the same CNF variable, after simplification. In this case, the coefficient on that variable is equal to the number of Δs it represents, but we can treat the $\Delta(i)$s as independent without loss of generality, as the result is the same.

```
input1#0Δ == (input1#0 != 1)

input2#0Δ == (input2#0 != 0)

input3#0Δ == (input3#0 != 1)

least#0Δ == (least#1 != 1)

most#0Δ == (most#1 != 1)

\guard#1Δ == (\guard#1 != FALSE)

most#1Δ == (most#2 != 0)

most#2Δ == (most#3 != 1)

\guard#2Δ == (\guard#2 != FALSE)

most#3Δ == (most#4 != 1)

most#4Δ == (most#5 != 1)

\guard#3Δ == (\guard#3 != TRUE)

most#5Δ == (most#6 != 0)

most#6Δ == (most#7 != 0)

\guard#4Δ == (\guard#4 != FALSE)

least#1Δ == (least#2 != 1)

least#2Δ == (least#3 != 1)
```

Figure 3.5: Δs for minmax.c and the counterexample in Figure 3.3

problem expressed as CNF, augmented with a pseudo-Boolean formula. In addition to solving for pseudo-Boolean constraints such as $d(a,b) = k$, $d(a,b) < k$, $d(a,b) \geq k$, PBS uses a binary search to solve pseudo-Boolean optimization problems, minimizing or maximizing $d(a,b)$. For error explanation, the pseudo-Boolean problem is to minimize the distance to the counterexample a.

From the counterexample shown in Figure 3.3, we can generate an execution (1) with minimal distance from the counterexample and (2) in which the assertion on line 13 is not violated. Constraints {-1}-{-14} are conjoined with the Δ constraints (Figure 3.5) and the *unnegated* verification claim {1}. The pseudo-Boolean constraints express an optimization problem of minimizing the sum of the Δs. The solution is an execution (Figure 3.6) in which a change in the value of `input2` results in `least <= most` being `true` at line 13. This solution is not unique. In general, there may be a very large set of executions that have the same distance from a counterexample.

The values of the Δs (Figure 3.8) allow us to examine precisely the points at which the two executions differ. The first change is the different value for `input2`. At least one of the inputs must change in order for the assertion to hold, as the other values are all completely determined by the three inputs. The next change is in the *potential* assignment to `most` at line 6. In other words, a change is reported at line 6 despite the fact that line 6 is not executed in either the counterexample or the successful execution. It is, of course, trivial to hide changes guarded by false conditions from the user; such changes are retained in this presentation in order to make the nature of the distance metric clear. Such assignments are automatically removed by the Δ-slicing technique presented in Chapter 4 (see Figure 4.6). This is an instance of

```
Initial State
----------------------------------------------------

State 1 line 2 function c::main
------------------------------------------(input1#0)
  input1 = 1

State 2 line 2 function c::main
------------------------------------------(input2#0)
  input2 = 1

State 3 line 2 function c::main
------------------------------------------(input3#0)
  input3 = 1

State 4 line 3 function c::main
-------------------------------------------(least#0)
  least = 1

State 5 line 4 function c::main
--------------------------------------------(most#0)
  most = 1
```

Figure 3.6: Closest successful execution for minmax.c

input1#0 = 1	most#3 = 1
input2#0 = 1	most#4 = 1
input3#0 = 1	\guard#3 = FALSE
least#0 = 1	most#5 = 1
most#0 = 1	most#6 = 1
\guard#1 = FALSE	\guard#4 = FALSE
most#1 = 1	least#1 = 1
most#2 = 1	least#2 = 1
\guard#2 = FALSE	

Figure 3.7: Closest successful execution values for minmax.c

```
Value changed:  input2#0 from 0 to 1
Value changed:  most#1 from 0 to 1
                file minmax.c line 6 function c::main
Guard changed:  least#0 > input2#0 (\guard#3) was TRUE
                file minmax.c line 9 function c::main
Value changed:  most#5 from 0 to 1
                file minmax.c line 10 function c::main
Value changed:  most#6 from 0 to 1
```

Figure 3.8: Δ values ($\Delta = 1$) for execution in Figure 3.6

the counter-intuitive nature of the SSA form: because the condition on line 5 is still not satisfied (indeed, none of the guards are satisfied in this successful execution), the value of `most` which reaches line 7 (`most#2`) is not changed. While one of the potential values for `most` at the merge point is altered, the "ϕ function", i.e., the conditional split on the guard for `most#2`, retains its value from the counterexample. The next change occurs at the guard to the erroneous code: `least#0` is no longer less than `input2#0`, and so the assignment to `most` at line 10 is not executed. The potential value that might have been assigned (`most#5`) is also changed, as `input2` has changed its value. Finally, the value of `most` that reaches the assertion, `most#6`, has changed from 0 to 1 (because line 10 has not been executed, although in this case executing line 10 would not change the value of `most`). The explanation shows that not executing the code at line 10, where the fault appears, causes the assertion to succeed. The error has been successfully isolated.

3.3 Closest Successful Execution Δs and Causal Dependence

The intuition that comparison of the counterexample with minimally different successful executions provides information as to the causes of an error can be justified by showing that Δs from a (closest) successful execution are equivalent to a cause c:

Theorem 1 *Let a be the counterexample trace and let b be any closest successful execution to a. Let D be the set of Δs for which the value is not 0 (the values in* *which a and b differ). If δ is a predicate stating that an execution disagrees with b for at least one of these values, and e is the proposition that an error occurs, e is causally dependent on δ in a.*

Proof:

A predicate e is causally dependent on δ in a iff for all of the closest executions for which $\neg\delta$ is true, $\neg e$ is also true. Since $\neg\delta$ only holds for executions which agree with b for all values in D, $\neg\delta(b)$ must hold. Additionally, $\neg e(b)$ must be true, as b is defined as a closest successful *execution to a. Assume that some trace b' exists, such that $\neg\delta(b') \wedge e(b') \wedge d(a, b') \leq d(a, b)$. Now, b' must differ from b in some value (as $e(b') \wedge \neg e(b)$). However, b' cannot differ from b for any value in D, or $\delta(b')$ would be true. Thus, if b' differs from b in a value other than those in D, b' must also differ from a in this value. Therefore, $d(a, b') > d(a, b)$, which contradicts our assumption. Hence, e must be causally dependent on δ in a.*

In the running example minmax.c, δ is the predicate `(input3#0 != 0)` $\vee$ `(most#3 != 0)` $\vee$ `(least#1 != 0)` $\vee$ `(least#2 != 0)`. Finding the closest successful execution also produces a predicate $c(\delta)$ on which the error is causally dependent. Actually, this proof holds for *any* successful execution. Minimizing the distance serves to minimize the number of terms in δ. A δ with minimal terms can be used as a starting point for hypotheses about a more general cause for the error. As Cleve and Zeller note [Cleve and Zeller, 2005], finding *a* cause (in some formal sense) is often neither difficult nor useful (the full program state at the point of failure is "a cause"; the set of inputs is "a cause", etc. — and these may indeed be causes by

Lewis' definition of causal dependence): the challenge is to find a *small* cause with sufficient explanatory power to provide an understanding of a fault.

More generally, this proof should hold for *any* metric that can be formulated in terms of a Levenshtein distance, i.e., any metric based on operations that can be represented by mutually exclusive independent terms — such as the atomic changes to the SSA form representation. Such a formulation should be possible for the non-SSA form metric presented in later chapters for abstract executions; however, the reduction to atomic terms is considerably less natural, and the value of the explanations as conjunctions in terms of predicates on states and predicates once alignment and position are taken into account is dubious. If an SSA-based abstract explanation approach is introduced, the formulation in these terms again becomes natural.

It is important to note that this proof *does not* mean that the explanation is in any sense guaranteed to be useful to a user. There is no reason to expect that always producing a "good" explanation (even in the limited sense of fault localization) is possible, given the partial specifications typically available. The proof guarantees that the explanation reflects a *cause* of error in Lewis' sense, but not that this will be a sense very useful to a programmer hoping to correct the error. Experimental results show that explanation along these lines often succeeds in providing useful feedback, and is typically much better for fault localization than competing techniques. This is an independent result, not a confirmation of any "implications" of the above proof.

Chapter 4

Δ-Slicing

The simplification of anything is always sensational.

- G. K. Chesterton, Varied Types

4.1 Motivation

A successful path with minimal distance to a counterexample may include changes in values that are not actually relevant to the specification. For example, changes in an input value are necessarily reflected in all values dependent on that input.

Consider the program and Δ values in Figures 4.1 and 4.2. The change to `z` is necessary but also irrelevant to the assertion on line 14. In this case, various static or dynamic slicing techniques [Tip, 1995] would suffice to remove the unimportant variable `z`. Generally, however, static slicing is of limited value as there may be some execution path other than the counterexample or successful path in which a variable

```
int main () {
  int input1, input2;
  int x = 1, y = 1, z = 1;
  if (input1 > 0) {
    x += 5;
    y += 6;
    z += 4;
  }
  if (input2 > 0) {
    x += 6;
    y += 5;
    z += 4;
  }
  assert ((x < 10) || (y < 10));
}
```

Figure 4.1: slice.c

```
Value changed:  input2#0 from 1 to 0
Guard changed:  input2#0 > 0 (\guard#2) was TRUE
                line 9 function c::main
Value changed:  x#4 from 12 to 6
                line 10 function c::main
Value changed:  y#4 from 12 to 7
                line 11 function c::main
Value changed:  z#4 from 9 to 5
                line 12 function c::main
```

Figure 4.2: Δ values for slice.c

is relevant. Dynamic slicing, in which a slice is computed based on a given set of program inputs (rather than over all possible program paths), raises the question of whether to consider the input values for the counterexample or for the successful path.

If we assume a failing run with the values 1 and 1 for `input1` and `input2`, a typical dynamic slice on the execution would indicate that lines 2, 3, 4, 5, 6, 9, 10, and 11 are relevant. In this case, however, the explanation technique has already focused our attention on a subset of the failing run: no changes appear other than at lines 9, 10, 11, and 12. If dynamic slicing was applied to these Δ locations rather than the full execution, lines 9, 10, and 11 would be considered relevant, as both `x` and `y` influence the assertion at line 14. Using the differences rather than the full execution goes beyond the reductions provided by a dynamic slice, whether we use the dynamic slice to reduce the size of the *Delta*s or the *Delta*s to filter a dynamic slice of the full execution.

Notice, however, that in order to avoid the error, it is *not* required that both `x` and `y` change values. A change in either `x` or `y` is sufficient. It is true that in the program as written, a change is only observed in `x` when a change is also observed in `y`, but the basic assumption of error explanation is that the program's behavior is incorrect. It might be useful to observe (which dynamic slicing will not) that *within a single execution* two routes to a value change that removes the observed error potentially exist.

This issue of two causal "routes" within an execution is independent of the possibility that there may be more than one successful execution at a particular distance . In the case of slice.c, there are clearly two possible explanations based on two executions at the same distance from the counterexample: one in which `input1` is altered and one in which `input2` is altered. If multiple explanations at the same distance exist, `explain` will arbitrarily select one. In the event that this choice reflects a way to avoid the *consequences* of an error rather than capturing the faulty code, assumptions must be used to narrow the search space, as described in Section 5.1.1. The Δ-slicing technique assumes that a single explanation has already been chosen. It should be noted that Δ-slicing can sometimes be used to "detect" a bad choice of explanation (as discussed in Section 5.2) in that an explanation may be reduced to a very small set of Δs that clearly cannot contain a fault.

4.2 Computing a Δ-Slice

The same approach used to generate the Δ values can be used to compute an even more aggressive "dynamic slice." In traditional slicing, the goal is to discover all assignments that are relevant to a particular value, either in any possible execution (static slicing) or in a single execution (dynamic slicing). In reporting Δ values, however, the goal is to discover precisely which *differences* in two executions are relevant to a value. Moreover, the value in question is always a predicate (the specification). A dynamic slice is an answer to the question: "What is the smallest subset of this program that always assigns the same values to this variable at this point?" Δ-slicing answers the question "What is the smallest subset of changes in values between these two executions that results in a change in the value of this predicate?"

To compute the Δ-slice, we use the same Δ and pseudo-Boolean constraints as presented above. The constraints on the transition relation, however are relaxed. For every variable v_i such that $\Delta(i) = 1$ in the counterexample with constraint $v_i = expr$, and values val_i^a and val_i^b in the counterexample and closest successful execution, respectively, a new constraint is generated:

$$(v_i = val_i^a) \vee ((v_i = val_i^b) \wedge (v_i = expr))$$

That is, for every value in this new execution that changed, the value must be either the same as in the original counterexample or the same as in the closest successful run. If the latter, it must also obey the transition relation, as determined by the constraint $v_i = expr$. For values that did not change ($\Delta(i) = 0$) the constant constraint $v_i = val_i^a$ is used (which is a simplification of the above constraint, given that we know both sides of the conjunction constrain the value to the same constant).

Consider the SSA form variable `x#3`, which has a value of 12 in both the counterexample (a) and the successful execution (b). The Δ value associated with `x#3` is 0, and so the old constraint[1] for `x#3`, `x#3 == x#2 + 6` is replaced in the Δ-slicing constraints with the constant constraint `x#3 == 12`.

The variable `y#4`, on the other hand, is assigned a value of 12 in the counterexample (a) and a value of 7 in the successful execution (b), and is therefore associated with a Δ value of 1. The constraint for this variable is `y#4 == (\guard#2 ? y#3 : y#2)`. To produce the new constraint on `y#4` for Δ-slicing, we take the general form above and substitute `y#4` for v_i, 12 for val_i^a, 7 for val_i^b, and `(\guard#2 ? y#3 : y#2)` for $expr$:

```
(y#4 == 12) || ((y#4 == 7) && (y#4 == (\guard#2 ? y#3 : y#2)))
```

The "execution" generated from these constraints may not be a valid run of the program (it will not be, in any case where the slicing reduces the size of the Δs). However, no invalid state or transition will be exposed to the user: the only part of the solution that is used is the new set of Δs. These are always a subset of the original Δs. The improper execution is only used to focus attention on the truly necessary changes in a proper execution. The change in the transition relation can be thought of as encoding the notion that we allow a variable to revert to its value

[1]Constraints are always the same for both counterexample and successful execution.

in the counterexample if this alteration is not observable with respect to satisfying the specification.

The Δ-slicing algorithm is:

1. Produce an explanation (a set of Δs) for a counterexample as described in Section 3.2.1.

2. Modify the SAT constraints on the variables to reflect the Δs between the counterexample and the chosen closest successful execution by

   - replacing the constraints for variables in the set of Δs with:

     $$(v_i = val_i^a) \vee ((v_i = val_i^b) \wedge (v_i = expr))$$

   - and replacing the constraints for all other variables with

     $$v_i = val_i^a$$

     (which is the same as $v_i = val_i^b$, in this case).

3. Use PBS to find a new (potentially better) solution to the modified constraint system, under the same distance metric as before.

Figure 4.3 shows some of the original constraints for slice.c. The modified constraints used for computing the Δ-slice are shown in Figure 4.4. The relaxation of the transition relation allows for a better solution to the optimization problem, the

```
 ...
{-7}  \guard#2 == (input2#0 > 0)
{-6}  x#3 == x#2 + 6
{-5}  y#3 == y#2 + 5
{-4}  z#3 == z#2 + 4
{-3}  x#4 == (\guard#2 ? x#3 : x#2)
{-2}  y#4 == (\guard#2 ? y#3 : y#2)
{-1}  z#4 == (\guard#2 ? z#3 : z#2)
|-------------------------
{1}   \guard#0 => x#4 < 10 || y#4 < 10
```

Figure 4.3: Partial constraints for slice.c

```
 ...
{-7}  \guard#2 == (input2#0 > 0)
{-6}  x#3 == 12
{-5}  y#3 == 12
{-4}  z#3 == 9
{-3}  (x#4 == 12) || ((x#4 == 6) && (x#4 == (\guard#2 ? x#3 : x#2)))
{-2}  (y#4 == 12) || ((y#4 == 7) && (y#4 == (\guard#2 ? y#3 : y#2)))
{-1}  (z#4 == 9) || ((z#4 == 5) && (z#4 == (\guard#2 ? z#3 : z#2)))
|--------------------------
{1}   \guard#0 => x#4 < 10 || y#4 < 10
```

Figure 4.4: Δ-slicing constraints for slice.c

```
Value changed:  input2#0 from 1 to 0
Guard changed:  input2#0 > 0 (\guard#2) was TRUE
                line 9 function c::main
Value changed:  y#4 from 12 to 7
```

Figure 4.5: Δ-slice for slice.c

Δ-slice shown in Figure 4.5. Another slice would replace `y` with `x`. It is only necessary to observe a change in *either* `x` or `y` to satisfy the assertion. Δ-slicing produces *either* lines 9 and 10 or 9 and 11 as relevant, while dynamic slicing produces the union of these two routes to a changed value for the assertion.

The Δ-slicer can be used to produce *all* of the possible minimal slices of a set of differences (in this case, these consist of a change to `x` alone and a change to `y` alone), indicating the possible causal chains by which an error can be avoided, when the first slice produced does not help in understanding the error. Additional slices can be produced by adding a constraint to the SAT representation that removes the latest slice from the set of possible solutions (i.e. a blocking clause). The new set of constraints are given to PBS, along with a pseudo-Boolean constraint restricting solutions to those at the same distance as the previous slice(s). This can be repeated (growing the constraints by one blocking clause each time) until the PBS constraints become unsatisfiable, at which point all possible slices have been produced. This division into causal "routes" is not a feature of traditional dynamic slicing.

```
Value changed:  input2#0 from 0 to 1
Guard changed:  least#0 > input2#0 (\guard#3) was TRUE
                file minmax.c line 9 function c::main
Value changed:  most#6 from 0 to 1
```

Figure 4.6: Δ-slice for minmax.c

Revisiting the original example program, we can apply Δ-slicing to the explanation in Figure 3.8 and obtain the smaller explanation shown in Figure 4.6.

In this case, the slicing serves to remove the changes in values deriving from code that is not executed that are introduced by the reliance on SSA form.

4.3 Explaining and Slicing in One Step

4.3.1 Motivation

The slicing algorithm presented above minimizes the changes in a given successful execution, with respect to a counterexample. However, it seems *plausible* that in some sense this is solving the wrong optimization problem: perhaps what we really want is to minimize the size of the final *slice*, not to minimize the pre-slicing Δs. It is not immediately clear which of two possible optimization problems will best serve our needs:

- Find an execution of the program P with minimal distance from the counterexample a. This distance, naturally, may take into account behavior that is irrelevant to the erroneous behavior and will be sliced away.

- Find an execution of the program P that minimizes the number of *relevant* changes to the counterexample a (where relevance is determined by Δ-slicing).

We refer to the second approach as *one-step* slicing, as the execution and slice are computed at the same time. As it turns out, while it is possible to formulate (in pseudo-Boolean terms) and solve the second problem and implement one-step slicing, the technique turns out to perform quite poorly. In Section 4.3.4 we use this surprising result to provide a better understanding of what precisely Δ-slicing accomplishes and how it differs from static slicing: the key insight is that "relevance" is a *static* notion in the case of static slices; in the case of Δ-slicing it is radically dependent on the executions in question in a way that goes even beyond the execution-relative nature of dynamic slicing.

Before returning to the issue of which approach is best, we will demonstrate that solving the second optimization problem is indeed feasible.

4.3.2 Naïve Approach

The simplest approach to computing a one-step slice would be to use the slicing constraints in place of the usual SSA unwinding in the original search for a closest execution. The constraint used in the two phase approach:

$$(v_i = val_i^a) \lor ((v_i = val_i^b) \land (v_i = expr))$$

relies upon a knowledge of val_i^b from an already discovered closest successful execution. Unfortunately, removing this term to produce the constraint:

$$(v_i = val_i^a) \lor (v_i = expr)$$

fails to guarantee that the set of observed changes will be consistent with any actual execution of the program (or even that each particular changed value will be contained in any valid execution of the program).

4.3.3 Shadow Variables

In order to preserve the property that the slice is a subset of an actual program execution, the one-step slicing algorithm makes use of *shadow* variables.

For each assignment in the original SSA, a *shadow* copy is introduced, indicated by a primed variable name. For each shadow assignment, all variables from the original SSA are replaced by their shadow copies, e. g.:

$$v_6 = v_3 + v_5$$

becomes

$$v_6' = v_3' + v_5'$$

and the constraints ensuring a successful execution are applied to the shadow variables. In other words, the shadow variables are exactly the constraints used to discover the most similar successful execution: the shadow variables are constrained to represent a valid successful execution of the program. Using Δs based on the shadow variables would give results exactly equivalent to the first step of the two-phase algorithm, in that the only change is the priming of variables.

The slicing arises from the fact that the distance metric is not computed over the shadow variables. Instead, the shadow variables are used to ensure that the observed changes presented are a subset of a single valid successful execution. The Δs for the distance metric are computed over non-primed variables constrained in a manner very similar to the first Δ-slicing algorithm:

$$(v_i = val_i^a) \vee ((v_i = val_i') \wedge (v_i = expr))$$

with val_i^b replaced by val_i'. Rather than first computing a minimally distant successful execution, the one-step slicing algorithm produces a (possibly non-minimally distant) successful execution as it computes a minimal slice. Because it cannot be known which variables will be unchanged, there are no constant constraints as in the two-step algorithm (recall that the constant constraints are just a simplification of the above expression, in any case).

The Δs are computed over the non-shadow variables using the same distance metric as in both steps of the two phase algorithm. The Δs that are reported to the user use the values from the non-primed variables: however, for all actual changes,

```
 ...
{-12} x#3' == 6 + x#2'
{-11} (x#3 == 12) || ((x#3 == x#3') && (x#3 == 6 + x#2))
{-10} y#3' == 5 + y#2'
{-9}  (y#3 == 12) || ((y#3 == y#3') && (y#3 == 5 + y#2))
{-8}  z#3' == 4 + z#2'
{-7}  (z#3 == 9) || ((z#3 == z#3') && (z#3 == 4 + z#2))
{-6}  x#4' == (\guard#2' ? x#3' : x#2')
{-5}  (x#4 == 12) || ((x#4 == x#4') &&
                     (x#4 == (\guard#2 ? x#3 : x#2)))
{-4}  y#4' == (\guard#2' ? y#3' : y#2)
{-3}  (y#4 == 12) || ((y#4 == y#4') &&
                     (y#4 == (\guard#2 ? y#3 : y#2)))
{-2}  z#4' == (\guard#2' ? z#3' : z#2)
{-1}  (z#4 == 9) || ((z#4 == z#4') &&
                    (z#4 == (\guard#2 ? z#3 : z#2)))
|--------------------------
{1}   \guard#0 => x#4 < 10 || y#4 < 10
{2}   \guard#0' => x#4' < 10 || y#4' < 10
```

Figure 4.7: One-step Δ-slicing constraints for slice.c

this will match the shadow value, which guarantees that all changes are a subset of a valid successful execution. Figure 4.7 shows a subset of the shadow and normal constraints produced for slice.c. In the case of slice.c, slicing in one-step produces no changes: the slice is already minimal.

4.3.4 Disadvantages of One-Step Slicing: The Relativity of Relevance

Interestingly, when the results of one-step and two-phase slicing differ, it is generally the case that the one-step approach produces less useful results. Table 5.4 in Section 5.4 shows the results for applying one-step slicing to various case studies. The one-step approach does not provide a significant improvement in localization over the original counterexamples, and is considerably less effective than the two-phase algorithm (results in Table 5.1): the explanations produced are, on average, of much lower quality, and take longer to produce.

That the two-phase approach is faster is not surprising. The PBS optimization problems in both phases will always be smaller than that solved in the one-step approach (by a factor of close to two, due to the need for shadow variables). The slicing phase is also highly constrained: setting one bit of any program variable may determine the value for 32 (or more) SAT variables, as each SSA form value has only two possible values.

The most likely explanation for the poor explanations produced by one-step slicing is that it solves the wrong optimization problem. In Δ-slicing, "relevance" is not a deterministic artifact of a program and a statement, as it is in static slicing. Instead, relevance is a *function of an explanation*: the Δ-slicing notion of relevance makes sense only in the context of a given counterexample and successful execution. If the successful execution is poorly chosen, the resulting notion of relevance (and hence the slice) will be of little value. Optimizing the size of the final slice is unwise if it is possible for a slice to be small *because* it is based on a bad explanation — and, as shown in Chapter 5, this is certainly possible. It is not so much that optimizing over "irrelevant" changes is desirable, but that it is impossible to know which changes are relevant until we have chosen an execution. Given that the distance metric already precludes irrelevant changes that are not forced by relevant changes, it is probably best to simply optimize the distance between the executions and trust that slicing will remove irrelevant behavior — once we have some context in which to define relevance.

Chapter 5

Case Studies and Evaluation for Concrete Explanation

Of course if I trace the details of how I got here I can come up with an explanation, but on a gut level I'm still not convinced.

- Haruki Murakami, Sputnik Sweetheart

5.1 Case Studies

Two case studies provide insight into how error explanation based on distance metrics performs in practice. The TCAS resolution advisory component case study allows for comparison of fault localization results with other tools, including a testing approach also based on similarity of successful runs. The μC/OS-II case study shows the applicability of the explanation technique to a more realistically sized example taken from production code for the kernel of a real-time operating system (RTOS). The fault localization results for both studies are quantitatively evaluated in Section 5.2.

5.1.1 TCAS Case Study

TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system used by all US commercial aircraft. The Georgia Tech version of the Siemens suite [Rothermel and Harrold, 1999] includes an ANSI C version of the Resolution Advisory (RA) component of the TCAS system (173 lines of C code) and 41 faulty versions of the RA component. A previous study of the component using symbolic execution [Coen-Porisini et al., 2001] provided a partial specification that was able to detect faults in 5 of the 41 versions (CBMC's automatic array bounds checking detected another 2 faults). The inability to detect other faults was a consequence of the partial specification, rather than a failure of the model checker: no more detailed assertional spec for the TCAS code was available. In addition to these assertions, it was necessary to include some obvious assumptions on the inputs[1].

Variation #1 of the TCAS code differs from the correct version in a single line (Figure 5.1). A $\geq$ comparison in the correct code has been changed into a $>$ comparison on line 100. Figure 5.2 shows the result of applying `explain` to the counterexample generated by CBMC for this error (after Δ-slicing). The counterexample

[1]CBMC reports overflow errors, so altitudes over 100,000 were precluded (commercial aircraft at such an altitude would be beyond the aid of TCAS in any case).

```
 100c100
// (correct version)
<       result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          (!(Down_Separation >= ALIM())));
---
// (faulty version #1)
>       result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          (!(Down_Separation > ALIM())));
```

Figure 5.1: diff of correct TCAS code and variation #1

```
Value changed:  Input_Down_Separation#0 from 400 to 159
Value changed:  P1_BCond#1 from TRUE to FALSE
                line 255 function c::main
```

Figure 5.2: First explanation for variation #1 (after Δ-slicing)

```
PrB = (ASTEn && ASTUpRA);
 ...
P1_BCond = ((Input_Up_Separation < Layer_Positive_RA_Alt_Thresh) &&
            (Input_Down_Separation >= Layer_Positive_RA_Alt_Thresh));
assert(!(P1_BCond && PrB)); // P1_BCond -> ! PrB
```

Figure 5.3: Code for violated assertion

passes through 90 states before an assertion (shown in Figure 5.3) fails.

The TCAS system monitors radar data to check for the presence of aircraft that might pose a collision threat by coming too close to the TCAS-equipped aircraft. When this occurs it is said that an *intruder* has entered the *protected zone*. TCAS estimates the time remaining to point of closest approach and calculates a vertical separation between the two aircraft, assuming that the TCAS carrying aircraft either maintains its trajectory or begins an upward or downward maneuver. Depending on this calculation, TCAS may compute a *Resolution Advisory* (RA), instructing the pilot to climb or descend in order to make collision less likely [Coen-Porisini et al., 2001]. The code examined in the case study is the portion of TCAS that is responsible for determining the best RA. The assertion in Figure 5.3 requires that:

If

- the separation between planes if an upward RA is chosen (`Up_Separation`) is less than the safe threshold for separation (`Layer_Positive_RA_Alt_Thresh`)

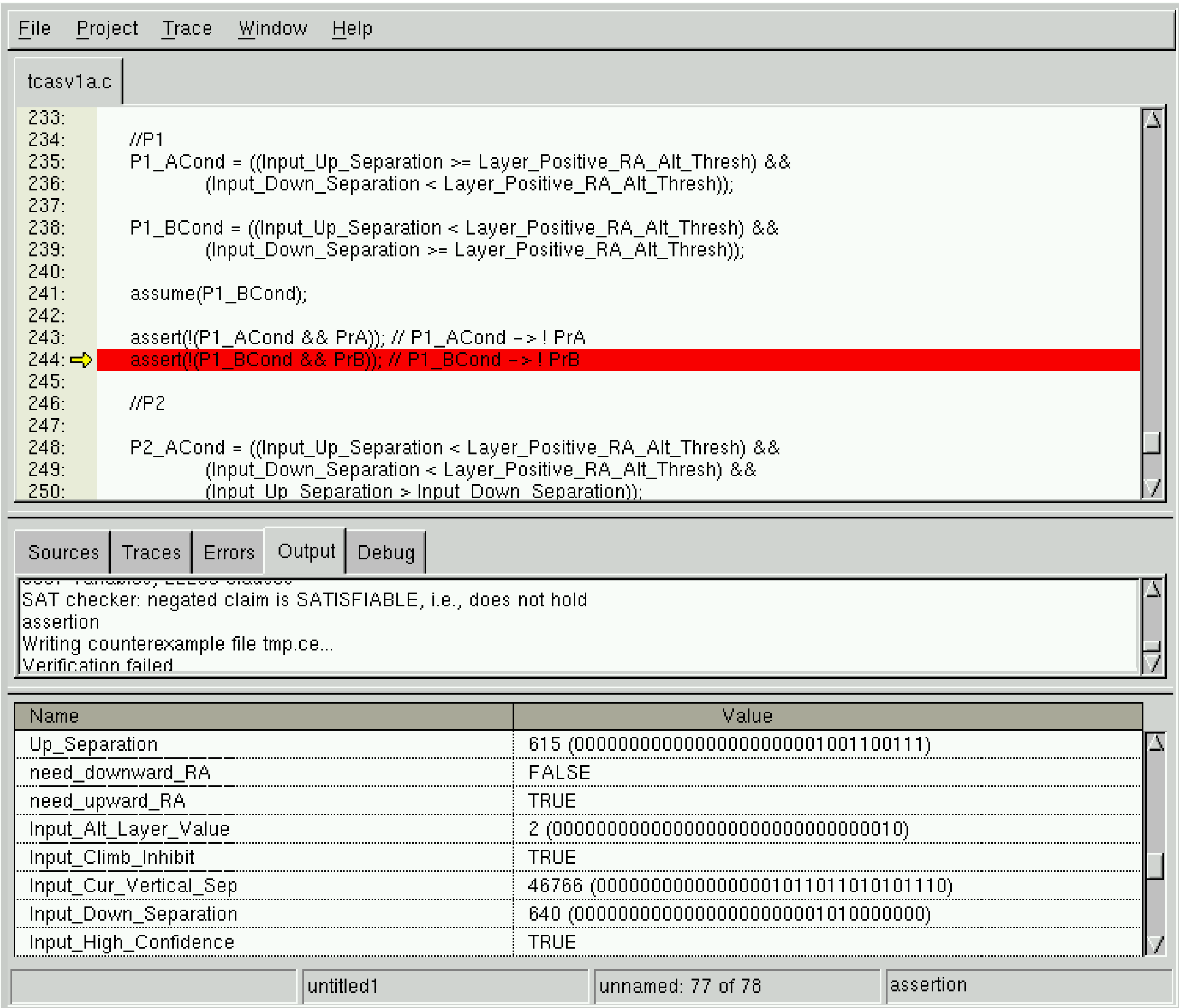


Figure 5.4: Explaining tcasv1.c

and

- the separation between planes if a downward RA is chosen (`Down_Separation`) is greater than or equal to the safe threshold for separation.

(together these conditions constitute `P1_BCond`) then it is not the case that:

- a resolution advisory has been computed (`ASTEn`) and
- a climbing Resolution Advisory (RA) has been selected (`ASTUpRA`)

(together these conditions constitute `PrB`).

In other words, the assertion requires that TCAS will not tell a pilot to climb if climbing *will not* produce an adequate vertical separation to avoid collision and descending *will* produce a safe separation. This is part of the property P1 in the partial specification (P1 also requires the symmetric case to hold when a downward RA is computed) [Coen-Porisini et al., 2001].

The explanation given is not particularly useful. The assertion violation has been avoided by altering an input so that the antecedent of the implication in the assertion is not satisfied — in other words, by changing inputs so that the downward RA is no longer clearly preferable to the upwards RA. This shows that the distance metric-based technique is not always fully automated; fortunately user guidance is easy to supply in this case. We are really interested in an explanation of why the second part of the implication (`PrB`) is true in the error trace, *given* that `P1_BCond` holds: how can TCAS suggest going upwards, when the conditions are such that a downwards RA is preferable? To coerce `explain` into answering this query, we add the constraint `assume(P1_BCond);` to variation #1. After model checking the program again we reapply `explain`. The new explanation (Figure 5.5) is far more useful.

In this particular case, which might be called the *implication-antecedent* problem, automatic generation of the needed assumption is feasible: the tool needs to observe only the implication structure of the failed assertion, and that the successful execution falsifies the antecedent. An assumption requiring the antecedent to hold can then be introduced. Even in this simple case, some kind of programmer annotation might be best, as the syntactic structure of the implication is lost in the transformation to a conjunction — not only must the tool arbitrarily decide which conjunct is an antecedent, but it must make the assumption that negated conjunction "really" represent implications. This issue could be avoided by introducing an operation for implication into the assertion language, or using comments (as above) to indicate implications.

The original counterexample after adding the assumption is still valid, as it clearly satisfies the assumption[2]. As noted in the introduction it is not to be expected that all assumptions about program behavior that are not encoded directly in the specification can be generated by the tool. In some cases, users may need to augment a program with "subtle" assumptions that the distance metric and specification do not capture (e.g. in one of the TCAS cases, the requirement that executions in which

[2]A new counterexample is used in the TCAS example to avoid having to adjust line numbers, but an automatically generated assumption would not require source code modification as the `assume` would not introduce a new program source line.

TCAS does not compute an advisory are unlikely to be useful for understanding errors in computing the advisory). Adding these assumptions doesn't require an understanding of the *error*, only of behavior that is (for some reason) not useful to compare with the counterexample.

Observe that, as in the first explanation, only one input value has changed. The first change in a computed value is on line 100 of the program — the location of the fault! Examining the source line and the counterexample values, we see that `ALIM()` had the value 640. `Down_Separation` also had a value of 640. The subexpression `(!(Down_Separation > ALIM()))` has a value of TRUE in the counterexample and FALSE in the successful run. The fault lies in the original value of TRUE, brought about by the change in comparison operators and only exposed when `ALIM()` = `Down_Separation`. The rest of the explanation shows how this value propagates to result in a correct choice of RA.

The utility of Δ-slicing can be shown by considering the additional values present in the unsliced version of the report (Figure 5.6).

Figures 5.4 and 5.7 show the explanation process as it appears in the `explain` GUI. The error is highlighted in red, and all source lines appearing in the explanation are highlighted in orange. Note that although the counterexample in the screenshots is actually for a different assertion violation (CBMC's initial settings, including the heuristics used to call the SAT solver, determine which counterexample is produced), the localization information is unchanged.

Appendix B provides the full listing of the original counterexample for TCAS Ver-

```
Value changed: Input_Down_Separation#0 from 500 to 504

Value changed: Down_Separation#1 from 500 to 504

               line 215 function c::main

Value changed: result#1 from TRUE to FALSE

               line 100 function c::Non_Crossing_Biased_Climb

Value changed: result#3 from TRUE to FALSE

Value changed: tmp#1 from TRUE to FALSE

               line 106 function c::Non_Crossing_Biased_Climb

Guard changed: \guard#1 && tmp#1 (#7) was TRUE

               line 144 function c::alt_sep_test

Value changed: need_upward_RA#1 from TRUE to FALSE

               line 144 function c::alt_sep_test

Guard changed: \guard#15 && need_upward_RA#1 (#16) was TRUE

               line 152 function c::alt_sep_test

Guard changed: \guard#15 && !need_upward_RA#1 (#17) was FALSE

               line 152 function c::alt_sep_test

Guard changed: \guard#17 && !need_downward_RA#1 (#19) was FALSE

               line 156 function c::alt_sep_test
```

Figure 5.5: Second explanation for variation #1 (after Δ-slicing)

```
Value changed: ASTUpRA#2 from TRUE to FALSE
Value changed: ASTUpRA#3 from TRUE to FALSE
Value changed: ASTUpRA#4 from TRUE to FALSE
Value changed: PrB#1 from TRUE to FALSE
               line 230 function c::main
```

Figure 5.5 (continued)

```
Value changed:  ASTUnresRA#3 from FALSE to TRUE
Value changed:  alt_sep#7 from 1 to 0
Value changed:  ASTUnresRA#4 from FALSE to TRUE
Value changed:  alt_sep#8 from 1 to 0
Value changed:  ASTUnresRA#5 from FALSE to TRUE
Value changed:  result#4 from TRUE to FALSE
Value changed:  alt_sep#9 from 1 to 0
Value changed:  need_upward_RA#2 from TRUE to FALSE
Value changed:  tmp#2 from TRUE to FALSE
Value changed:  r#1 from 1 to 0
                line 166 function c::alt_sep_test
```

Figure 5.6: Values removed by Δ-slicing from report

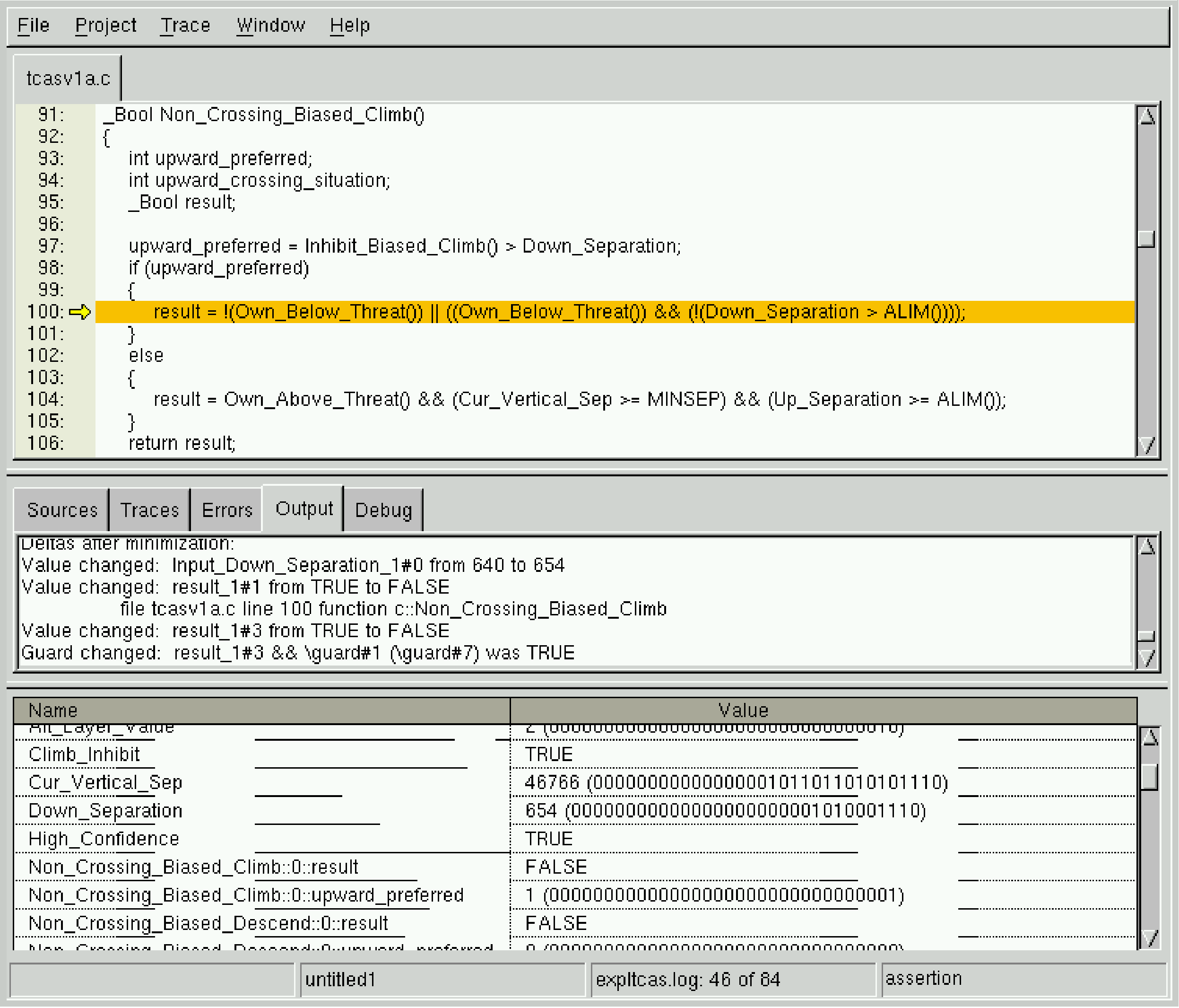


Figure 5.7: Correctly locating the error in tcasv1.c

sion #1. For comparison,the full pre- and post-assumption explanations (including the successful executions generated) are provided in Appendices C and E, respectively. Appendix D provides the new counterexample produced after the assumption is added (the old counterexample remains a valid counterexample, but the modification of the program causes the SAT solver to generate a different solution to the Bounded Model Checking query).

For one of the five interesting[3] variations (#40), a useful explanation is produced without any added assumptions. Variations #11 and #31 also require assumptions about the antecedent of an implication in an assertion. The final variation, #41, requires an antecedent assumption and an assumption requiring that TCAS is enabled (the successful execution finally produced differs from the counterexample to such an extent that changing inputs so as to disable TCAS is a closer solution). The second assumption differs from the *implication-antecedent* case in that adding the assumption requires genuine understanding of the structure and behavior of TCAS. Automation of *this kind* of programmer knowledge of which behaviors are relevant to a particular counterexample (e.g., that comparison to executions in which TCAS does not activate is not very helpful) is implausible.

[3] The two errors automatically detected by CBMC are constant-valued array indexing violations that are "explained" sufficiently by a counterexample trace.

5.1.2 μC/OS-II Case Study

μC/OS-II [μC/OS-II Website] is a real-time multitasking kernel for microprocessors and microcontrollers. CBMC applied to a (now superseded) version of the kernel source discovered a locking protocol error that did not appear in the developers' list of known problems with that version of the kernel. The checked source code consists of 2,987 lines of C code, with heavy use of pointers and casts. The counterexample trace contains 43 steps (passing through 82 states) and complete values for various complex data structures used by the kernel. Reading this counterexample is not a trivial exercise.

Figure 5.8 shows the basic structure of the code containing the error. For this error, the actual conditions in the guards are irrelevant: the error can occur even if various conditions are mutually exclusive, so long as the condition at line 1927 is not invariably false. Figure 5.9 shows the explanation for the error produced by `explain`.

The μC/OS locking protocol requires that the function `OS_EXIT_CRITICAL` should never be called twice without an intervening `OS_ENTER_CRITICAL` call. The code guarded by the conditional on line 1927 (and thus not executed in the successful execution) makes a call to `OS_EXIT_CRITICAL` and sets a value to 1. The explanation indicates that the error in the counterexample can be avoided if the guard on line 1927 is falsified. This change in control flow results in a change in the variable `LOCK` (by removing the call to `OS_EXIT_CRITICAL`) and the variable `error`, which is set by the code in the branch. Δ-slicing removes the change in `error`.

```
1925  OS_ENTER_CRITICAL(); ...
1927  if ( ... ) {
...
1929    OS_EXIT_CRITICAL(); ...
1931    (*err) = 1;
        /*  missing return here! */
1932  } ...
1934  if ( ... ) { ...
1938    OS_EXIT_CRITICAL();
...
1941  } else { ...
1943    if ( ... ) { ...
1945      OS_EXIT_CRITICAL(); ...
1948    } else {
...
1956      OS_EXIT_CRITICAL();
...
1981 return;
```

Figure 5.8: Code structure for μC/OS-II error

```
Guard changed:  ( ... ) && \guard#1 (#2) was TRUE
                line 1927 function c::OSSemPend
Value changed:  LOCK#9 from 0 to 1
Value changed:  error#3 from 1 to 0
```

Figure 5.9: Explanation for μC/OS-II error

The source code for this branch *should* contain a `return` statement, forcing an exit from the function `OSSemPend` (the `return` should appear between the assignment at line 1931 and the end of the block at line 1932); it does not. The missing `return` allows execution to proceed to a condition on line 1934. Both the `if` and `else` branches of this conditional eventually force a call to `OS_EXIT_CRITICAL`, violating the locking protocol whenever the guard at line 1927 is satisfied. `explain` has correctly localized the error as far as is possible. The problem is a code omission, which prevents the explanation from pinpointing the precise line of the error (no change between executions can occur in *missing* source code, obviously), but the explanation has narrowed the fault down to the four lines of code guarded by line 1927.

CBMC produces a counterexample for μC/OS-II in 44 seconds, and `explain` generates an explanation in 62 seconds. Δ-slicing requires an additional 59 seconds, but is obviously not required in this case. The SAT instance for producing a counterexample consists of 235,263 variables and 566,940 clauses. The PBS instance for explanation consists of 236,064 variables and 568,886 clauses, with 69 variables appearing in the pseudo-Boolean constraint. Appendices F and G provide the full counterexample and explanation, including the generated successful execution (a large number of variables representing uninitialized pointer structures are omitted from the appendices).

5.2 Evaluation of Fault Localization

Renieris and Reiss [Renieris and Reiss, 2003] propose a scoring function for evaluating error localization techniques based on program dependency graphs (PDGs) [Horwitz and Reps, 1992]. A PDG is a graph of the structure of a program, with nodes (source code lines in this case) connected by edges based on data and control dependencies. For evaluation purposes, they assume that a correct version of a program is available. A node in the PDG is a *faulty* node if it is different than in the correct version. The score assigned to an error report (which is a set of nodes) is a number in the range 0 - 1, where higher scores are better. Scores approaching 1 are assigned to reports that contain *only faulty nodes.* Scores of 0 are assigned to reports that either include every node (and thus are useless for localization purposes) or contain only nodes that are very far from faulty nodes in the PDG. Consider a breadth-first search of the PDG starting from the set of nodes in the error report R. Call R a *layer*, BFS_0. We then define BFS_{n+1} as a set containing BFS_n and all nodes reachable in one directed step in the PDG from BFS_n. Let BFS_* be the smallest layer BFS_n containing at least one faulty node. The score for R is $1 - \frac{|BFS_*|}{|PDG|}$. This reflects how much of a program an ideal user (who recognizes faulty lines on sight) could avoid reading if performing a breadth-first search of the PDG beginning from the error report. This scoring method has been sufficiently accepted in the fault localization community to be used by Cleve and Zeller in evaluating their latest improvements to the delta-debugging technique [Cleve and Zeller, 2005].

Renieris and Reiss report fault localization results for the entire Siemens suite [Renieris and Reiss, 2003]. Their fault localization technique requires only a set of test cases (and a test oracle) for the program in question. The Siemens suite provides test cases and a correct version of the program for comparison. To apply the `explain` tool a specification must be provided for the model checker; unfortunately, most of the Siemens suite programs have not been specified in a manner suitable for model checking. It would be possible to hard-code values for test cases as very specific assertions, but this obviously does not reflect useful practice — "successful" runs produced might be erroneous runs not present in the test suite. Most of the Siemens programs are difficult to specify using assertions. The TCAS component, however, is suitable for model checking with almost no modification, as it computes an output from a set of inputs and a logical specification [Coen-Porisini et al., 2001] for aspects of this behavior is available.

Table 5.1 shows scores for error reports generated by `explain`, JPF, and the approach of Renieris and Reiss. The score for the CBMC counterexample is given as a baseline. CodeSurfer [Anderson and Teitelbaum, 2001] generated the PDGs and code provided by Manos Renieris computed the scores for the error reports.

The first two columns under the "`explain`" heading show scores given to reports provided by `explain` without using added assumptions, before and after Δ-slicing.

	`explain`			`assume`			JPF		R & R		CBMC	
Var.	`exp`	slice	time	assm	slice	time	JPF	time	n-c	n-s	CBMC	time
#1	0.51	0.00	4	0.90	**0.91**	4	0.87	1,521	0.00	0.58	0.41	1
#11	0.36	0.00	5	0.88	**0.93**	7	0.93	5,673	0.13	0.13	0.51	1
#31	0.76	0.00	4	0.89	**0.93**	7	FAIL	-	0.00	0.00	0.46	1
#40	0.75	**0.88**	6	-	-	-	0.87	30,482	0.83	0.77	0.35	1
#41	0.68	0.00	8	0.84	0.88	5	0.30	34	0.58	**0.92**	0.38	1
Average	0.61	0.18	5.4	0.88	**0.91**	5.8	0.59	7,542	0.31	0.48	0.42	1
μC/OS-II	**0.99**	**0.99**	62	-	-	-	N/A	N/A	N/A	N/A	0.97	44
μC/OS-II*	**0.81**	**0.81**	62	-	-	-	N/A	N/A	N/A	N/A	0.00	44

Explanation execution times in seconds. Best results in boldface. FAIL indicates memory exhaustion ($>$ 768MB used). * indicates alternative scoring method.

Table 5.1: Scores for localization techniques

For versions #1, #11, #31, and #41, the original explanation includes a faulty node as a result of an input change; however, the faulty node is only "accidentally" present in the report, and is removed by slicing. This is not a failure of the slicing algorithm, but a sign that the explanation is poor: the changes required to avoid the error *do not* include a faulty node, but, because the TCAS code includes many dependencies on the inputs, a change in a fault location happens to arise from the input change. In other words, an input change needed to avoid the error affects a result computed in faulty code by sheer coincidence. The coincidental change is then removed by slicing, because it is completely irrelevant to the success of the execution. Because relevance in Δ-slicing is defined with respect to a given execution (i.e., explanation), slicing a bad explanation may produce a very small and clearly useless result, as in these cases — it is relatively easy to rule out changes in inputs only as being a satisfactory localization of a TCAS bug. These "negative" results suggest that Δ-slicing can be used to detect very poor explanations: if nothing "interesting" (possibly faulty) remains after slicing, the original explanation is almost certainly reflecting behavior that we do not want to compare to the counterexample, such as in the implication-antecedent case. The columns under the "`assume`" heading show `explain` results after adding appropriate assumptions, if needed.

The next group of scores and times (under the "JPF" heading) show the results of applying JPF's error explanation tools [Groce and Visser, 2003] to the TCAS example. Because JPF does not produce a single report in the same fashion as `explain`, a combination of results from the various analyses produced by JPF, specifically $only(pos) \cup only(neg) \cup (all(neg) \backslash all(pos)) \cup (all(pos) \backslash all(neg))$ for transitions and transforms, was used to evaluate the fault localization. The details of this computation are somewhat involved, but at a high level this report is based on a sample of successful and failed executions of the program, and contains: (1) nodes appearing in either only successful or only failing runs and (2) those nodes appearing in either all successful but not all failed or all failed but not all successful runs. In order to produce any results (or even find a counterexample) with JPF it is necessary to constrain input values to either constants or very small ranges based on a counterexample produced by CBMC. Comparison with the JPF scores is therefore of somewhat dubious value.

The columns under the "R & R" heading show *average* scores for two of the localization methods described by Renieris and Reiss [Renieris and Reiss, 2003]. The scores for their methods vary depending on which failing test case is used as a basis for computing the localization. For the most part, the difference between the minimum, maximum, and average scores for each variation were small (less than 0.04), except for variation #11, with a maximum score of 0.95 and a minimum of 0.00, producing a low average. The representation of executions used in Renieris and Reiss' approach is that of program spectra [Harrold et al., 2000; Reps et al., 1997] containing basic information on control flow and loop execution, rather than a detailed representation including internal variable values such as is used in `explain`. The many low scores produced by these methods probably indicate *collisions*: cases in which the spectra used are too coarse to distinguish between some failing run and some successful run [Renieris and Reiss, 2003]. Run-times for these methods were not reported by Renieris and Reiss, but should be similar to the time needed to run the various test cases, plus some overhead. Running the test suite takes 1.6 seconds.

Previous to the most recent improvement to delta-debugging [Cleve and Zeller, 2005], the nearest neighbor techniques proposed by Renieris and Reiss had achieved the best fault localization on the Siemens suite examples, and so form a good basis for comparison. We have not yet obtained TCAS-specific data to see how the gains over Renieris and Reiss' results achieved by Cleve and Zeller compare to our improvements: for the suite in general, they report that 45% of test runs received scores less than 0.60, but it is possible that TCAS results differ importantly from these average scores. We do know that our average run-times for TCAS explanation compare favorably to the delta-debugging technique, which took 184.8 seconds on average for TCAS runs.

The last two columns provide a baseline for all results: scores and times for the counterexamples generated by the CBMC model checker. When a method does not improve on the score for the counterexample, that method can probably safely be considered to have been a *hindrance* to debugging efforts in that instance.

After introducing assumptions and slicing, 0.88 was the lowest score for an `explain` report. Ignoring pre-assumption accidental inclusions of faults, Δ-slicing always resulted in improved scores. The best average results for any method are those for the `explain` approach after adding assumptions and slicing.

The μC/OS-II explanation receives a score of 0.99 (Δ-slicing does not change the score in this case). Somewhat surprisingly, the CBMC counterexample itself receives a score of 0.97: it is very short in comparison to the complete source code, and (naturally) passes through the faulty node. Any fault-containing and succinct report will receive a good evaluation for a sufficiently large program. Another useful way to view the results in the μC/OS-II case is that the explanation points directly to the error and contains four lines of results for a user to read. The counterexample also includes the error, but contains over 450 lines of text for a user to understand. Even after removing over 200 lines of program state information, the counterexample contains over 220 lines. Reading from the end of counterexample, 30 lines (from state 82 to state 65) must be read before encountering the faulty node. It is presumably far less likely that the user will grasp the significance of this branch when it is not presented in isolation. To remedy the difficulty in distinguishing report quality for large programs, a modified formula suggested by Manos Renieris uses the size of the counterexample as a baseline in the formula, in place of $|PDG|$: $1 - \frac{|BFS_*|}{|CE|}$. Using this formula (results marked with a * in Table 5.1), the counterexample itself receives a score of 0.00[4], and the μC/OS-II explanation is given a score of 0.81 (a perfect explanation would receive a score of 0.95, as it must contain at least one node: even the best explanation cannot reduce the user's required reading below 5% of the original counterexample nodes).

5.3 Evaluation of Modifications to the Distance Metric

Given the small number of realistic examples available for study, it would be unwise to draw strong conclusions about the utility of fine-tuning the distance metric. However, it is worth examining the effects of a few natural variations on the basic distance metric presented in Chapter 3.

5.3.1 Measuring Distance Over Input Changes Only

One obvious modification to the metric is to restrict the computation of distances to a coarser granularity, such as would be available in testing. Using only input variables to compute the distance between executions (but using all Δs to compute the localization, as input declarations are unlikely to be the source of error) should indicate to some extent the source of the advantages of our approach over testing-

[4] In principle, a report could receive a negative score if it did not contain a faulty node; the counterexample will always receive a score of 0.00, as it is the same size as itself and must contain a faulty node.

	explain			assume			CBMC	
Var.	exp	slice	time	assm	slice	time	CBMC	time
#1	0.83	**0.88**	5	0.84	**0.88**	4	0.41	1
#11	0.46	0.46	4	0.51	**0.54**	5	0.51	1
#31	0.84	0.74	4	0.84	**0.91**	4	0.46	1
#40	0.84	**0.86**	4	-	-	-	0.35	1
#41	0.62	0.42	4	0.82	**0.84**	4	0.38	1
Average	0.72	0.67	5.3	0.75	**0.79**	4.3	0.42	1
μC/OS-II	0.96	**0.97**	62	-	-	-	0.97	44
μC/OS-II*	0.52	**0.62**	62	-	-	-	0.00	44

Table 5.2: Scores for metric over inputs-only

based analysis. If the primary benefit is the ability to search all executions, then measuring over only inputs should still produce good results; if the high precision with respect to internal behavior is the key factor, measuring distance over the externally observable inputs only should result in localization more like that produced by the technique of Renieris and Reiss.

Figure 5.2 shows the results of computing the distance metric (but not the localization) over inputs only. For the TCAS variations, the average evaluation is 0.81, considerably lower than the 0.90 for the more precise distance metric. It is interesting to note that the pre-assumption results for measuring over inputs alone were actually better than those for the original metric — because all of the TCAS explanations, for either metric, are the result of exactly one input change, the coarser metric allows for "more distant" solutions that are masked by the unfortunate implication-assumption executions when using the very precise metric. However, the inputs-only metric fails to localize variation #11, even when an assumption is added. In general, the TCAS results suggest that the most important gain over testing is probably in the ability to consider all possible executions, but that the more precise distance metric also contributes significantly to the success of localization. The μC/OS-II results also show a considerable loss of localization effectiveness when using the coarser metric, particularly when using the counterexample as a baseline: the localization is still better than that provided by the counterexample, but is not as accurate as when using the fine-grained metric.

5.3.2 Increasing the Weight for Input Changes

As an alternative to measuring distance in terms of input changes alone, we can simply increase the *weight* of input changes in the metric computation. Rather than defining $d(a, b)$ as simply:

$$\sum_{i=0}^{n} \Delta(i)$$

where

$$\Delta(i) = \begin{cases} 0 \texttt{ if } & val_i^a = val_i^b \\ 1 \texttt{ if } & val_i^a \neq val_i^b \end{cases}$$

we can use a weighted set of Δs:

$$\Delta(i) = \begin{cases} 0 \texttt{ if } & val_i^a = val_i^b \\ w \texttt{ if } & val_i^a \neq val_i^b \end{cases}$$

where w depends on whether the change is in a program variable, an input, or a guard.

If we make w 100 in the case of inputs and 1 in all other instances (thus weighting changes to inputs 100 times more heavily in the optimization problem than other alterations to the counterexample), we observe surprisingly little change in the explanations produced. For the TCAS variations, only #11 — and it only after the addition of assumptions — changes sufficiently to affect the localization (in other cases the specific values to which inputs are altered change, but not so as to affect either localization or the basic explanation). For #11, the heavier weighting of inputs reduces the quality of the localization: the report receives before and after slicing scores of 0.46 and 0.49 respectively (vs. 0.88 and 0.93 for the original metric). The μC/OS-II explanation is unchanged by the alteration of the distance metric.

As stated above, drawing final conclusions about the utility of changes to the metric is difficult without comparing results over a much larger variety of programs and errors. It is safe to say that at present there is little indication that a heavier weighting of input changes will contribute to better explanations, and some small indication that such weighting might reduce the effectiveness of the technique.

5.3.3 Increasing the Weight for Control Flow Changes

Another plausible alteration to the distance metric would be to make changes in *control flow* contribute more to the distance between executions than changes to program variables. After all, a change in control flow *is* presumably more important

	explain		assume		CBMC	
Var.	`exp`	time	assm	time	CBMC	time
#1	0.33	46	0.33	61	**0.41**	1
#11	0.71	26	**0.72**	23	0.51	1
#31	0.00	38	**0.67**	54	0.46	1
#40	**0.94**	9	-	-	0.35	1
#41	0.00	12	0.29	65	**0.38**	1
Average	0.40	26.2	**0.50**	50.8	0.42	1
μC/OS-II	0.62	116	-	-	**0.97**	44

Table 5.3: Scores when interventions are allowed

than a variable value change. At the least, this seems to be the case in the "intuitive" informal measures of similarity we use when speaking of program executions.

Increasing the weight for guard Δs (i.e., changes in control flow) to **100 times** that for other changes does not result in *any* changes in localizations for the TCAS or μC/OS-II examples. It seems possible that weighting control flow more heavily might be useful in some cases, but no evidence for this can be found in our experiments.

5.3.4 Allowing Arbitrary Value Changes (Interventions)

Instead of changing the distance metric, it is also possible to relax our definition of an *execution* of the program (as discussed in Section 3.1.2) by allowing *interventions*: arbitrary injections of new values. Interventions resemble the relaxation of the transition relation introduced by Δ-slicing, but are more radical: no restrictions are made as to the interjected values, and the explanation is based on a potentially completely unrealistic program execution, rather than a subset of a valid execution.

Figure 5.3 shows results when the restriction to valid executions is relaxed by allowing interventions. We omit Δ-slicing results, as when interventions are allowed, Δ-slicing can provide no further reduction to an explanation (there are no secondary effects of input changes, etc. — all changes are required in order to avoid the error). The time taken to produce explanations increases considerably, and the average quality of the explanations is greatly reduced. In one case (TCAS variation #40), a very good explanation is obtained, but in three cases the explanations produced (even in the post-assumption case for TCAS variations) score substantially *worse* than the localization provided by the CBMC counterexample. The μC/OS-II explanation is of such poor quality that we omit the alternative scoring method, as the localization is clearly much worse than for the counterexample when the score is based on the full PDG (this localization merits a *negative* score by the alternative method).

5.4 Evaluation of One-Step Slicing

Table 5.4 shows the poor results obtained by applying one-step slicing (Section 4.3) to the case studies. Execution times are on average slightly over 4 times greater for the TCAS results (and > 3.5 times longer for the μC/OS-II example, which includes lengthy parsing and processing times). More importantly, the explanations produced are of much lower quality. Without assumptions, the average quality drops *below* that of the raw counterexamples. With assumptions, the explanations are only slightly better than the counterexamples, on average. Averaging the best results overall gives a score of 0.55, while for the two-phase algorithm, the average is a respectable 0.91.

	`explain`		`assume`		CBMC	
Var.	`exp`	time	assm	time	CBMC	time
#1	0.00	26	0.33	26	**0.41**	1
#11	0.46	18	0.46	16	**0.51**	1
#31	0.00	31	**0.81**	29	0.46	1
#40	**0.84**	15	-	-	0.35	1
#41	0.00	27	0.33	26	**0.38**	1
Average	0.26	23.4	**0.48**	24.3	0.42	1
μC/OS-II	0.00	223	-	-	0.97	44
μC/OS-II*	0.00	223	-	-	0.00	44

Table 5.4: Scores with one-step slicing

The problems with one-step slicing arise in part from the ability to avoid an error by changing only an input value and a very small number of intermediate values. The SSA form allows most of the computational changes produced by such an alteration to (correctly) be sliced away, but computing the distance metric over this tiny slice is meaningless, given that the original executions were radically different. As explained previously, Δ-slicing is relevant to an already chosen explanation; in the case of a poor explanation (indicated by a large pre-slice distance), the definition of relevance will result in a slice that provides poor explanation and localization.

5.4.1 One-Step Slicing in Action

Consider, for example, the explanation produced for the TCAS variation #1 by one-step slicing (Figure 5.10).

The code shown in Figure 5.11 is used to determine if a Resolution Advisory

```
Value changed:  Input_Other_Capability_1#0 from 2 to 1
Value changed:  Other_Capability#1 from 2 to 1
                line 217 function c::main
Value changed:  tcas_equipped_1#1 from FALSE to TRUE
                line 136 function c::alt_sep_test
Value changed:  ASTEn#2 from TRUE to FALSE
Value changed:  PrB#1 from TRUE to FALSE
                line 230 function c::main
```

Figure 5.10: One-step slicing report for TCAS variation #1

is computed by TCAS: the properties for TCAS are, in a sense, predicated on the assumption that `ASTEn` is set to true (indicating a resolution has been computed). The implication in the assertion (`P1_BCond` $\Rightarrow$ `!PrB`) is always satisfied if no advisory is computed, because this will force `PrB` to be false (see Figure 5.3). The change in this explanation results in the `if` branch in this code not being taken. Although this causes a large change in the program values, the slicing algorithm correctly notes that the only value crucial for the property change is the alteration to the value of `ASTEn` used in computing `PrB`.

Similar issues result in poor explanations for the other TCAS examples. It might well be *possible* to generate good explanations with one-step slicing in its current form, but the need to introduce a large number of user-produced assumptions makes the technique of very limited value, given the better performance of two-phase slicing.

```
_Bool enabled, tcas_equipped, intent_not_known;
_Bool need_upward_RA, need_downward_RA;
int alt_sep;
ASTBeg = 1;
enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) &&
          (Cur_Vertical_Sep > MAXALTDIFF);
tcas_equipped = Other_Capability == TCAS_TA;
intent_not_known = Two_of_Three_Reports_Valid && Other_RAC ==
  NO_INTENT;
alt_sep = UNRESOLVED;
if (enabled && ((tcas_equipped && intent_not_known)
                || !tcas_equipped))
{
    ASTEn = 1;
```

Figure 5.11: Code for determining if RA is computed

In practice, it appears that computing distances over complete executions is simply better than optimizing the Δ-slices, in the absence of some fundamental reworking of one-step slicing.

Chapter 6

Causal Dependence and Explanation

Your 'if' is the only peacemaker; much virtue in 'if'.

- William Shakespeare, As You Like It, Act 5, Scene 4

6.1 Hypothesizing and Checking Causal Dependence

Chapter 1 presents a notion of causal dependence (Definition 3) based on David Lewis' counterfactual theory of causality [Lewis, 1973a]. That definition states that a predicate e is causally dependent on a predicate c in an execution a iff:

1. $c(a) \wedge e(a))$

2. $\exists b \,.\, \neg c(b) \wedge \neg e(b) \wedge (\forall b' \,.\, (\neg c(b') \wedge e(b')) \Rightarrow (d(a,b) < d(a,b')))$

In other words, e is causally dependent on c in a if the cause and effect both appear in a and executions in which neither the cause nor effect appear are more like a than executions featuring only the cause.

The technique presented in chapters 3-5 does not rely on checking causal dependence. Determining if e depends on c is only useful *after* arriving at a likely candidate cause. This would be putting the cart before the horse, as the chief goal of error explanation is to help the user move from awareness of the existence of an error to a small set of candidate causes.

6.1.1 Motivation

Unfortunately, differences in actual variable values are often *too specific.* The relevant information is often a change in *relationships* between variables: i.e., not that `x` was 100 and must be changed to 200 to avoid violating an assertion, but that in the failing run `x < y` and in the successful run, `x > y`. The standard `explain` approach may, unfortunately, completely omit `y` from an explanation if only the value of `x` is altered in the successful execution. Because the distance metric minimizes the number of changes, such omissions are very likely to occur. A more general notion of Δs would report to the user all *predicates* whose values are different for the counterexample and the successful execution; however, as the set of changed predicates is potentially infinite (comparisons of variables with constant values, etc.), only a subset of the potential Δs can realistically be considered.

Directly presenting the set of changed Δ predicates is not particularly useful: changes in important variables are likely to introduce many accidental and unimportant changes, hiding the relevant differences in a large set of uninteresting results. However, the set of changes can be used as a set of *candidate causes* for *checking causal dependence*. Presenting only Δs on which the error is causally dependent results in a practically applicable method. We can make use of the same mechanisms used to produce the explanations in earlier chapters to automatically hypothesize causes for an error.

In order to reduce the space of predicates to be explored, we restrict our attention to ordering and equality relations between program variables, e.g. `x == y`, `x < y`, `x > y`, `x <= y`, etc. Section 6.1.4 discusses alternative schemes for exploring the space of potential causes for an error.

The set of predicate Δs that need to be checked is further reduced by requiring that one of the variables being compared has changed its value in the successful execution. This is a conservative reduction (ignoring no potential causes): if neither variable has changed value, the predicate value must be unchanged.

6.1.2 Algorithm for Checking Causal Dependence

Given a possible cause c, the counterexample execution a, and an error (or effect) e, checking causal dependence requires two steps:

1. Find an execution b such that:

- c does not hold and
- the distance $d(a, b)$ is minimal.

In order to find b, we make use of the same PBS-based approach used to find a maximally similar successful execution (described in Chapter 3), except that in place of the verification condition V, the negation of the hypothesized cause is used ($\neg c$).

The execution b is as similar as possible to the counterexample a, except that the potential cause c is present in a but not in b. **If the error e is present in b, e is not causally dependent on c and the algorithm terminates.**

2. Perform bounded model checking over all executions such that

   - c does not hold and
   - the distance to a is equal to $d(a, b)$.

   Again, this is made possible by the distance metric representation presented in Chapter 3. The PBS constraints encode the restriction on the distance to a (in this case an exact distance rather than an optimization problem), and we use $\neg c \wedge e$ as the claim. If this PBS formula is unsatisfiable, the error cannot occur without the cause in executions at distance d from a.

   If all such executions are error free (e does not hold, i.e., the PBS formula is unsatisfiable), then e is causally dependent on c.

```
void f (int a, int b, int c)
{
  int temp;
  if (a > b) {
    temp = a;
    a = b;
    b = temp;
  }
  if (b > c) {
    temp = b;
    b = c;
    c = temp;
  }
  if (a < b) {
    temp = a;
    a = b;
    b = temp;
  }
  assert ((a <= b) && (b <= c));
}
```

Figure 6.1: sort.c

```
Counterexample:

Initial State
----------------------------------------------------
  temp=-1 (11111111111111111111111111111111)
  a=0 (00000000000000000000000000000000)
  b=0 (00000000000000000000000000000000)
  c=-1 (11111111111111111111111111111111)
State 6 file sort.c line 10 function c::f
----------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 7 file sort.c line 11 function c::f
----------------------------------------------------
  b=-1 (11111111111111111111111111111111)
State 8 file sort.c line 12 function c::f
----------------------------------------------------
  c=0 (00000000000000000000000000000000)
Failed assertion: assertion file sort.c line 19 function c::f
```

Figure 6.2: Counterexample for sort.c

```
Deltas after minimization:
Value changed:  c#0 from -1 to 0
Guard changed:  !(b#2 <= c#0) (\guard#2) was TRUE
                file sort.c line 9 column 2 function c::f
Value changed:  b#4 from -1 to 0
Value changed:  b#6 from -1 to 0
```

Figure 6.3: Explanation for sort.c

```
Error is causally dependent on these predicates:
  c#0 < a#0
  c#0 < b#0
```

Figure 6.4: Causes for sort.c

```
Error is causally dependent on these predicates:

  Input_Down_Separation#0 == Layer_Positive_RA_Alt_Thresh#1

  Input_Down_Separation#0 <= Layer_Positive_RA_Alt_Thresh#1

  Down_Separation#1 == Layer_Positive_RA_Alt_Thresh#1

  Down_Separation#1 <= Layer_Positive_RA_Alt_Thresh#1
```

Figure 6.5: Causes for TCAS error #1

6.1.3 Checking Causal Dependence in Practice

Figure 6.4 shows a subset of the causes discovered for the counterexample shown in Figure 6.2 (the sliced explanation for the error appears as Figure 6.3). In this case, the only causes shown are those which relate two *input* values. The algorithm actually detects 63 additional causes, relating inputs to intermediate values, or intermediate values to each other. For this reason, an option is provided to only check for relationships between input variables. The high degree of causal dependence in this case derives from the nature of the code: for a faulty sorting routine, ordering relations will obviously be crucial to the occurrence of the error, unless the sorting routine is invariably incorrect. The relationships between intermediate values are somewhat uninteresting in this case, as the set of input values is equivalent to the set of all values computed by the program.

For variation # 1 of the TCAS case study [Coen-Porisini et al., 2001; Rothermel and Harrold, 1999] a much smaller set of causes (Figure 6.5) is produced without restriction to input values. Figure 5.1 shows the error in the TCAS code as a diff between correct and incorrect versions. The automatically generated explanation, as described in Chapter 5, directs attention to line 100. The function call to `ALIM()` on this line always returns a value that is equal to `Layer_Positive_RA_Alt_Thresh#1`. Any user familiar with the specification of the TCAS code will be aware of this equivalence. Knowing (i) that the fault can be localized to line 100 and (ii) that the error is causally dependent on the predicate:

```
Down_Separation#1 == Layer_Positive_RA_Alt_Thresh#1
```

a user should be able to quickly conclude that the `>` comparison on line 100 should be a `>=` comparison.

Unfortunately, checking for causal dependency is considerably more expensive than the basic explanation approach. Producing and slicing the closest execution for TCAS variation #1 takes only 4 seconds; it takes 74 seconds to produce and check a set of possible causes for the error. The additional cost of computing and checking hypotheses varies, taking 207 more seconds than computing only a set of Δs for variation #11, adding 63 seconds for #31, 64 more seconds for #40, and an additional 353 seconds for variation #41.

6.1.4 Alternative Approaches for Hypothesis Selection

The particular choice of predicates for which to check causal dependence involves a number of trade-offs: using too many predicates will increase computation time and may result in redundant results; using too few may miss causal dependencies. An obvious alternative method is to use predicates taken from guards and Boolean assignments in the program source. Such comparisons should be generalized: if `x > y` appears in a guard, checking `x <= y`, `x == y`, and so forth is necessary to catch cases where the choice of comparison operations is incorrect. The primary differences between this generalization and the method implemented in CBMC is that no causality checking is done for (1) comparisons with constants and (2) comparisons with temporary results that are never stored in a variable (i.e. `x > (y + 50)`). On the other hand, comparisons between values that do not appear in guards together are checked. Causal dependencies that are directly present in a guard in the source code are generally not as difficult to detect as indirect dependencies: a change in guard value is likely to appear in the explanation. For this reason, it seems at least reasonable to expect that the current trade-off is often the correct choice.

Another alternative approach would be to leverage predicate abstraction. Chapter 7 presents an explanation technique based on abstract executions of programs. The predicates used in the abstract model could be tested for causal dependence. Checking causal dependence is of less interest when using abstract executions, however, as the explanations are presented in terms of changes in relationships between variables in the first place, and irrelevant Δs in predicates are suppressed by the metric and the abstract model.

Chapter 7

Explaining Abstract Counterexamples

The difference between these two abstractions consists in the fact that in the abstraction of the universal from the particular, that from which the abstraction is made does not remain. . .

- St. Thomas Aquinas, Summa Theologica, I, 40.

7.1 Motivation

The explanation method presented in Chapter 3 may be generalized to apply to other model checkers and representations for executions (Figure 7.1):

1. Generate a counterexample C for a specification $Spec$ of a program P using a

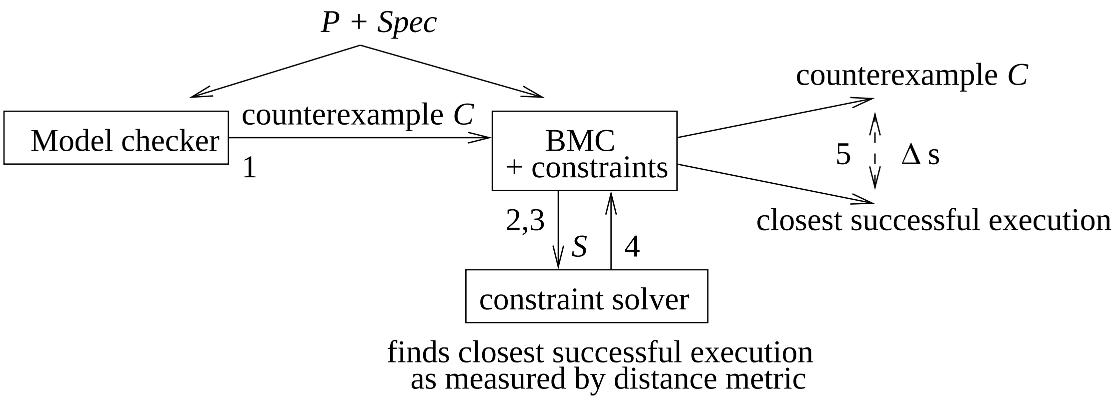


Figure 7.1: Error explanation with distance metrics

model checker[1].

2. Use bounded model checking (BMC) [Biere et al., 1999] to unwind the transition relation of P to a finite bound[2] and produce a propositional formula S that represents exactly the executions of P that do *not* violate *Spec*.

3. Extend S with variables and constraints representing an optimization problem: find a satisfying assignment that is *as similar as possible to the counterexample* C, as measured by a *distance metric* on executions of P.

4. Solve the optimization problem from the previous step, producing a successful execution with minimal distance from the counterexample.

5. Present the differences (Δs) between the successful execution and the coun-

[1] The counterexample now may be a finite path or a stem and cycle representation of an infinite path containing a loop.

[2] In Chapter 3-6, this was typically the same as the bound used to produce the counterexample C.

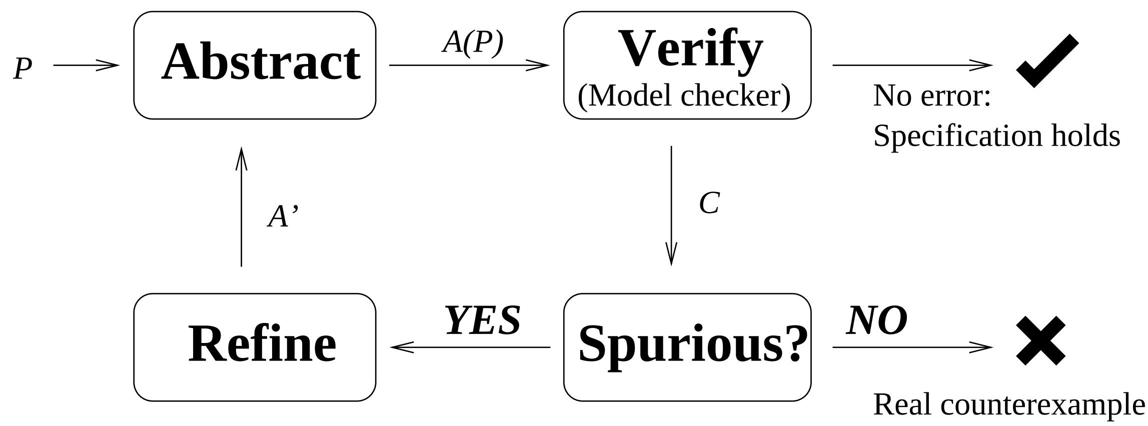


Figure 7.2: Counterexample-guided abstraction refinement (CEGAR)

terexample as *explanation and localization* for the error.

In Chapters 3-6, the counterexample and successful execution were always *concrete executions* produced by the bounded model checker CBMC [Kroening et al., 2004] and the `explain` tool [Groce et al., 2004]. The Δs between successful and failing runs were presented as changes at the level of the C type system, e.g. `x = 2147483615` vs. `x = 255`. In this Chapter, we will preserve the *structure* of the explanation method presented (and the use of BMC + pseudo-Boolean constraints), but generalize to *Delta*s over logical predicates, e.g. `x > y` vs. `x <= y`.

7.1.1 Predicate Abstraction

Many successful software model checking projects, such as SLAM, BLAST, and MAGIC [Ball and Rajamani, 2001; Chaki et al., 2004a; Henzinger et al., 2002] have been based on predicate abstraction [Graf and Saidi, 1997] and counterexample-guided abstraction refinement (CEGAR) [Ball and Rajamani, 2000; Clarke et al., 2000a; Kurshan, 1995]. Rather than model checking a representation of the concrete state-space of a system, these tools check properties of conservative abstractions of programs, and refine the abstractions until either the program is shown to satisfy its specification or a counterexample is generated. The CEGAR framework for verifying a program P with specification *Spec* consists of four main steps (shown in Figure 7.2):

1. **Abstract:** Create a (finite-state) abstraction $A(P)$ which safely abstracts P by construction.

2. **Verify:** Check if $A(P) \models Spec$ holds. That is, determine whether the abstracted program satisfies the specification of P. If it does, P must also satisfy the specification, and the program is successfully verified.

3. **Check spurious:** If $A(P)$ does not satisfy the specification, a counterexample C is generated. C may be *spurious*: not a valid execution of the concrete program P. If C is not spurious, P does not satisfy its specification.

4. **Refine:** If C is spurious, refine $A(P)$ in order to eliminate C, which represents behavior that does not agree with the actual program P. Return to step 1[3].

SLAM, BLAST, and MAGIC use abstraction because concrete state-spaces are often intractably large (or infinite). The reduced state-spaces produced by predicate

[3] This process may not terminate, as the problem is in general undecidable.

abstraction have not, typically, been viewed as useful objects for human examination. They are artifacts of the verification process, used for refuting or proving a property of a system and then discarded. These automatically generated abstractions are usually more complex and less intuitive than those produced by humans, and the state-spaces are still generally too large to be presented directly to users. Nonetheless, we show that these automatically generated abstractions are useful for program understanding, and that the *predicates* produced by the verification process can enhance error explanation.

Abstract error explanation, described in detail below, is a *selective* use of automatically generated predicate abstractions. Even though the abstracted *program* may not be useful or interesting to a user, the *differences* in predicate values between successful and faulty executions of a program may be very useful and interesting. The key insight is that while the abstracted program is very complex, the *predicates* produced by the abstraction process are highly meaningful, and often encode a concise logical understanding of the behavior of the system with respect to a property. Far from being meaningless by-products of verification, the predicates are the values that must be known to distinguish real (faulty) runs of the program from spurious behaviors that do not reflect actual execution. As we show, predicates sufficient to find a non-spurious counterexample also describe non-spurious successful behavior well in many cases.

```
int main () {
  int input1, input2, input3;
  int least = input1;
  int most = input1;
  if (most < input2)
    most = input2;
  if (most < input3)
    most = input3;
  if (least > input2)
    most = input2;                  //ERROR!
  if (least > input3)
    least = input3;
  assert (least <= most);
}
```

Figure 7.3: minmax.c

```
Value changed:  input3#0 from 1 to -2
Value changed:  most#3 from 1 to -2
                line 8 function c::main
Guard changed:  least#0 > input3#0 (\guard#4) was FALSE
                line 11 function c::main
Value changed:  least#1 from 1 to -2
                line 12 function c::main
Value changed:  least#2 from 1 to -2
```

Figure 7.4: Concrete Δ values for minmax.c

7.1.2 Motivating Example

As a motivating example, consider the program in Figure 7.3, presented previously in Chapter 3. We have implemented error explanation (as described below) for the MAGIC [Chaki et al., 2003a, 2004a] predicate-abstraction based software model checker, and applied the `explain` tool and MAGIC's error explanation facility to this program in order to produce a concrete and an abstract explanation for the fault in minmax.c.

Again, recall that the uninitialized values `input1, input2,` and `input3` represent nondeterministically chosen inputs in both MAGIC and `explain`, and that line 13 provide the program specification, requiring that the (supposed) minimum value among the three inputs determined by the program must be less than or equal to

```
Control location deleted (step #5):
  10:  most = input2
  {most = [ $0 == input2 ]}
------------------------
Predicate changed (step #5):
  was:  most < least
  now:  least <= most
Predicate changed (step #5):
  was:  most < input3
  now:  input3 <= most
------------------------
Predicate changed (step #6):
  was:  most < least
  now:  least <= most
Action changed (step #6):
  was:  assertion_failure
------------------------
```

Figure 7.5: Abstract Δ values for minmax.c

the (supposed) maximum value the program discovers.

Figure 7.4 shows a concrete explanation of the error produced by the `explain` tool. The explanations produced by `explain` can be highly sensitive to the particular options chosen when producing the constraints for the SAT solver. CBMC, for reasons of efficiency, provides a number of command line options that modify the equations generated by the SSA-like transformation: constant propagations, arithmetic simplifications, and variable substitutions are all controllable, and it is often difficult to predict which choices will result in the most easily solvable SAT formulas. In this case (though not in any of our larger case studies), applying certain (semantics-preserving) arithmetic simplifications provided by CBMC results in a different explanation than the one shown in Chapter 3. For the same counterexample, this choice of options produces a (rather difficult to understand) explanation that notably does not successfully isolate the error to line 10. Recall that many successful executions may exist at a minimal distance from a counterexample. For this program, using the same options for `explain` as used to produce the counterexample (Figure 3.3 results in a good explanation (Figure 3.8), while slightly altering the CBMC options (perhaps in order to speed up the explanation by applying further simplifications to the constraints) produces an equidistant trace that does not provide a good explanation. Interestingly, adding an assumption that the input values are all `>= 0` also causes `explain` to produce this weak explanation (except that the `-2` is changed to a `0`); adding such an assumption does not alter the results for abstract explanation. The problem of multiple closest executions is less likely to appear in the abstract domain, as the number of possible program executions is (typically) much reduced by abstraction. The importance of this concern is not clear: for non-toy programs with larger executions (and proportionally more behavior irrelevant to an error) the issue of multiple executions at the same distance did not generally result in poor explanations.

Figure 7.5 shows the explanation produced by MAGIC using abstract Δs[4]. The explanation consists of a set of atomic changes to the counterexample produced by MAGIC. In the counterexample, line 10 (the line with the error) is executed. In the most similar successful execution, line 10 is not executed (a control location in the counterexample is *deleted* – an explanation may also show an *insertion*, in which a guard that was not satisfied in the counterexample becomes true in the successful execution). The change in control flow does not result from a change in predicate values; the abstraction is imprecise, and so the guard (`least > input2`) is a nondeterministic choice. The change in control flow forces a change in predicate values: if `least > input2`, then `input2` is assigned to `most`. Given that `least > input2` and `most = input2`, it follows that `least > most`, which will cause the assertion on line 13 to be violated. If the guard is false and the assignment does not take place, the abstraction is precise enough to prove the invariant `least <= most`[5], preventing the `assertion_failure` action[6].

Note that MAGIC does not use a single monolithic set of predicates. A different

[4]Output is slightly simplified for readability.

[5]The other predicate change is a result of `least` being equal to `input3` at this point.

[6]In MAGIC, actions are events that might appear in a specification, such as calls to obtain locks, function return values, and assertion violations [Chaki et al., 2004a]. The `epsilon` action represents unobservable behavior.

set of predicates may be tracked at each program control location. Notice that at state 1 in the counterexample, the predicate `input2 < least` is tracked, but that the relationship between `input2` and `least` is not determined by the predicates in state 3. The particular predicates used at each location are computed by an algorithm [Chaki et al., 2004b, 2003b] based on iterating weakest preconditions [Dijkstra, 1973; Hoare, 1983] to determine which predicates to associate with each control location (similar to the approach of Namjoshi and Kurshan [Namjoshi and Kurshan, 2000]).

Figure 7.6 shows the entire abstract state space of minmax.c, as generated by MAGIC. Figures 7.7 and 7.9 show the counterexample and closest successful execution graphs produced by MAGIC (with actions removed for clarity). Figures 7.8 and 7.10 are more readable textual representations of the paths.

The abstract explanation is produced more quickly[7] and highlights precisely the nature of the error. The most similar successful execution avoids performing the assignment at line 10, which ensures that the assertion holds: the fault is clearly localized to line 10, and the predicates pinpoint the nature of the problem. The concrete explanation, in contrast, presents changes to input values that do not immediately indicate the nature of the problem. The control flow is altered, but in a way that affects non-faulty code, in part because of the distance metric's comparison of values from non-executed code.

Because software model checkers use conservative abstractions, a non-spurious counterexample *can* (in principle) be produced from even a very coarse abstraction

[7]The SAT instances are much smaller, as the 32-bit integers in the concrete case are replaced by a few predicates.

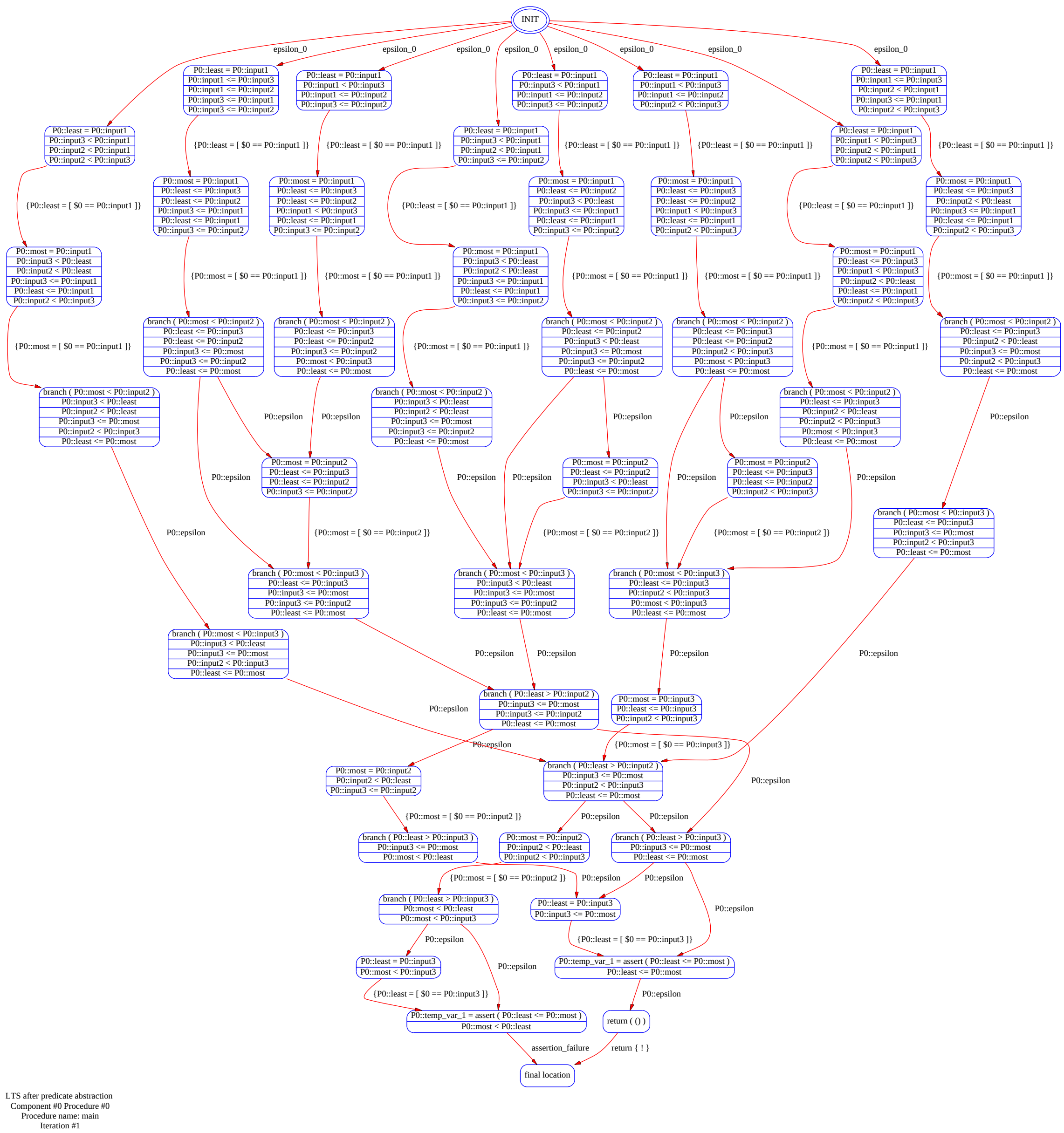


Figure 7.6: Abstract state space for minmax.c

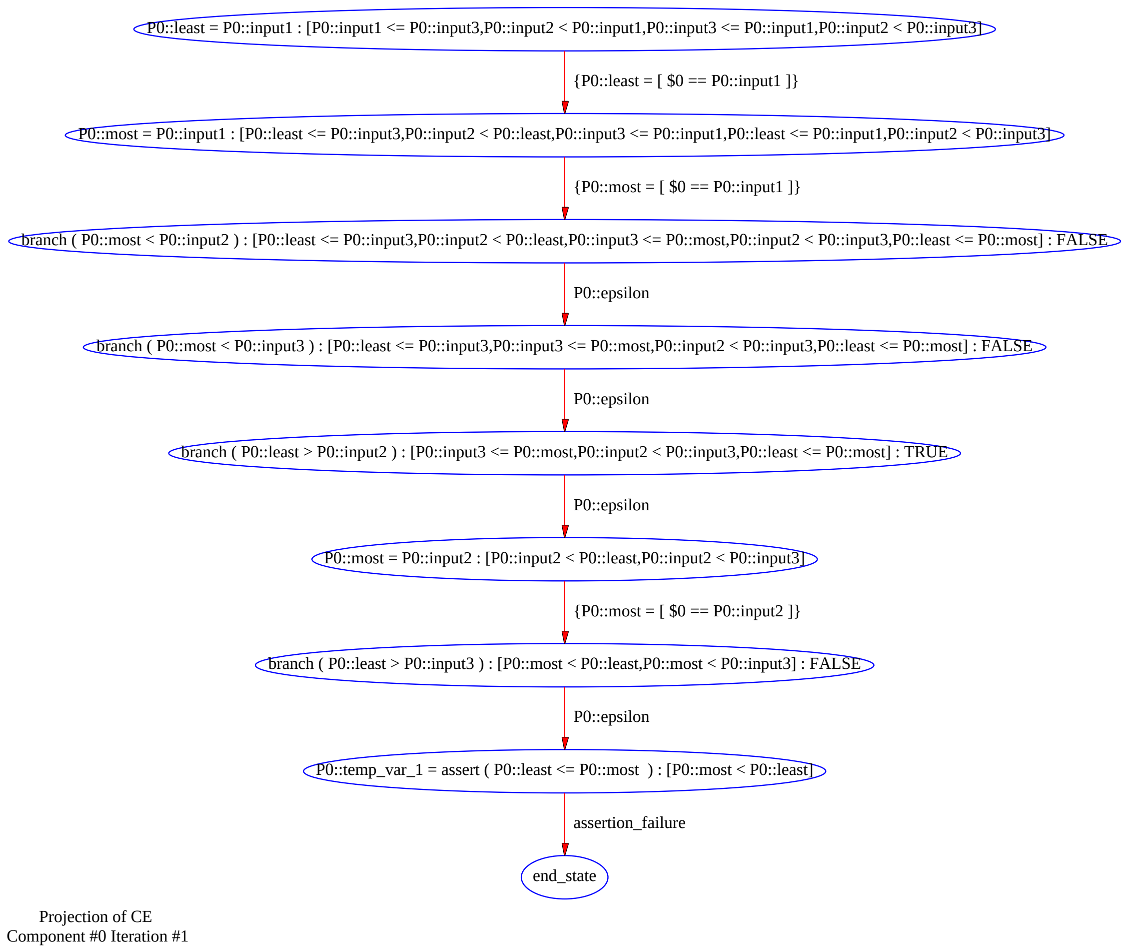


Figure 7.7: Abstract counterexample graph for minmax.c

```
State 0:  3:  least = input1
  [input3 < input1, input2 < input1, input2 < input3]
State 1:  4:  most = input1
  [input3 <= input1, input3 < least, input2 < input3,
   input2 < least, least <= input1]
State 2:  5:  branch ( most < input2 ) : FALSE
  [input3 <= most, input3 < least, input2 < input3,
   input2 < least, least <= most]
State 3:  7:  branch ( most < input3 ) : FALSE
  [input3 <= most, input3 < least, input2 < input3, least <= most]
State 4:  9:  branch ( least > input2 ) : TRUE
  [input3 <= most, input2 < input3, least <= most]
State 5:  10:  most = input2
  [input2 < input3, input2 < least]
State 6:  11:  branch ( least > input3 ) : FALSE
  [most < least, most < input3]
State 7:  13:  assert ( least <= most  )
  [most < least]
ASSERTION FAILURE
```

Figure 7.8: Abstract counterexample for minmax.c.

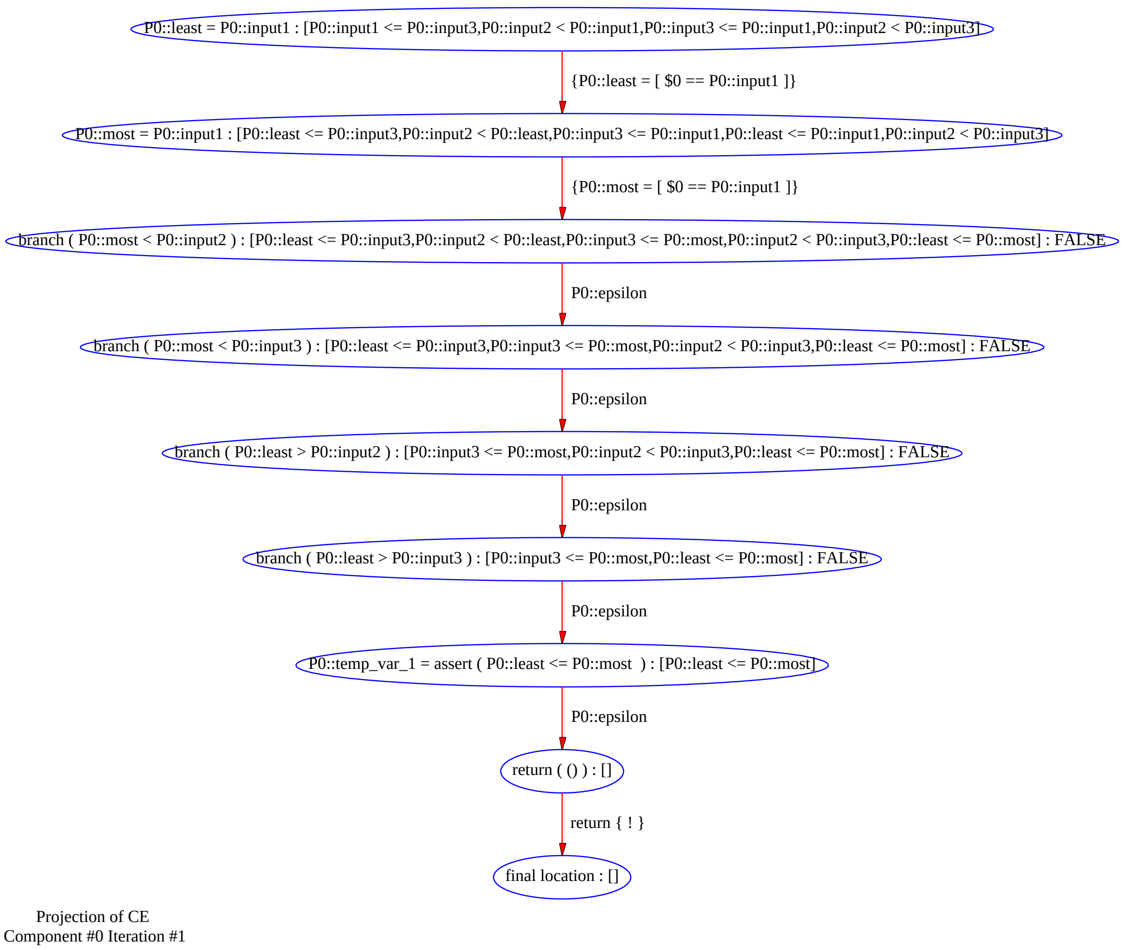


Figure 7.9: Abstract successful execution graph for minmax.c

```
State 0:  3:  least = input1
  [input3 < input1, input2 < input1, input2 < input3]

State 1:  4:  most = input1
  [input3 <= input1, input3 < least, input2 < input3,
   input2 < least, least <= input1]

State 2:  5:   branch ( most < input2 ) : FALSE
  [input3 <= most, input3 < least, input2 < input3,
   input2 < least, least <= most]

State 3:  7:  branch ( most < input3 ) : FALSE
  [input3 <= most, input3 < least, input2 < input3, least <= most]

State 4:  9:  branch ( least > input2 ) : FALSE
  [input3 <= most, input2 < input3, least <= most]

State 5:  11:  branch ( least > input3 ) : FALSE
  [input3 <= most, least <= most]

State 6:  13:  assert ( least <= most  )
  [least <= most]
```

Figure 7.10: Abstract successful execution for minmax.c

of a program. An over-approximation of system behaviors ensures that if a concrete counterexample exists, an abstract counterexample will also exist. However, the search algorithms used typically have no bias in favor of non-spurious counterexamples, and will often first discover a spurious counterexample if an abstraction is too coarse. The difficulty of finding the needle of a non-spurious error path in a haystack of unrealistic behaviors is why predicate abstraction is used in place of the less expensive analysis of control flow alone. That the likelihood of generating a non-spurious counterexample increases as the abstraction more closely captures the real behavior of the system is a primary motivation behind the CEGAR approach. The success of recent software model checking projects has shown that in many cases a "good" abstraction can be found in the territory between the control flow graph (CFG) of a program and its full concrete state-space. We propose that these "good" abstractions, which are provided "for free" by efficient verification tools, can also be used to improve program understanding and provide effective fault localization and debugging assistance. Results produced by an implementation of error explanation for the MAGIC tool [Chaki et al., 2004a] are presented in Chapter 9 as evidence of this claim.

7.2 Abstract Error Explanation

The explanation approach shown in Figure 7.1 can be used by a predicate abstraction and CEGAR based model checker, with the following three changes:

1. S, the formula representing executions of the program, is produced by unwind-

ing the transition relation of the *abstract* program $A(P)$ to a finite depth.

2. An outer loop must be added to the explanation process: solutions to the optimization problem, corresponding to abstract executions, may represent spurious behaviors, requiring multiple iterations to find a non-spurious most-similar successful execution.

3. The tool reports Δs to the user in terms of different control flow and *predicate values* rather than concrete variable values.

The first difference presents challenges when encoding the distance metric. Our previous explanation metrics relied on a static single assignment (SSA) [Alpern et al., 1988] encoding of execution. SSA provided a means to avoid the issue of *alignment*, i.e., which states of the successful execution should be compared to which states of the counterexample. In SSA, all executions are represented by a set of assignments to the same variables, and states are in a sense only implicit. SSA introduced a serious drawback, however: the distance metric was computed over values from *all possible control flow paths*. In some cases, a weak explanation was produced because the executions produced very similar values *in portions of the control flow not executed in either the successful execution or the counterexample.* The distance metric presented in Section 7.3 relies on alignments to avoid this counter-intuitive and questionable comparison over purely "hypothetical" values.

Another issue raised in unwinding the abstract transition relation is the choice of an unwinding depth. In the approach taken in Chapter 3, the original counterexample is produced by bounded model checking, and the bound used to discover a counterexample can (often) be reused in explanation. Furthermore, in CBMC this bound determined an upper limit for unrollings of loops, rather than a total number of steps. The depth used for abstract explanations limits the total number of steps in the successful execution. In practice, using a depth equal to the number of steps in the counterexample plus a small constant factor (to allow for previously untaken control branches) appears to suffice for most programs. As with SSA (see Section 3.1.3), while no upper bound can be given on the length of the closest successful execution, it is possible to guarantee, given that a successful execution of length j exists, a maximum upper bound within which a closer execution may be found (see Section 7.4).

When a spurious successful execution is generated, a blocking clause is added to the formula S to force generation of a different successful execution. The hypothesis is that in order to generate a non-spurious counterexample, the model checker will typically find a "good enough" abstraction to ensure that this process will converge rapidly. Experimental results support this conclusion in most cases. It is also possible to reuse the CEGAR abstraction refinement process at this stage in order to remove the spurious behavior, treating a spurious successful execution in the same manner as a spurious counterexample. However, this necessitates an expensive recomputation of the transition relation and counterexample.

Finally, and most importantly, the changes necessary to avoid (or induce) error are presented as Δs of predicates of variables, rather than as concrete values. The abstraction refinement process used to find a counterexample automatically, as a side-effect, produces a high-level model of the behavior of the program. With concrete explanations, the user must generalize to the logical causes of error from the overly specific values in the Δs. An abstract (but non-spurious) counterexample or successful execution, however, may represent *many* concrete behaviors of a program. The predicates necessary to find a non-spurious path will provide a description of the *logical* difference between these *sets* of concrete executions. As a simple example, a concrete Δ might indicate that in the counterexample `x` had the value of 47 and in the closest successful execution `x` had the value of 91. An abstract Δ, on the other hand, might state that in the counterexample, `x < y` and in the successful execution, `x >= y`. It is easy to see which explanation is more likely to capture the underlying essence of the erroneous behavior. The abstract Δ not only generalizes the constraint on `x`, but introduces the information that this constraint is relative to `y`. This claim relies on the assumption that in order to find a non-spurious counterexample, refinement will typically have to produce an abstraction that *effectively captures important aspects of a program's behavior.*

7.3 A Distance Metric for Abstract Executions

The distance metric used for explanations is dependent on the representation of program executions. For the MAGIC tool, an execution is an ordered sequence of state-action pairs: $\{(s_0, \alpha_0), (s_1, \alpha_1), \ldots (s_n, \alpha_n)\}$. We will refer to a state-action pair as a step. Each state, s, is composed of a control location $c(s)$ and a predicate valuation $p(s)$. Predicate valuations are vectors of values for the predicates associated with a particular control location. In the MAGIC abstraction framework, different predicates may be tracked at different control locations [Chaki et al., 2004b, 2003b]. For all states with the same control location, however, $p(s)$ will have the same size.

As an example, consider the control location at line 3 in Figure 7.3, `int least = input1`. In the abstraction used to generate the counterexample, for any state in which $c(s) = 3$ (using the line number to represent the unique control location), $|p(s)| = 3$. The components of $p(s)$ are values for distinct predicates. We write $p_i(s)$ to refer to the ith component of $p(s)$. $p_1(s)$ restricts the relationship of `input1` to `input3`. The possible values are: (`input1 < input3`), (`input3 < input1`), and (`input3 = input1`). The second and third components relate `input1` to `input2` and `input2` to `input3`.

Recall that the set of predicates associated with a given control location are determined by a combination of weakest precondition iteration [Namjoshi and Kurshan, 2000] plus an optimization-based counterexample guided abstraction refinement strategy [Chaki et al., 2004b, 2003b]. The exact choice of predicates at a given location may be difficult to understand intuitively; however, because MAGIC attempts to minimize (using pseudo-Boolean constraints) the set of predicates used to find a counterexample (or prove a property), we expect the predicate sets to always be relevant to "understanding" the program's behavior with respect to a property.

The distance metric d is defined with respect to two executions, a and b. We will assume that a is the counterexample, for the sake of convenience (the metric is symmetric). We will use a superscript notation (e.g. s_i^a) to distinguish states, actions, and control locations of a and b.

7.3.1 Alignment

The distance metric is based on a comparison of states and actions. An obvious approach would be to compare the ith step of a with the ith step of b. The two executions, however, may be of different lengths — if any changes in control flow are necessary to avoid the error, this will almost certainly be the case. In order to properly compare a and b, it is necessary to determine an *alignment* [Sankoff and Kruskal, 1983] mapping steps in a to steps in b. We will define alignment as a relation between elements of a and b, such that if $align(i, j)$, the ith step of a should be compared with the jth step of b:

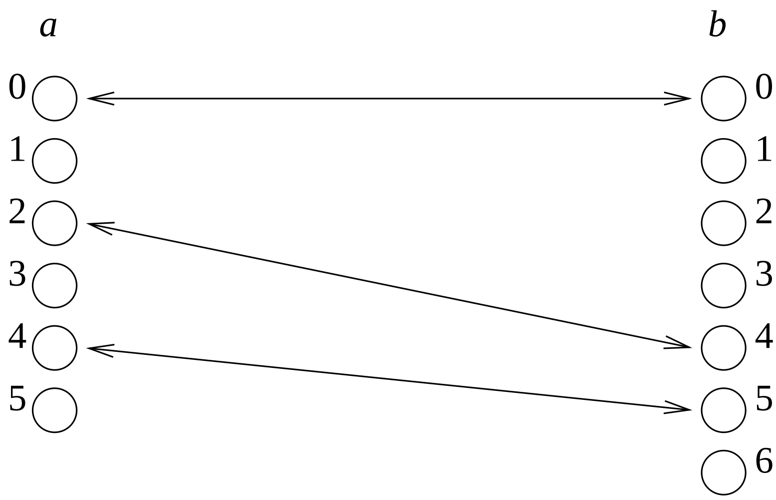
Definition 5 (alignment, $align(i, j)$)

$$align(i,j) = \begin{cases} 1 \texttt{ if } & c(s_i^a) = c(s_j^b) \\ & \wedge\ \forall k \neq j\ .\ align(i,k) = 0 \\ & \wedge\ \forall \ell \neq i\ .\ align(\ell,j) = 0 \\ & \wedge\ \forall m > i, n < j\ .\ align(m,n) = 0 \\ & \wedge\ \forall m < i, n > j\ .\ align(m,n) = 0 \\ 0 & \end{cases}$$

where $i, \ell, m < |a|$ *and* $j, k, n < |b|$.

The conditions for alignment require that:

- Steps can be aligned only if they have matching control locations.



Given the alignments shown, step 3 of a cannot be aligned with steps 0, 4, or 5 of b because alignments must be unique. Step 3 cannot be aligned with steps 0, 1, 2, 3, or 6 of b because alignments are not allowed to cross. Given these alignments, step 3 must remain unaligned.

Figure 7.11: Alignments for executions

- Alignments are *unique*: each step in a is aligned with at most one step in b and vice-versa.

- Alignments preserve ordering: e.g., if i is aligned with j, no earlier step in a may align with a later step in b, and no later step in a may align with an earlier step in b. Visually, this means that alignments cannot cross.

See Figure 7.11 for an example of the consequences of these constraints. For a given a and b, there may be multiple alignments consistent with these conditions. In the presence of loops, there may be several steps in a or b (or both) with the same control location. There may also be steps in a or b that are not aligned with any step in the other execution. These steps are *unaligned*:

Definition 6 (unaligned, $unalign_{a/b}(i/j)$**)**

$$unalign_a(i) = \begin{cases} 1 \texttt{ if } \ \forall j \ . \neg \, align(i, j) \\ 0 \texttt{ otherwise} \end{cases}$$

$$unalign_b(j) = \begin{cases} 1 \texttt{ if } \ \forall i \ . \neg \, align(i, j) \\ 0 \texttt{ otherwise} \end{cases}$$

where $i < |a|$ *and* $j < |b|$.

A step may be impossible to align — either because no control location in the other execution is matching, or, as in Figure 7.11, because certain other alignments preclude the conditions from holding. It is important to note, however, that the first condition requires only that *if two steps are aligned*, the conditions must hold. There is no requirement that *if the conditions hold*, two steps must be aligned. The empty relation is always a valid alignment. We define the conditions under which steps *may* be aligned, rather than *must* be aligned.

The distance metric is defined so as to pick the best (i.e., distance-minimizing) alignment. Defining the "true" distance as the minimal distance over the choice of all possible alignments is a standard method in string and sequence comparison [Sankoff and Kruskal, 1983].

7.3.2 The Distance Metric d

Given a and b we define the distance $d(a, b)$ based on the number of atomic changes (Δs) needed to transform a into b. The first component of the metric (i.e., subset of the total Δs) is possible alterations to predicate values. Δp is defined over *steps* i and j of a and b as well as over whole executions. The sum of individual step differences is used to define the total Δp between two complete executions:

Definition 7 ($\Delta p(i, j, v), \Delta p(a, b)$)

$$\Delta p(i, j, v) = \begin{cases} 1 \texttt{ if } & align(i, j) \wedge p_v(s_i^a) \neq p_v(s_j^b) \\ 0 \texttt{ otherwise} & \end{cases}$$

where $i < |a|$*,* $j < |b|$*, and* $v < |p(s_i^a)|$*.*

$$\Delta p(a, b) = \sum_{i=0}^{|a|-1} \sum_{j=0}^{|b|-1} \sum_{v=0}^{|p(s_i^a)|-1} \Delta p(i, j, v)$$

$\Delta p(i, j, v)$ is 1 iff step i and step j are aligned and have differing predicate values for the vth component of their predicate valuations. This comparison is always valid if i and j are aligned since this requires that they share a control location. In MAGIC requiring alignment before comparing predicates is particularly crucial, as predicates from different control locations may not be comparable.

Changes in actions are defined in a similar alignment-based manner:

Definition 8 ($\Delta\alpha(i, j), \Delta\alpha(a, b)$)

$$\Delta\alpha(i, j) = \begin{cases} 1 \texttt{ if } & align(i, j) \wedge \alpha_i^a \neq \alpha_j^b \\ 0 \texttt{ otherwise} & \end{cases}$$

where $i < |a|$*, and* $j < |b|$*.*

$$\Delta\alpha(a, b) = \sum_{i=0}^{|a|-1} \sum_{j=0}^{|b|-1} \Delta\alpha(i, j)$$

These Δs account for all possible differences in aligned states. In order to describe control flow differences between a and b, the metric d must also take into account the unaligned states of the executions:

Definition 9 ($\Delta c(a, b)$)

$$\Delta c(a, b) = \sum_{i=0}^{|a|-1} unalign_a(i) + \sum_{j=0}^{|b|-1} unalign_b(j)$$

The distance metric is then defined as the minimal weighted sum of predicate, action, and control Δs, over all possible alignments:

Definition 10 (distance, $d(a, b)$)

$$d(a, b) = min_{align}(W_p \cdot \Delta p(a, b) + W_\alpha \cdot \Delta\alpha(a, b) + W_c \cdot \Delta c(a, b))$$

W_p, W_α, and W_c may reasonably vary, depending on the user's interest in similarity of observable actions, predicate values, and control locations. However, in accordance with the principle that it is best to compare steps whenever possible, it is suggested that W_c be chosen such that it is greater than the maximum possible Δp + $\Delta\alpha$ for a single step. In our experimental results, we have uniformly used $W_p = 1$, $W_\alpha = 1$, and $W_c = max(|p(s^a)|) + 2$. With positive values for these weights, d satisfies the standard *nonnegative*, *zero*, *symmetry*, and *triangle inequality* properties of a distance metric [Sankoff and Kruskal, 1983]. The use of an alignment-based metric, such as d, is not intrinsically tied to abstraction. For any notion of executions based on explicit steps and states, Δp (really Δs: changes to state components other than control location) could be defined over the appropriate elements.

7.4 Finding a Successful Execution

The procedure for finding a successful execution b that is as similar as possible to a counterexample a is as follows:

1. Unwind the transition relation of $A(P)$ to a finite depth to produce a propositional constraint S. Solutions of S will represent executions of $A(P)$ that do not violate *Spec*. Any solution of S represents a potential b to be compared against the counterexample.

2. For a fixed counterexample a, add to S Boolean variables for all possible alignments of a and b. For each $i < |a|$ and $j < max(|b|)$[8], a variable for $align(i, j)$ may be introduced. Rather than adding all $|a| \times max(|b|)$ variables, we observe that $align(i, j)$ can be 1 only if it is possible for $c(s_i^a)$ to equal $c(s_j^b)$. In many cases, the unwinding of the transition relation will show that this condition cannot be satisfied. The constraints given in Definition 5 are also introduced only for alignments not ruled out by the transition relation: e.g., if unwinding shows that states 3 and 10 of a and b cannot have matching control locations under any possible execution sequence, there is no need to introduce an alignment variable for $align(3, 10)$.

3. Add Δ-*variables* for each possible difference between a and b. For each $i < |a|$, $j < max(|b|)$, and $v < |p(s_i^a)|$, a variable is introduced for: $unalign_a(i)$, $unalign_b(j)$, $\Delta p(i, j, v)$, and $\Delta\alpha(i, j)$. The values are constrained in accordance with the definitions in Section 7.3. When $j \geq |b|$ (because the successful execution is shorter than the unwinding depth), the associated $align$ variable is forced to be 0, ensuring that Δs variables for steps not in b are also 0.

4. Assign weights to the Δ-variables to produce a 0-1 ILP problem. Variables representing $unalign_a(i)$ and $unalign_b(j)$ are given a weight equal to W_c. $\Delta p(i, j, v)$ and $\Delta\alpha(i, j)$ are weighted according to W_p and W_α, respectively. The optimization problem is to minimize the weighted sum over all Δ variables and alignments. This weighted sum is equal to $d(a, b)$.

[8] $max(|b|)$ is the unwinding depth.

5. Use a 0-1 ILP solver to produce an alignment and b that minimize $d(a, b)$. One of two conditions may make it impossible to find such a b:

   (a) All executions of the program P violate the specification $Spec$.

   (b) All abstract executions in $A(P)$ that represent at least one successful execution also represent at least one counterexample. Because success is measured in the abstract state-space, no b that does not represent a counterexample can be found.

   Using a shallower unwinding depth may make it possible to find b even if one of these conditions holds. An execution that, if extended, will always become a counterexample will be considered successful if it does not reach an error state before $max(|b|)$ steps. Clearly, this is sub-optimal. As noted in Chapter 1, the distance metric based technique is unsuitable for programs that always fail.

   It is also possible to know, if we have discovered a successful execution b, whether a closer execution can potentially be found using a larger unwinding depth. If the counterexample a is of length i and the successful execution b is of length $j > i$, it is easily seen that there must be at least $j - i$ unaligned steps. Therefore, $\Delta c(a, b)$ must be at least $j - i$. The minimum distance possible between a and b is thus $W_c(j - i)$. If a closest successful execution b is found at a distance d, there is no need to consider the possibility that an execution of length j, such that $W_c(j - i) \geq d$, can be closer to a.

6. Check that the execution b is not spurious. If b is spurious, add a blocking clause to S forcing a different choice of b and re-solve the ILP problem.

7. Present b to the user. Use the Δ-variable values to present to the user the changes that must be made to a in order to avoid the error (and produce b).

In our implementation, we use the pseudo-Boolean solver PBS [Aloul et al., 2002], which combines a fast SAT procedure with special techniques for 0-1 ILP, to efficiently perform step 5. Returning to the motivating example (Figure 7.3), we observe that it requires 421 Boolean variables to represent the transition relation for $A(P)$ up to a depth of 12 steps (this is sufficient to encode all possible executions of the program). An additional 110 variables are required to represent the possible alignments and Δs. The full SAT instance has 531 variables and 1,841 clauses. The CBMC representation, even without the overhead of alignment variables, has 1,759 variables and 5,747 clauses.

The output shown in Figure 7.5 is produced by examining the values of the Δ and alignment values in the solution to the ILP problem. In this case, there is one unaligned control location in a, at line 10 (the location of the error). The (aligned) control locations in a and b that follow this change in control flow (lines 11 and 13) differ in predicate values, because the assignment of `input2` to `most` has been performed in a but not in b. The counterexample's final action is an `assertion_failure`, while in b the assertion holds.

Chapter 8

Explaining LTL Property Failures

HURRY UP PLEASE IT'S TIME

- T. S. Eliot, "The Waste Land"

8.1 Successful Executions for LTL Properties

The above explanation procedure and distance metric can be applied without modification to explain counterexamples to Linear Temporal Logic (LTL) formulas. The BMC unwinding of the abstract transition relation, however, must be modified to take into account a different notion of a successful execution. The implementation of LTL property explanation described is for abstract executions, but the approach presented in this section will work for concrete executions equally well.

For reachability properties, successful execution is guaranteed by adding constraints such that no error state can appear in b. For an LTL property $A\phi$, a successful execution is a *counterexample* to $A\neg\phi$. A counterexample to $A\phi$ demonstrates that ϕ does not hold for all paths. A counterexample to $A\neg\phi$ demonstrates that ϕ *can hold* for some path. This is not guaranteed to be true — no such path my exist, just as a program may have no successful executions at all, when considering safety properties.

LTL model checking in MAGIC uses the standard approach in which a Büchi automaton for the negation of the property is constructed [Gerth et al., 1995]. Counterexamples are executions in the product automaton (the product of the model and the Büchi automaton for the negation of the property) that contain a cycle that passes through an accepting state. Accepting states in the product automaton are projected from the Büchi automaton. In order to check for successful executions, MAGIC unwinds a Büchi automaton for the property (rather than its negation) along with the transition relation and adds constraints requiring a cycle through an accepting condition to appear in the execution.

The use of a Büchi condition adds an additional possibility to the list of reasons given in step 5 of the procedure in Section 7.4 for inability to find a successful execution b: the unwinding depth may be insufficient to allow a cycle through an accepting state. For reachability properties, a "spurious"[1] successful execution can sometimes be produced by lowering the unwinding depth. For LTL properties, the unwinding depth may need to be *increased* in order to find a successful execution, but any b that is discovered will represent an infinite behavior, and thus be immune

[1]Here, spurious is used in the sense that the execution will eventually violate the property, rather than in the sense of abstraction-introduced behavior.

to extension to error — further steps cannot take a path with a cycle into an error state, as the entire future of the path is represented by the stem and cycle[2].

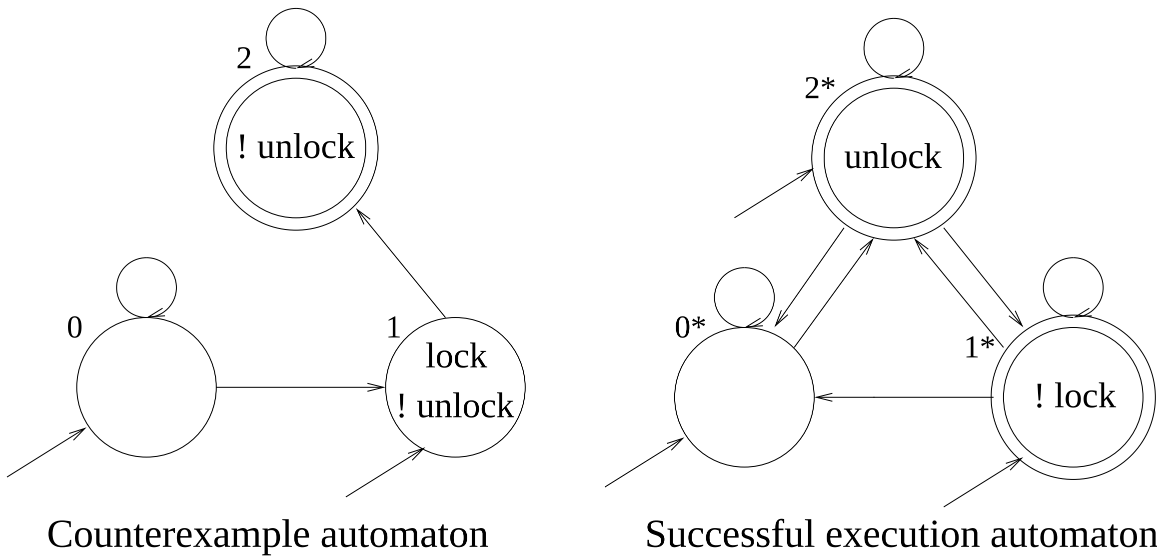
Because determining the unwinding depth sufficient to allow for a cycle is difficult, MAGIC will automatically increase the unwinding depth (up to a given maximum) when it fails to find a solution for b in an LTL explanation.

The underlying technique is quite similar, these changes aside, to the basic method for abstract explanation, as shown in the example that follows: we find an abstract execution that is successful and maximally similar to a counterexample, and present differences between these to the user. The change in the definition of success and the infinite length of counterexample and successful execution are sufficient to handle LTL properties without further novel additions to the explanation approach.

8.2 Example of LTL Explanation

As an example of the notion of successful execution used when explaining LTL counterexamples, consider the property $(A)G(lock \Rightarrow F(unlock))$, which requires that on all paths, at all steps, locking requires eventually unlocking. The Büchi automaton on the left in Figure 8.1 accepts counterexamples to this property: executions in which a lock is acquired (state 1) but not released (state 2). Because state 2 is the only accepting state, counterexamples must have a cycle through program states that do not unlock, and can only reach this state after having locked at least once

[2]Though b may, of course, be a spurious behavior introduced by abstraction.



Büchi automata for $(A)G(lock \Rightarrow F(unlock))$, negated and un-negated. Each state is labeled with a set of constraints: e.g., state 1 requires both lock and not unlock.

Figure 8.1: Successful executions for LTL properties

without unlocking (because state 2 is not an initial state of the automaton).

The automaton on the right accepts *successful executions*: executions which either never lock or never lock after having unlocked (cycles on state 1*), or which unlock infinitely often (cycles including state 2*). In this case minimizing the distance to the counterexample increases the chance of finding a successful execution that locks, as the counterexample will be forced to lock at least once.

The code in Figure 8.2 produces a non-spurious counterexample (Figure 8.3) when checked against the LTL formula $(A)G(lock \Rightarrow F(unlock))$. It is possible to exit the body of `process` without making a call to `Unlock`.

The explanation in Figure 8.4, based on the successful execution in Figure 8.5, describes the conditions under which the error appears: in the counterexample, `y` is `<` 1 but *not* equal to 0. In the successful execution, `y > 0`. The error is avoided because the assignment of `y` to `z` on line 9 now ensures that `z` will satisfy the condition at line 15, creating a cycle in which `process` unlocks. The change in `y` has focused our attention on the real problem with this code: the programmer has neglected to take negative values of `y` into account, assuming that `y != 0` implies `y > 0`.

For this example, MAGIC requires 247 milliseconds to unwind the transition relation and 952 milliseconds to produce an explanation. The pseudo-Boolean constraints are over 2,825 variables and 20,246 clauses. The total time taken to produce the abstraction (and the non-spurious counterexample) is 1,586 milliseconds, mostly spent in four iterations of abstraction-refinement, in order to produce the 4 predicates used in the abstraction (`y == 0`, `y > 0`, `x == 0`, and `z > 0`). The first explanation produced is non-spurious; neither depth increase iterations or blocking clauses are needed in this case (though for even more unrealistic toy LTL examples we did observe a need for both blocking clauses and depth increase iterations).

```
int process () {
  int x, y, z;
  z = 0;
  Lock ();
  if (x == 0)
    if (y == 0)
      z = 1;
  if (y != 0) {
    z = y;
  }
  if (x != 0) {
    z = 2;
    Unlock ();
  }
  else if (z > 0) {
    z = 3;
    Unlock ();
  }
}
```

Figure 8.2: locks.c

```
int main () {
  while (1)
    process ();
}
```

Figure 8.2 (continued)

```
 21:  branch ( 1 ) : [process::x == 0,process::y != 0,pro-
cess::y < 1] : TRUE
############ epsilon ############
 3:  process::z = 0 : [process::x == 0,process::y != 0,pro-
cess::y < 1]
############ {process::z = [ $0 == 0 ]} ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1]
############ epsilon ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1]
############ lock ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1]
############ epsilon ############
 5:  branch ( process::x == 0 ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1] : TRUE
############ epsilon ############
```

Figure 8.3: Counterexample for locks.c

```
 6:  branch ( process::y == 0 ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1] : FALSE
############ epsilon ############
 8:  branch ( process::y != 0 ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1] : TRUE
############ epsilon ############
 9:  process::z = process::y : [process::x == 0,pro-
cess::y != 0,process::y < 1]
############ {process::z = [ $0 == process::y ]} ############
 11:  branch ( process::x != 0 ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1] : FALSE
############ epsilon ############
 15:  branch ( process::z > 0 ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1] : FALSE
############ epsilon ############
 21:  branch ( 1 ) : [process::x == 0,process::y != 0,pro-
cess::y < 1] : TRUE
############ epsilon ############
```

Figure 8.3 (continued)

```
 3:  process::z = 0 : [process::x == 0,process::y != 0,pro-
cess::y < 1]
############ {process::z = [ $0 == 0 ]} ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1]
############ epsilon ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1]
############ lock ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y != 0,process::y < 1]
############ epsilon ############
```

Figure 8.3 (continued)

```
Predicate changed (steps #0-18):
  was:  process::y != 0 , process::y < 1
  now:  process::y > 0
------------------------
Predicate changed (steps #9-10):
  was:  process::z < 1
  now:  process::z > 0
------------------------
Control location inserted (step #11):
  16:  process::z = 3
  {process::z = [ $0 == 3 ]}
------------------------
Control location inserted (step #12):
  17:  process::temp_var_6 = Unlock (  )   [epsilon]
------------------------
Control location inserted (step #13):
  process::temp_var_6 = Unlock (  )        [unlock]
------------------------
Control location inserted (step #14):
  17:  process::temp_var_6 = Unlock (  )   [epsilon]
```

Figure 8.4: Abstract Δ values for locks.c

```
 21:  branch ( 1 ) : [process::x == 0,process::y > 0] : TRUE
############ epsilon ############
 3:  process::z = 0 : [process::x == 0,process::y > 0]
############ {process::z = [ $0 == 0 ]} ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y > 0]
############ epsilon ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y > 0]
############ lock ############
 4:  process::temp_var_4 = Lock (  ) : [process::z < 1,pro-
cess::x == 0,process::y > 0]
############ epsilon ############
 5:  branch ( process::x == 0 ) : [process::z < 1,pro-
cess::x == 0,process::y > 0] : TRUE
############ epsilon ############
 6:  branch ( process::y == 0 ) : [process::z < 1,pro-
cess::x == 0,process::y > 0] : FALSE
############ epsilon ############
```

Figure 8.5: Successful execution for locks.c

```
 8:  branch ( process::y != 0 ) : [process::z < 1,pro-
cess::x == 0,process::y > 0] : TRUE
############ epsilon ############
 9:  process::z = process::y : [process::x == 0,process::y > 0]
############ {process::z = [ $0 == process::y ]} ############
 11:  branch ( process::x != 0 ) : [process::x == 0,pro-
cess::y > 0,process::z > 0] : FALSE
############ epsilon ############
 15:  branch ( process::z > 0 ) : [process::x == 0,pro-
cess::y > 0,process::z > 0] : TRUE
############ epsilon ############
 16:  process::z = 3 : [process::x == 0,process::y > 0]
############ {process::z = [ $0 == 3 ]} ############
 17:  process::temp_var_6 = Unlock (  ) : [process::x == 0,pro-
cess::y > 0]
############ epsilon ############
 17:  process::temp_var_6 = Unlock (  ) : [process::x == 0,pro-
cess::y > 0]
############ unlock ############
```

Figure 8.5 (continued)

```
 17:  process::temp_var_6 = Unlock (  ) : [process::x == 0,pro-
cess::y > 0]
############ epsilon ############
 21:  branch ( 1 ) : [process::x == 0,process::y > 0]
############ epsilon ############
```

Figure 8.5 (continued)

Chapter 9

Case Studies and Evaluation for Abstract Explanation

Go in fear of abstraction.

- Ezra Pound

9.1 Experimental Results

We applied the MAGIC implementation of error explanation to several faulty programs. Table 9.1 summarizes the results. The results strongly support the claim that finding a non-spurious successful execution will require very few iterations. No benchmark required the addition of even one blocking clause to prevent a spurious successful execution, even though in two cases considerable refinement was required to produce a non-spurious counterexample. Results not shown in the table also required no blocking clauses. Such iterations were observed for a few artificial (and unrealistic) examples with LTL properties: it seems reasonable to expect that spurious cycles are somewhat harder to eliminate than spurious non-cyclic executions.

9.1.1 Benchmarks

The examples presented in Table 9.1 are taken from several sources. The smaller benchmarks were taken from the regression tests for MAGIC (small fragments of Linux kernel code with seeded errors). Additional benchmarks were taken from the C source code of `OpenSSL-0.9.6c`, with seeded errors. In particular, SSL-1 and SSL-2 are from faulty versions of the initial handshake protocol. Section 9.4 presents the SSL-1 explanation (Figure 9.6) in greater detail as a case study. The final benchmark is the source code for the μC/OS-II [μC/OS-II Website] real-time multitasking kernel (RTOS) for microprocessors and microcontrollers. The error explained was original to the source code, rather than added for our experiments.

The first Δ can appear after the "first" fault in some cases (even for a good explanation) because the order of control flow does not strictly follow line ordering (these programs contain function calls). The information is included in order to show the large number of cases in which the program fault is located at the point of the the first Δ.

Program	LOC	T(Unwind)	T(Search)	PredIt	Preds	ExplIt	1st Δ	Fault	Score	CE
mutex-n-01.c (lock)	343	0.015	0.027	1	1	1	250	251*	0.785	6
mutex-n-01.c (unlock)	343	0.017	0.027	1	0	1	285	251*	0.993	6
pci-n-01.c	60	0.006	0.062	2	1	1	39	58	0.782	9
pci-rec-n-01.c	64	0.009	0.076	1	0	1	45	32*	0.720	8
SSL-1	2487	0.947	7.118	72	5	1	1213	1213	0.999	29
SSL-2	2487	0.369	3.084	16	5	1	1223	1223	0.999	52
μC/OS-II 2.00	2981	0.109	0.653	1	0	1	1936	1924	0.000	19

Program is the program with an error to be explained. Where a single program was used with multiple specifications, the $Spec$ is also given. **LOC** is the # of lines of code for each example. **T(Unwind)** is total explanation unwinding time, **T(Search)** is total explanation search time (all times in seconds). **PredIt** is the number of iterations required to discover a non-spurious counterexample and generate the final $A(P)$. **Preds** gives the final number of predicates needed to disprove the property. **ExplIt** is the number of iterations required to find a non-spurious successful execution. **1st Δ** is the line# of the first (in source code execution ordering) Δ reported. **Fault** is the line# of the first fault in the program, with a * beside cases with multiple faults. **Score** is the score for the full set of Δs, by Renieris and Reiss' evaluation. **CE** is the number of steps in the counterexample.

Table 9.1: Experimental results for MAGIC examples

9.2 Evaluation of Fault Localization

The results presented in this section make use of the same scoring function as those presented in chapter 5. The numeric results are again quite satisfactory. Unfortunately, for the MAGIC examples, an alternative predicate-abstraction based model checker was not available for comparison, and many of the examples either cannot be checked by CBMC or make use of specifications that are not easily encoded as CBMC-style assertions. Comparison to the SLAM [Ball et al., 2003] error explanation facilities would be suitable, but in the current implementation of SLAM that feature is not supported.

9.3 Benchmark: Mutex Explanation

Figures 9.1 and 9.2 show faulty and correct fragment of the code for the `mutex-n-01.c` benchmark. The fault is at line 251: the incorrect code fails to make a required call to `__pthread_lock`.

Figure 9.3 shows the explanation produced by MAGIC for a counterexample to the locking property (presented in MAGIC's FSP-like [Magee and Kramer, 1999] notation in Figure 9.4) for the faulty code. The explanation shows that if the branch at line 250 is taken and the program returns (and terminates), the locking property is not violated. The branch condition depends on an equality that is not covered by the abstraction, so this branch is a non-deterministic choice, and the successful execution can diverge from the counterexample without any predicate value changes.

```
return (0);
case 2:
self = thread_self();
if ((unsigned long)mutex->__m_owner == (unsigned long) self)
{return (35);
}
mutex->__m_owner = self;
return (0);
```

Figure 9.1: Incorrect version of mutex code fragment

```
return (0);
case 2:
self = thread_self();
if ((unsigned long)mutex->__m_owner == (unsigned long) self)
{return (35);
} __pthread_lock(& mutex->__m_lock, self);
mutex->__m_owner = self;
return (0);
```

Figure 9.2: Correct version of mutex code fragment

```
Control location inserted (step #4):

  250:  return ( 35 )

  return  $0 == 35

------------------------

Control location inserted (step #5):

  final location

------------------------

Control location deleted (step #6.4):

  252:  mutex -> __m_owner = self

  mutex -> __m_owner = [ $0 == self ]

------------------------

Control location deleted (step #6.5):

  253:  return ( 0 )

------------------------
```

Figure 9.3: Abstract Δ values for mutex lock failure

```
cprog  pthread_mutex_lock = __pthread_mutex_lock {
    abstract mutex_lock_01,{$1->__m_count >= 0},PthreadLock;
}
cproc __pthread_lock {
    abstract {__pthread_lock_abs,1,LockSpec};
}
cproc __pthread_mutex_lock {
    abstract {mutex_lock_01,1,PthreadLock};
}
LockSpec = ( lock -> return {} -> STOP ).
PthreadLock = ( lock -> Locked | {$1->__m_count = [$0 == $1-
>__m_count + 1]} -> Locked | return {$0 == 22} -> STOP | re-
turn {$0 == 35} -> STOP ),
Locked = ( return $0 == 0 -> STOP | {$1->__m_count = [$0 == 0]} -
> return {$0 == 0} -> STOP ).
```

Figure 9.4: Mutex locking property

```
s->shutdown = 0;
ret = ssl3_get_client_hello(s);
if (ret <= 0) {
  ret = ssl3_get_client_hello(s);
  goto end;
}
got_new_session = 1;
s->state = 8496;
s->init_num = 0;
```

Figure 9.5: SSL-1 code fragment

Both executions are non-spurious. The explanation correctly identifies the control flow necessary to avoid the error.

In the program dependency graph, of course, the fault on line 251 is immediately dependent on this branch, resulting in a relatively good localization score (the presence of irrelevant lines 252 and 253 in the explanation prevent the explanation from being scored more highly).

9.4 SSL Explanation

The specification for the SSL handshake protocol requires that if the `get_client_hello` action is performed, then a `send_server_hello` must be performed, or the server call

```
Action changed (step #25):
  was:  get_client_hello
  now:  {ret = [ $0 == -1 ]}
------------------------
Control location deleted (step #26):
  1213:  ret = ssl3_get_client_hello ( s  )  [{ret = [ $0 == 1 ]}]
------------------------
Predicate changed (step #26):
  was:  ret == 1
  now:  ret == -1
------------------------
Predicate changed (step #27):
  was:  ret == 1
  now:  ret == -1
Action changed (step #27):
  was:  return { $0 == 1 }
  now:  return { $0 == -1 }
------------------------
Control location inserted (step #28):
  final location
```

Figure 9.6: Abstract Δ values for SSL-1

```
 1124:  Time = tmp
############ {Time = [ $0 == tmp ]} ############
 1125:  cb = 0
############ {cb = [ $0 == 0 ]} ############
 1126:  ret = -1
############ {ret = [ $0 == -1 ]} ############
 1127:  skip = 0
############ {skip = [ $0 == 0 ]} ############
 1128:  got_new_session = 0
############ {got_new_session = [ $0 == 0 ]} ############
 1129:  * tmp___0 = 0
############ {* tmp___0 = [ $0 == 0 ]} ############
 1132:  branch ( s -> info_callback != 0 ) : TRUE
############ epsilon ############
 1134:  cb = s -> info_callback
############ {cb = [ $0 == s -> info_callback ]} ############
 1136:  s -> in_handshake = s -> in_handshake + 1
############ {s -> in_handshake = [ $0 == s -> in_handshake + 1 ]}
############
```

Figure 9.7: Counterexample for SSL-1

```
 1148:  branch ( 1 ) : TRUE
############ epsilon ############
 1149:  state = s -> state
############ {state = [ $0 == s -> state ]} ############
 1151:  branch ( s -> state == 12292 ) : FALSE
############ epsilon ############
 1153:  branch ( s -> state == 16384 ) : FALSE
############ epsilon ############
 1154:  branch ( s -> state == 8192 ) : FALSE
############ epsilon ############
 1155:  branch ( s -> state == 24576 ) : FALSE
############ epsilon ############
 1156:  branch ( s -> state == 8195 ) : FALSE
############ epsilon ############
 1192:  branch ( s -> state == 8480 ) : FALSE
############ epsilon ############
 1193:  branch ( s -> state == 8481 ) : FALSE
############ epsilon ############
 1204:  branch ( s -> state == 8482 ) : FALSE
############ epsilon ############
```

Figure 9.7 (continued)

```
 1207:  branch ( s -> state == 8464 ) : TRUE
############ epsilon ############
 1210:  s -> shutdown = 0
############ {s -> shutdown = [ $0 == 0 ]} ############
 1211:  ret = ssl3_get_client_hello ( s  )
############ epsilon ############
 1211:  ret = ssl3_get_client_hello ( s  )
############ {ret = [ $0 == -1 ]} ############
 1212:  branch ( ret < 1 ) : TRUE
############ epsilon ############
 1213:  ret = ssl3_get_client_hello ( s  )
############ epsilon ############
 1213:  ret = ssl3_get_client_hello ( s  )
############ get_client_hello ############
 1213:  ret = ssl3_get_client_hello ( s  )
############ {ret = [ $0 == 1 ]} ############
 1461:  s -> in_handshake = s -> in_handshake - 1
############ {s -> in_handshake = [ $0 == s -> in_handshake -
1 ]} ############
 1464:  return ( ret )
############ return { $0 == 1 } ############
```

Figure 9.7 (continued)

must return a value of -1. The fault introduced at line 1213 (Figure 9.5) allows a re-assignment of the return value `ret` (and presents another opportunity for a successful client hello action). In the correct code, the assignment at line 1213 is not present. The counterexample for this property (Figure 9.7) contains 29 states and actions that a user must sort through in order to understand the error. Error explanation produces a successful execution that differs in two actions, two predicates, and two control locations (Δs in Figure 9.6). The key to the error is indicated as being the faulty assignment at line 1213: if this call fails as the first call did (causing the branch at line 1212 to be taken), the specification is not violated. In the counterexample, the server call succeeds, having failed the first time, and the server returns success without having responded to the received client hello. In the successful execution, the second attempt to get a client hello also fails, and the value of `ret` correctly indicates failure. The error has been localized to line 1213, and the precise conditions under which the faulty assignment will result in erroneous behavior are indicated.

9.5 Comparing Concrete and Abstract Explanation

9.5.1 Is Abstract Superior to Concrete?

Predicate abstraction tools such as SLAM, BLAST, and MAGIC are popular because abstraction is a powerful tool for dealing with the state-space explosion problem. It is at the least probable that predicate abstraction will typically scale better than bounded model checking of concrete state-spaces. Abstract explanation improves the expressiveness of explanations, allowing Δs over predicates of values: with concrete explanation, the change `x == y` vs. `x > y` is simply not expressible. A concrete Δ, in fact, will only refer to the value of either `x` or `y`, but not both, hiding the essential point that the relationship between these values is important.

It might appear that as the number of predicates grows, abstract explanations would become increasingly difficult to read. However, only the predicates that must *change* in order to avoid error will appear in an explanation. In general, it is reasonable to expect that even with a large number of predicates, the number of predicate changes would be roughly equivalent to the number of concrete value changes. In the case that more predicate changes are present, important variable relationships would be missing from the concrete explanation. In the case studies presented here, very few predicates are included in the abstraction (precluding the possibility of overwhelming numbers of changes): the Linux kernel fragments used only *one* predicate, and the SSL examples showed changes in only one out of five total predicates in the abstract model. Because MAGIC attempts to minimize the number of predicates in the model, it is reasonable to expect that few, if any, irrelevant predicates will be included in the model, and that subsumption will automatically handle cases where both `y < 0` and `y < 5` change, for example, unless these are independently important.

Another reasonable expectation is that abstraction, by creating a smaller state space and thus, typically, fewer possible program executions, will help to avoid the problem shown in the minmax.c example: multiple nearest successful executions at the same distance, only some of which provide a good explanation. The distance in the abstract case is based, we hope, on a smaller number of possible changes. In no cases did MAGIC demonstrate the sensitivity to choice of model checking methods that `explain` demonstrated; on the other hand, `explain` only shows this difficulty on minmax.c. It is plausible that, while this issue might favor abstract explanation, it typically arises only in the case of small, toy programs where the range of distances is very small. Unless more examples in which the issue arises for `explain` are discovered, this is a potentially minor advantage of abstract explanation.

These arguments present a tempting case for the claim that abstract explanation is simply better than concrete explanation, at least when a program can be successfully abstracted. For some programs, of course, bounded model checking is more effective than predicate abstraction: when a short counterexample exists and data structures, pointer usage, or reliance on precise modeling of finite language semantics (e.g. arithmetic overflow) make abstraction difficult, concrete model checking can be a very effective alternative. This reflects differences in model checking techniques rather than explanation techniques *per se*.

9.5.2 Is Concrete Superior to Abstract?

The result for μC/OS-II in Table 9.1 is startling: the explanation is of no value for localization! Inspection shows that the unwinding depth allows the system to avoid the consequences of a missing return statement by delaying the calls that expose the error. The counterexample fails almost immediately after taking the branch guarding the location of the missing return. Because the counterexample fails immediately after error, any successful execution will be forced to insert new control locations. The distance metric, unfortunately, ensures that it is "better" to introduce irrelevant steps that delay the unlock call that exposes the error than to avoid the branch that ensures failure (which requires that even more new control locations be added and forces a costly unalignment after the branch). CBMC [Kroening et al., 2004] and `explain` [Groce et al., 2004], in contrast, produce the optimal explanation, which avoids taking the branch guarding the missing return location. With static single assignment [Alpern et al., 1988], the change for the untaken branch is represented by a pair of Δs (one for the condition and one for the control flow change) and there is no need to insert new control locations. For errors best explained purely in terms of control flow, concrete explanation is just as expressive as abstract execution.

Although MAGIC is capable of model-checking the TCAS examples [Rothermel and Harrold, 1999] used in the original presentation of distance metric based explanation [Groce, 2004], it fails to produce explanations for the errors discovered. The TCAS counterexamples are very lengthy and require many alignment variables. To produce non-spurious executions, numerous predicates must be introduced at most control locations in the program, although the values on which the predicates are based are only assigned to at the beginning of execution. The PBS constraints produced by MAGIC for TCAS are simply too large for PBS to solve (e.g., 287,081 variables and 48,432,204 clauses, with 16,374 of the variables appearing in the pseudo-Boolean constraints), as a result of the very large number of alignment possibilities and predicates required.

In contrast, the SSA unwinding used by CBMC only has to produce constraints for these inputs at possible assignment or branching points. Because the TCAS code is essentially a computation of a function with a very small range (3 values) from a large set of unaltered inputs, CBMC and `explain`, despite using full 32-bit integers in place of abstract values, produce a much simpler 0-1 ILP problem than MAGIC.

9.5.3 Choosing a Distance Metric

It is probably incorrect to ascribe these differences to concrete vs. abstract explanation. A tool using SSA with abstract assignments would likely match or improve upon the results produced by concrete explanation[1]. To our knowledge, no tool supporting SSA and predicate abstraction currently exists[2]. For the time being, for some programs, CBMC and `explain` may be the best model checking tools for error explanation. It may be that the counter-intuitive SSA-based metric is, in fact, better for some errors than the alignment-based metric used for abstract explanations in this paper. Ideally, the choice of an SSA or alignment based distance metric is orthogonal to the use of an abstract state-space.

The advantages shown by concrete explanation are empirical, and plausibly understood as artifacts of alignment vs. SSA form. The arguments in Section 9.5.1 are more definitive. It is reasonable to conclude that abstract explanation is superior to

[1]In such a tool, SSA would be applied to the abstracted program, $A(P)$ to generate a BMC instance, in place of the current direct unwinding of the transition relation.

[2]CBMC is used for predicate abstraction, but only to produce a transition relation for non-BMC model checking.

(and subsumes) concrete explanation, but that the choice of a distance metric can negate this theoretical superiority.

The choice of which explanation approach to use is, in practice, not something that a user will typically be forced to think about. Error explanation is an "afterthought" in the sense that the choice of explanation method will be driven by the choice of a model checking tool. If a program is more suitable for CBMC than for MAGIC, concrete explanation is likely to be used — cases where bounding the search depth is easy, or the exact semantics of ANSI C overflow or pointer behavior is crucial will typically fall into this category. On the other hand, errors in programs requiring substantial abstraction or requiring modular specification and verification (or specified with LTL properties) will naturally be explained using MAGIC's abstract explanation features. The search for errors or verification is primary; the need for explanation is a secondary concern that will seldom be the determining factor in choosing which model checker to use.

Chapter 10

Conclusions

My pen halts, though I do not. Reader, you will walk no more with me. It is time we both take up our lives.

- Gene Wolfe, The Citadel of the Autarch

10.1 Conclusions

Any final conclusion about the supremacy of explanation based on distance metrics, beyond the existential claim that for *some programs* and *some errors* it works very well, would be premature. The scoring method proposed by Renieris and Reiss provides a quantitative means for comparing fault localizations; unfortunately, in the absence of competing tools and methods that apply to the same programs and errors, the raw scores are difficult to assess, other than as a marked improvement on raw counterexamples. Improvement over counterexamples demonstrates that in the cases considered, explanation was (almost always) of considerable value, given the (reasonable) assumption that a localization to the precise neighborhood of the error is valuable.

It is unlikely, explanation being at heart, perhaps, a *psychological* notion, that any one approach to error explanation can ever be *proven* to be optimal or even "correct" in a purely logical sense. The best demonstration of superiority would lie in user testing to empirically demonstrate that programmers' efficiency in debugging is improved by an explanation technique. That said, the approach to explanation presented in here is

- based on David Lewis' widely used [Galles and Pearl, 1997; Sosa and Tooley, 1993] notion of causality [Lewis, 1973a] and
- provides an effectively computable notion of explanation.

Experimental results do indicate that the method often produces very effective *fault localization* information, and that this localization is, in the examples considered, (on average) much better than that provided by testing-based localization methods or other model checking localizations. Again, as suggested in the introduction, if we accept the claim that localization/isolation is the most difficult part of the debugging task [Vesey, 1985], this quantitative demonstration of effective localization provides a strong expectation that the technique provides effective explanation as well.

The method has been successfully applied to concrete executions of programs, using a somewhat counter-intuitive distance metric influenced by hypothetical values computed by un-executed code. The novel Δ-slicing algorithm improves explanations, and introduces a notion of slicing that is based on *causality* and works directly with a pair of executions to determine why certain predicates are true in one execution and false in another.

The basic approach can be generalized to apply to abstract executions, use a more intuitive distance metric, and explain Linear Temporal Logic property violations. Experimental results demonstrate the utility of abstract explanation, but also indicate that the original SSA-based metric and tool have some advantages over the implementation of abstract explanation for MAGIC.

The most interesting lesson to be drawn from abstract explanation is that the predicate abstractions introduced to model checking in order to combat the state-space explosion problem are also useful for improving program understanding. The fact that each abstract execution potentially represents many counterexamples or successful executions provides an automatic generalization to the logical causes of an error. It should be possible to exploit this generalization of program behaviors (or the production of a set of predicates that are relevant to a given property, etc.) for other program understanding goals, such as program exploration, reverse engineering, specification mining, etc. — e.g., in cases where a type-inference based static analysis [O'Callahan and Jackson, 1997] or (dynamically discovered) invariants [Ernst et al., 1999] might be used. Rather than viewing the abstract state-spaces automatically produced by software model checkers as disposable artifacts of verification, we must at least consider the possibility that *the abstractions themselves are valuable by-products that can be mined for information.*

10.2 Future Work

10.2.1 SSA and Abstract Explanation

The TCAS and μC/OS-II results indicate that predicate abstraction plus SSA-form BMC might be a fruitful combination for error explanation. The high overhead of introducing alignment variables into the distance metric is the most important motivation for this combination: the occasional production of metric problems that PBS cannot solve is a large drawback to the abstract explanation approach. The cases in which SSA form simply produces a better explanation (e.g., μC/OS-II) also motivate a combination of techniques. While it is reasonable to expect cases in which non-SSA form based metrics produce better results, the one large program for which both methods have been tried receives a (much) better explanation under the SSA form metric. Preliminary experiments with hand-encodings of SSA form versions of abstract programs do suggest that the counter-intuitive metrics may combine poorly in some cases with SSA form. One hypothesis is that the metric must be altered to take into account both the predicates that are relevant at different locations (not natural to SSA form) and the difference between abstraction-based nondeterminism[1] and nondeterminism based on program inputs.

Abstraction makes the presence of irrelevant Δs in an explanation less likely but does not fully eliminate the need for causally-aware slicing. Adapting the Δ-

[1]A completely deterministic program transition in the concrete program may become nondeterministic under an abstraction that is too coarse to, for example, determine if a given branch should be taken.

slicing method [Groce, 2004] used with concrete explanations to an alignment-based distance metric is not obviously sensible; for these reasons, an SSA-based abstract explanation method would appear to be the most practically important advance on the current methods.

10.2.2 Slicing

An appealing compromise between two-phase and one-step slicing would be to compute the original distance metric only over SSA form values and guards present in a *static slice* with respect to the error detected in the original counterexample.

All that would be required is to modify the SSA form distance metric to reflect a static slice of the program and error. Using a dynamic slice based on the counterexample would potentially introduce the "relativity" problems presented by one-step slicing, but a static slice would provide a completely conservative notion of relevance: differences removed by static slicing simply could not be important for understanding the failure in question.

A less concrete area for future research would be an investigation of the conditions under which Δ-slicing differs from some dynamic slice or combination of dynamic slices. The interaction between slicing and the full-execution distance metric remains somewhat unclear: it is possible that some restriction on slicing or change in the metric might eliminate the "relativity of relevance" issue that makes one-step slicing perform badly.

10.2.3 Concurrency

The current MAGIC and `explain` implementations of error explanation do not apply to concurrent programs. CBMC does not support concurrency, and the MAGIC facilities apply only to executions of a single thread (MAGIC does support message-passing concurrency, with reduction of counterexamples to traces in the individual threads). In principle, a technique such as Qadeer's context-bounded approach [Qadeer and Wu, 2004] to concurrency could be used to explain errors by transforming a concurrent example into a sequential model checking problem. Of course, there is no particular difficulty in formulating distance metrics that allow for an interleaving semantics, although it might well be very desirable to include *transposition* of steps as an atomic operation in order to encode the fundamental semantics of interleaving. The primary difficulty with adding concurrency to explanation lies in the poor performance of bounded model checking for concurrent software (and a lack of BMC tools that support concurrency). Recent work has addressed this problem to some degree [Grumberg et al., 2005], but the effectiveness of Bounded Model Checking for concurrent software remains largely unproven.

The JPF implementation of error explanation [Groce and Visser, 2003] supports concurrency, and within the limits of JPF's selection of executions to examine, can produce explanations based on minimal thread scheduling changes. It may be that a similar technique for BMC, based on a more principled technique, such as context-bounding [Qadeer and Wu, 2004] may prove most suitable for explaining concurrency errors. An important question to investigate here is whether an execution with the same (or fewer) context-switches is usually capable of avoiding an error: it seems at least highly plausible that this will be the case, which suggests context-bounding might work well even when the error is produced by a different approach, if the counterexample has few context-switches.

Another possibility, suggested by the most useful results produced by the JPF implementation and the work of Zeller [Choi and Zeller, 2002] would be to consider *only changes in thread scheduling*: base a distance metric on scheduling alone, hold the program inputs constant, and search for a most-similar *schedule* that avoids an error. Whether this would constrain the search space sufficiently to allow for efficient bounded model checking or complete explicit-state model checking is unclear.

A final concern raised by some approached to concurrency is that there might be interference between *partial order reduction* [Peled, 1998] and the search for a most similar execution: it seems possible that the most similar path might not be considered because it is equivalent under the partial order reduction to another, more distant from the counterexample, path.

10.2.4 Explicit-State Approaches

The JPF implementation of error explanation [Groce and Visser, 2003] demonstrates that explanation can be incorporated into an explicit-state model checker. At present, explicit-state model checkers such as Java PathFinder 2 [Visser et al., 2003], Bogor [Robby et al., 2004], and SPIN [Holzmann, 2003] (or dSPIN [Demartini et al., 1999]) are more popular for exploring the behavior of certain kinds of programs than abstraction-based, bounded, or symbolic model checkers. The reasons for this preference include the success of partial-order reductions, close modeling of actual execution semantics, handling of dynamic object and thread creation, and other related factors. Efficient (possibly heuristic) methods for applying distance metric based explanation in the context of explicit-state checkers would provide an alternative to the BMC (and SAT/PBS) dependent approach presented here. Explicit-state model checkers are currently perhaps the best choice for verifying/debugging concurrent software: research into explicit-state techniques is therefore also an alternate approach to those suggested above for addressing concurrency.

10.2.5 Explanation and Symbolic Execution

The generalization achieved by predicate abstraction should also be obtainable through model checking techniques based on symbolic execution of programs. The techniques proposed by Khurshid, Păsăreanu, and Visser [Khurshid et al., 2003; Păsăreanu and Visser, 2004] provide an alternative means to (something like) the same ends as predicate abstraction. The use of explicit-state model checking to handle concurrency in these cases might address some of the difficulties of concurrency. A possible drawback is that infeasible paths might be harder to avoid in this case, and that in an explicit-state context, only an approximation of the closest execution might be obtainable. The second objection depends on the distance metric used: the formulation of a distance metric for this approach does not appear to pose any fundamental difficulties beyond those encountered in the abstract case, though including concurrency does require some attention, as noted above.

10.2.6 Metrics for More Complex Counterexample Forms

The metrics considered in this work address executions of programs, either finite or stem and cycle infinite executions (for LTL properties). Counterexamples to properties cannot always be given as executions, however [Clarke and Veith, 2003]. For CTL properties, a branching structure in the counterexample [Clarke et al., 2002] may be necessary. To demonstrate that a program cannot simulate a specification, a game strategy involving branching on system and specification moves may be produced, as in the MAGIC tool [Chaki et al., 2004a].

Extending the distance-metric based approaches to these kinds of counterexample requires producing metrics for distances between tree-like structures, and an efficient method for computing these distances. Simulation in particular offers a number of potentially interesting questions to address. Is a distance metric based on distance in the simulation lattice the best method? If a Levenshtein distance is to be used, what basic operations reflect the underlying reasons why one system fails to simulate another?

10.2.7 Further Empirical Evaluation and User Studies

A more extensive empirical study of explanation approaches and distance metrics is in order, as are user studies to discover how genuinely useful explanations are for debugging.

An empirical justification of the evaluation method proposed by Renieris and Reiss would serve to improve confidence in any fault localization techniques that provide evidence of good results by their measure. The intuition behind the hypothetical notion of a perfect debugger (used to justify the breadth-first search with termination as soon as a faulty node is encountered — see Section 5.2) seems reasonable, as does the notion of measuring a distance in the Program Dependency Graph to the actual error from the reported location. Nonetheless, it would be highly desirable to show a direct correlation between better scores under the evaluation method and actual user debugging experience.

10.2.8 Automated Program Correction

Using distance metrics to generate maximally similar executions that avoid an error naturally introduces the possibility of using a distance metric to generate the closest *program* that avoids an error. That is, rather than localizing a fault in the indirect manner, a distance metric could be used to discover a *correction* for an error, making localization merely a side effect of the real goal of the debugging task.

A general outline of such an approach might work as follows:

1. Use the `explain` engine to encode a BMC query to determine, under a fixed set of possible program mutations [Budd, 1980], a mutation that:
    - Minimizes the distance to the original program.
    - Satisfies all properties for the inputs in the counterexample.

2. Model check the proposed fix to see if it introduces any new counterexamples.

3. If the new program is error-free (or error-free up to some bounded length, at least), present it to the user as a correction for the original error.

4. If the proposed fix introduces other errors, add a blocking clause to remove this solution and return to the first step of the process.

The iterative refinement is required because the first pseudo-Boolean query can check only whether the new program works correctly for a fixed set of inputs (e.g., the inputs used in the counterexample); finding a program that works for all inputs would require quantifier alternation.

The encoding required for the first step depends on the set of program mutations allowed. For many possible mutations, this encoding is no more difficult than that required to allow for the current approach: the program counter for a particular source line is available to the conversion routines, and a case split on possible alternatives could be introduced into the transition relation.

Experiments with small examples, however, indicate that a very general set of allowed mutations produces numerous "corrections" that apply only to one set of inputs. For practical purposes, to avoid a long sequence of iterations, only mutations corresponding to very common program errors might be allowed: off by one errors, bad conditional choices, and common loop and pointer mistakes. Unfortunately, in the case of simple, common errors it is unclear that the time to produce and verify (by hand) an automatically produced correction would be an improvement over hand debugging with a good localization. The preliminary results of Jobstmann, Staber, Griesmayer, and Bloem indicate that the possibility of automated correction at least merits investigation [Jobstmann et al., 2005; Staber et al., 2005].

10.3 Summary

We have presented a novel approach to error explanation and fault localization, based on distance metrics for program executions. The use of distance metrics is suggested by common intuition [Groce and Visser, 2003; Renieris and Reiss, 2003; Zeller and Hildebrandt, 2002] and an important theory of causality [Lewis, 1973a]. More importantly, the use of distance metrics is justified by empirical evidence of generally high quality fault localizations for a number of case studies, as reported in Chapters 5 and 9. The utility of automatically discovered predicates in abstract explanation suggests that tool-generated abstractions used in verification are potentially valuable artifacts for the purpose of program understanding.

Numerous directions for future error explanation research are presented above; the topic of error explanation has recently attracted a considerable amount of attention (an entire session of the 2004 SIGSOFT Symposium on the Foundations of Software Engineering was devoted to the topic "Error Explanation" [Dwyer, 2004]). We expect that the work presented here is a promising beginning, rather than a conclusion, in the field of model checking for error explanation and fault localization.

Bibliography

Note: Chapters 3, 4, and 5 of this document are partly based on the text of a TACAS 2004 paper [Groce, 2004]; Chapter 6 is partly based on a BMC 2004 paper [Groce and Kroening, 2004]; Chapters 7, 8, and 9 are partly based on an FSE 2004 paper [Chaki et al., 2004c].

Hira Agrawal, Joseph Horgan, Saul London, and W. Eric Wong. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering*, pages 143–151, Toulouse, France, October 1995.

Fadi Aloul, Arathi Ramani, Igor Markov, and Karem Sakallah. PBS: A backtrack search pseudo Boolean solver. In *Symposium on the theory and applications of satisfiability testing (SAT)*, pages 346–353, Cincinnati, OH, May 2002.

Bowen Alpern, Mark Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages*, pages 1–11, San Diego, CA, January 1988.

María Alpuente, Marco Comini, Santiago Escobar, Moreno Falaschi, and Salvador Lucas. Abstract diagnosis of functional programs. In *Logic Based Program Synthesis and Tranformation, 12th International Workshop*, Madrid, Spain, September 2002.

Paul Anderson and Tim Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering*, Paris, France, July 2001.

AskIgor Website. `http://www.askigor.com`.

Thomas Ball and Stephen Eick. Software visualization in the large. *Computer*, 29 (4):33–43, April 1996.

Thomas Ball, Mayur Naik, and Sriram Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages*, pages 97–105, New Orleans, LA, January 2003.

Thomas Ball and Sriram Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.

Thomas Ball and Sriram Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop Model Checking of Software*, pages 103–122, Toronto, Canada, May 2001.

Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut Fur Informatik, 1995.

Jonathan Bennett. Counterfactuals and temporal direction. *Philosophical Review*, 93:57–91, 1984.

Jonathan Bennett. Event causation: The counterfactual analysis. In James E. Tomberlin, editor, *Philosophical Perspectives, 1, Metaphysics*. Ridgeview Publishing Company, 1987.

Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *ERCIM Workshop in Formal Methods for Industrial Critical Systems*, volume 66 of *Electronic Notes in Theoretical Computer Science*, University of Malaga, Spain, July 2002.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Amsterdam, The Netherlands, March 1999.

Timothy Alan Budd. *Mutation Analysis of Program Test Data*. 1980. PhD thesis, Yale University.

CBMC Website. `http://www.cs.cmu.edu/~modelcheck/cbmc/`.

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering*, pages 385–395, Portland, OR, May 2003a.

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004a.

Sagar Chaki, Edmund M. Clarke, Alex Groce, Joel Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2-3):129–166, September-November 2004b. Special issue on software model checking.

Sagar Chaki, Edmund M. Clarke, Alex Groce, and Ofer Strichman. Predicate abstraction with minimum predicates. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 19–34, L'Aquila, Italy, October 2003b.

Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 73–82, Newport Beach, CA, November 2004c.

William Chan. Temporal-logic queries. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 450–463, Chicago, IL, July 2000.

Marsha Chechik and Arie Gurfinkel. Proof-like counter-examples. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 160–175, Warsaw, Poland, April 2003.

Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220, Rome, Italy, July 2002.

Edmund M. Clarke and E. Emerson. The design and synthesis of synchronization skeletons using temporal logic. In *Workshop on Logics of Programs*, pages 52–71, Yorktown Heights, NY, May 1981.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, Chicago, IL, July 2000a.

Edmund M. Clarke, Orna Grumberg, Ken McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*, pages 427–432, San Francisco, CA, June 1995.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000b.

Edmund M. Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *IEEE Symposium on Logic in Computer Science*, pages 19–29, Copenhagen, Denmark, July 2002.

Edmund M. Clarke, Somesh Jha, and Will Marrero. Verifying security protocols with Brutus. *ACM Transactions of Software Engineering and Methodology*, 9(4): 443–487, October 2000c.

Edmund M. Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pages 208–224, 2003.

Holger Cleve and Andreas Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, St. Louis, MO, May 2005. To appear.

Jamie Cobleigh, Dimitra Giannakopoulou, and Corina Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346, Warsaw, Poland, April 2003.

Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezze. Using symbolic execution for verifying safety-critical systems. In *European Software Engineering Conference/Foundations of Software Engineering*, pages 142–151, Vienna, Austria, September 2001.

Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proceedings of the 6th International SPIN Workshop Model Checking of Software*, pages 261–276, Toulouse, France, September 1999.

Edsger W. Dijkstra. A simple axiomatic basis for programming language constructs. Lecture notes from the International Summer School on Structured Programming and Programmed Structures, 1973.

Nii Dodoo, Alan Donovan, Lee Lin, and Michael Ernst. Selecting predicates for implications in program analysis. URL `http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications-abstract.ht%ml`. `http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.ps`, 2000.

Richard Durbin, Sean Eddy, Aanders Krogh, and Graeme Mitchison. *Biological sequence analysis: Probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998.

Matthew Dwyer, editor. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Newport Beach, CA, November 2004.

Michael Ernst, Jake Cockrell, William Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, May 1999.

David Galles and Judea Pearl. Axioms of causal relevance. *Artificial Intelligence*, 97 (1-2):9–43, 1997.

Robert Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, 1995.

Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, Haifa, Israel, June 1997.

Alex Groce. Error explanation with distance metrics. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, Barcelona, Spain, March-April 2004.

Alex Groce and Daniel Kroening. Making the most of BMC counterexamples. In *Workshop on Bounded Model Checking*, pages 71–84, Boston, MA, July 2004.

Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with `explain`. In *Proceedings of the 16th International Conference on Computer Aided Verification*, pages 453–456, Boston, MA, July 2004.

Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *International Symposium on Software Testing and Analysis*, pages 12–21, Rome, Italy, July 2002.

Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Proceedings of the 10th International SPIN Workshop Model Checking of Software*, pages 121–135, Portland, OR, May 2003.

Alex Groce and Willem Visser. Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*, 2004. Online first.

Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Principles of Programming Languages*, pages 122–131, Long Beach, CA, January 2005.

Arie Gurfinkel, Benet Devereux, and Marsha Chechik. Model exploration with temporal logic query checking. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 139–148, Charleston, SC, November 2002.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, Orland, FL, May 2002.

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

Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, Portland, OR, January 2002.

C. A. R. Hoare. An axiomatic basis for computer programming (reprint). *Communications of the ACM*, 26(1):53–56, 1983.

Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

Nicholas J. Hopper, Sanjit A. Seshia, and Jeannette M. Wing. A comparison and combination of theory generation and model checking for security protocol analysis. In *Workshop on Formal Methods in Computer Security*, Chicago, IL, July 2000.

Paul Horwich. *Asymmetries in Time*, pages 167–176. MIT Press, 1987.

Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, May 1992.

David Hume. *A Treatise of Human Nature*. London, 1739.

David Hume. *An Enquiry Concerning Human Understanding*. London, 1748.

HoonSang Jin, Kavita Ravi, and Fabio Somenzi. Fate and free will in error traces. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–458, Grenoble, France, April 2002.

Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. URL `http://www.ist.tugraz.at/verify/pub/Projects/ProgramRepair/repair.ps`. Unpublished manuscript, 2005.

James Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, pages 467–477, Orlando, FL, May 2002.

Sarfraz Khurshid, Corina Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, Warsaw, Poland, April 2003.

Jaegwon Kim. Causes and counterfactuals. *Journal of Philosophy*, 70:570–572, 1973.

Darrell Kindred and Jeannette Wing. Fast, automatic checking of security protocols. In *USENIX Workshop on Electronic Commerce*, pages 41–52, Oakland, CA, November 1996.

Gabriella Kókai, László Harmath, and Tibor Gyimóthy. Algorithmic debugging and testing of Prolog programs. In *Workshop on Logic Programming Environments*, pages 14–21, Leuven, Belgium, July 1997.

Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Barcelona, Spain, March-April 2004.

Nirman Kumar, Viraj Kumar, and Mahesh Viswanathan. On the complexity of error explanation. In *Verification, Model Checking and Abstract Interpretation*, pages 448–464, Paris, France, January 2005.

Robert Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata- Theoretic Approach*. Princeton University Press, 1995.

K. Rustan Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 2004. To appear.

David Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973a.

David Lewis. *Counterfactuals*. Harvard University Press, 1973b. [revised printing 1986].

Peter Lucas. Analysis of notions of diagnosis. *Artificial Intelligence*, 105(1-2):295–343, 1998.

J. L. Mackie. Causes and conditions. *American Philosophical Quarterly*, 2:245–264, 1965.

Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.

Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Model-based debugging of Java programs. In *Workshop on Automatic Debugging*, Munich, Germany, August 2000.

Wolfgang Mayer and Markus Stumptner. Model-based debugging using multiple abstract models. In *International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 2003.

Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

Edjard Mota, Edmund M. Clarke, W. de Oliveira, Alex Groce, J. Kanda, and M. Falcao. VeriAgent: an approach to integrating UML and formal verification tools. In *Sixth Brazilian Workshop on Formal Methods*, pages 111–129, Universidade Federal de Campina Grande, Brazil, October 2003. Electronic Notes in Theoretical Computer Science 95 (May 2004).

μC/OS-II Website. `http://www.ucos-ii.com/`.

Kedar Namjoshi. Certifying model checkers. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 2–13, Paris, France, July 2001.

Kedar Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 435–449, Chicago, IL, July 2000.

P. Pandurang Nayak and Brian Williams. Fast context switching in real-time propositional reasoning. In *National Conference on Artificial Intelligence*, pages 50–56, Providence, RI, July 1997.

Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

Doron Peled. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 17–28, Vancouver, BC, Canada, June-July 1998.

Doron Peled, Amir Pnueli, and Lenore D. Zuck. From falsification to verification. In *Foundations of Software Technology and Theoretical Computer Science*, pages 292–304, Bangalore, India, December 2001.

Corina Păsăreanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of the 11th International SPIN Workshop Model Checking of Software*, pages 164–181, Barcelona, Spain, April 2004.

Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 2003.

Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *Conference on Programming Language Design and Implementation*, pages 14–24, Washington, DC, June 2004.

Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351, Torino, Italy, April 1982.

Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, Barcelona, Spain, March-April 2004.

Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, pages 30–39, Montreal, Canada, October 2003.

Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *European Software Engineering Conference*, pages 432–449, Zurich, Switzerland, September 1997.

Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 404–420, Barcelona, Spain, 2004.

Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.

Wesley Salmon. Probabilistic causality. *Pacific Philosophical Quarterly*, 61:50–74, 1980.

David Sankoff and Joseph Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison Wesley, 1983.

Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

Natasha Sharygina and Doron Peled. A combined testing and verification approach for software reliability. In *Formal Methods Europe*, pages 611–628, Berlin, Germany, March 2001.

ShengYu Shen, Ying Qin, and Sikun Li. Bug localization of hardware system with control flow distance minimization. In *International Workshop on Logic and Synthesis*, Temecula, CA, June 2004a.

ShengYu Shen, Ying Qin, and Sikun Li. Debugging complex counterexample of hardware system using control flow distance metrics. In *IEEE Midwest Symposium on Circuits and Systems*, pages 501–504, Hiroshima, Japan, July 2004b.

ShengYu Shen, Ying Qin, and Sikun Li. Localizing errors in counterexample with iteratively witness searching. In *Automated Technology for Verification and Analysis*, pages 456–469, Taipei, Taiwan, October-November 2004c.

ShengYu Shen, Ying Qin, and Sikun Li. Minimizing counterexample with unit core extraction and incremental SAT. In *Verification, Model Checking, and Abstract Interpretation*, pages 298–312, Paris, France, January 2005.

Reid Simmons and Charles Pecheur. Automating model checking for autonomous systems. In *AAAI Spring Symposium on Real-Time Autonomous Systems*, 2000.

Ernest Sosa and Michael Tooley, editors. *Causation*. Oxford University Press, 1993.

Stefan Staber, Barbara Jobstmann, and Roderick Bloem. Diagnosis is repair. Unpublished manuscript, 2005.

Robert Stalnaker. A theory of conditionals. In N. Rescher, editor, *Studies in Logical Theory*. Oxford University Press, 1968.

Perdita Stevens and Colin Stirling. Practical model-checking using games. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 85–101, Lisbon, Portugal, March-April 1998.

Li Tan and Rance Cleaveland. Evidence-based model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 455–470, Copenhagen, Denmark, July 2002.

Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

I. Vesey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

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

Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. 1979. PhD thesis, University of Michigan.

Franz Wotawa. On the relationship between model-based debugging and programm mutation. In *International Workshop on Principles of Diagnosis*, Sansicario, Italy, March 2001.

Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002.

Andreas Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 1–10, Charleston, SC, November 2002.

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

Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *International Conference on Software Engineering*, pages 319–329, Portland, OR, May 2003.

Appendix A

Command Line Options for the `explain` Tool

```
Usage:                          Purpose:
 explain [-?] [-h] [--help]         show help
 explain file.c ...                 source file names
Additonal options:
 --program-only                   only show program expression
 --function name                  set main function name
 --no-simplify                    do not simplify
 --all-claims                     keep all claims
 --unwind nr                      unwind nr times
 --unwindset nr                   unwind given loop nr times
 --claims-only                    only show claims
 --decide                         run decision procedure
 --dimacs                         generate CNF in DIMACS format
 --document-subgoals              generate subgoals documentation
 --no-remove-equations            do not remove obsolete equations
 --no-substitution                do not perform substitution
 --no-simplify-if                 do not simplify ?:
 --no-assertions                  ignore assertions
 --no-bounds-check                do not do array bounds check
 --no-div-by-zero-check           do not do division by zero check
 --no-pointer-check               do not do pointer check
 --bound nr                       number of transitions
```

--module name	module to unwind
--counterexample file	counterexample file to write
--explain-error file	counterexample file to explain
--slice-now	slice as you explain
--minimize-deltas	attempt to remove unneeded deltas
--deltas-past-error	use counterexample values past the error
--interventions	use metric based on arbitrary interventions
--inputs-only	only generate deltas for inputs
--inputs-guards-only	only generate deltas for inputs and guards
--guard-weight	changes the weight for guard deltas
--input-weight	changes the weight for input deltas
--write-deltas file	write changes to a file
--write-delta-values file	write changes and actual values to a file
--find-error	try to find a different error path
--maximize	maximize distance to path
--assume file	restrict to satisfy predicates in file
--assume-not	use negation of assumptions
--assume-or	use OR, not AND for assumptions
--find-causes	check delta for causal dependence
--closest-causes	check deltas in closest positive for c.d.
--no-prior-changes	do not allow changes prior to cause delta
--inputs-beginning	treat all inputs as occuring initially
--check-cause file	check for causal dependence on predicates in file
--check-not	use negations of causes to check
--check-or	use OR on causes, not AND
--write-causes file	write causes as predicates to a file
--effect file	check effect, not error
--effect-not	use negations of effects to check
--effect-or	use OR on effects, not AND
--pbs-path	path at which pbs executable resides

Appendix B

TCAS Version #1 Counterexample

```
file tcasv1.c: Parsing
Converting
Starting Bounded Model Checking
Unwinding recursion iteration 0 (c::main)
...
size of program expression: 165 assignments
Generated 41 claims, 30 remaining
Passing to decision procedure
...
Running decision procedure
Solving with ZChaff version ZChaff 2003.6.16
7317 variables, 23483 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
Counterexample:
Initial State
----------------------------------------------------
  ASTBeg=FALSE
  ASTDownRA=FALSE
  ASTEn=FALSE
  ASTUnresRA=FALSE
  ASTUpRA=FALSE
  Alt_Layer_Value=0
  Climb_Inhibit=FALSE
  Cur_Vertical_Sep=0
```

```
  Down_Separation=0
  High_Confidence=FALSE
  Other_Capability=0
  Other_RAC=0
  Other_Tracked_Alt=0
  Own_Tracked_Alt=0
  Own_Tracked_Alt_Rate=0
  P1_ACond=FALSE
  P1_BCond=FALSE
  P2_ACond=FALSE
  P3_BCond=FALSE
  P5_ACond=FALSE
  Positive_RA_Alt_Thresh= 0, 0, 0, 0
  PrA=FALSE
  PrB=FALSE
  Two_of_Three_Reports_Valid=FALSE
  Up_Separation=0
State 1
----------------------------------------------------
  r=(assignment removed)
State 2
----------------------------------------------------
  Input_Cur_Vertical_Sep=2061
State 3
----------------------------------------------------
  Input_High_Confidence=TRUE
State 4
----------------------------------------------------
  Input_Two_of_Three_Reports_Valid=FALSE
State 5
----------------------------------------------------
  Input_Own_Tracked_Alt=25350
State 6
----------------------------------------------------
  Input_Own_Tracked_Alt_Rate=-31071 (11111111111111111000011010100001)
State 7
----------------------------------------------------
  Input_Other_Tracked_Alt=98557
```

```
State 8
----------------------------------------------------
  Input_Alt_Layer_Value=2
State 9
----------------------------------------------------
  Input_Up_Separation=65089
State 10
----------------------------------------------------
  Input_Down_Separation=640
State 11
----------------------------------------------------
  Input_Other_RAC=2
State 12
----------------------------------------------------
  Input_Other_Capability=2
State 13
----------------------------------------------------
  Input_Climb_Inhibit=FALSE
State 14
----------------------------------------------------
  Layer_Positive_RA_Alt_Thresh=(assignment removed)
State 30 file tcasv1.c line 207 function c::main
----------------------------------------------------
  Cur_Vertical_Sep=2061
State 31 file tcasv1.c line 208 function c::main
----------------------------------------------------
  High_Confidence=TRUE
State 32 file tcasv1.c line 209 function c::main
----------------------------------------------------
  Two_of_Three_Reports_Valid=FALSE
State 33 file tcasv1.c line 210 function c::main
----------------------------------------------------
  Own_Tracked_Alt=25350
State 34 file tcasv1.c line 211 function c::main
----------------------------------------------------
  Own_Tracked_Alt_Rate=0
State 35 file tcasv1.c line 212 function c::main
----------------------------------------------------
```

```
  Other_Tracked_Alt=98557
State 36 file tcasv1.c line 213 function c::main
----------------------------------------------------
  Alt_Layer_Value=2
State 37 file tcasv1.c line 214 function c::main
----------------------------------------------------
  Up_Separation=65089
State 38 file tcasv1.c line 215 function c::main
----------------------------------------------------
  Down_Separation=640
State 39 file tcasv1.c line 216 function c::main
----------------------------------------------------
  Other_RAC=2
State 40 file tcasv1.c line 217 function c::main
----------------------------------------------------
  Other_Capability=2
State 41 file tcasv1.c line 218 function c::main
----------------------------------------------------
  Climb_Inhibit=FALSE
State 43 file tcasv1.c line 65 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 0, 0, 0
State 44 file tcasv1.c line 66 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 0, 0
State 45 file tcasv1.c line 67 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 0
State 46 file tcasv1.c line 68 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 740
State 47
----------------------------------------------------
  enabled=(assignment removed)
State 48
----------------------------------------------------
  tcas_equipped=(assignment removed)
State 49
```

```
----------------------------------------------------
  intent_not_known=(assignment removed)
State 50
----------------------------------------------------
  need_upward_RA=FALSE
State 51
----------------------------------------------------
  need_downward_RA=FALSE
State 52
----------------------------------------------------
  alt_sep=(assignment removed)
State 53 file tcasv1.c line 133 function c::alt_sep_test
----------------------------------------------------
  ASTBeg=TRUE
State 54 file tcasv1.c line 135 function c::alt_sep_test
----------------------------------------------------
  enabled=TRUE
State 55 file tcasv1.c line 136 function c::alt_sep_test
----------------------------------------------------
  tcas_equipped=FALSE
State 56 file tcasv1.c line 137 function c::alt_sep_test
----------------------------------------------------
  intent_not_known=FALSE
State 57 file tcasv1.c line 139 function c::alt_sep_test
----------------------------------------------------
  alt_sep=0
State 59 file tcasv1.c line 143 function c::alt_sep_test
----------------------------------------------------
  ASTEn=TRUE
State 60
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::upward_preferred_1=0
State 61
----------------------------------------------------
  c::Non_Crossing_Biased_Climb::0::upward_crossing_situation_1=(assignment re-
moved)
State 62
----------------------------------------------------
```

```
  Non_Crossing_Biased_Climb::0::result_1=FALSE
State 63 file tcasv1.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_1=65089
State 65 file tcasv1.c line 97 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::upward_preferred_1=1
State 67 file tcasv1.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_2=TRUE
State 70 file tcasv1.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_3=TRUE
State 73 file tcasv1.c line 73 function c::ALIM
----------------------------------------------------
  tmp_4=640
State 75 file tcasv1.c line 100 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::result_1=TRUE
State 82 file tcasv1.c line 106 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  tmp=TRUE
State 85 file tcasv1.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_7=TRUE
State 87 file tcasv1.c line 144 function c::alt_sep_test
----------------------------------------------------
  need_upward_RA=TRUE
State 88
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::upward_preferred_1=0
State 89
----------------------------------------------------
  c::Non_Crossing_Biased_Descend::0::upward_crossing_situation_1=(assignment re-
moved)
State 90
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::result_1=FALSE
```

```
State 91 file tcasv1.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_9=65089
State 93 file tcasv1.c line 115 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::upward_preferred_1=0
State 101 file tcasv1.c line 83 function c::Own_Above_Threat
----------------------------------------------------
  tmp_12=FALSE
State 109 file tcasv1.c line 122 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::result_1=TRUE
State 110 file tcasv1.c line 124 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  tmp_8=TRUE
State 113 file tcasv1.c line 83 function c::Own_Above_Threat
----------------------------------------------------
  tmp_15=FALSE
State 115 file tcasv1.c line 145 function c::alt_sep_test
----------------------------------------------------
  need_downward_RA=FALSE
State 120 file tcasv1.c line 153 function c::alt_sep_test
----------------------------------------------------
  ASTUpRA=TRUE
State 121 file tcasv1.c line 154 function c::alt_sep_test
----------------------------------------------------
  alt_sep=1
State 127 file tcasv1.c line 166 function c::alt_sep_test
----------------------------------------------------
  r=1
State 129 file tcasv1.c line 229 function c::main
----------------------------------------------------
  PrA=FALSE
State 130 file tcasv1.c line 230 function c::main
----------------------------------------------------
  PrB=TRUE
State 131 file tcasv1.c line 232 function c::main
----------------------------------------------------
```

```
  Layer_Positive_RA_Alt_Thresh=640
State 132 file tcasv1.c line 236 function c::main
----------------------------------------------------
  P1_ACond=FALSE
State 133 file tcasv1.c line 239 function c::main
----------------------------------------------------
  P1_BCond=FALSE
State 136 file tcasv1.c line 248 function c::main
----------------------------------------------------
  P2_ACond=FALSE
State 138 file tcasv1.c line 255 function c::main
----------------------------------------------------
  P3_BCond=TRUE
Failed assertion: assertion file tcasv1.c line 257 function c::main
Symbols used in assertion:
  P3_BCond=TRUE
  PrB=TRUE
Total number of steps: 74
Sum of values: 551058
Writing counterexample file tcasv1.ce...
VERIFICATION FAILED
Runtime SAT: 0.658s
1.180u 0.030s 0:01.20 100.8%        0+0k 0+0io 1142pf+0w
```

Appendix C

TCAS Version #1 First Explanation

```
Parsing tcasv1.c
Converting
Checking tcasv1
Starting Bounded Model Checking
Reading counterexample from file tcasv1.ce...
...
size of program expression: 165 assignments
Generated 41 claims, 38 remaining
Passing to decision procedure
Running decision procedure
Solving with PBS - Pseudo Boolean/CNF Solver and Optimizer
8267 variables, 26529 clauses
PBS checker: system is SATISFIABLE (distance 10)
Counterexample:
Initial State
----------------------------------------------------
  ASTBeg=FALSE
  ASTDownRA=FALSE
  ASTEn=FALSE
  ASTUnresRA=FALSE
  ASTUpRA=FALSE
  Alt_Layer_Value=0
```

Climb_Inhibit=FALSE
Cur_Vertical_Sep=0
Down_Separation=0
High_Confidence=FALSE
Non_Crossing_Biased_Climb::0::result_1=FALSE
Non_Crossing_Biased_Climb::0::upward_preferred_1=0
Non_Crossing_Biased_Descend::0::result_1=FALSE
Non_Crossing_Biased_Descend::0::upward_preferred_1=0
Other_Capability=0
Other_RAC=0
Other_Tracked_Alt=0
Own_Tracked_Alt=0
Own_Tracked_Alt_Rate=0
P1_ACond=FALSE
P1_BCond=FALSE
P2_ACond=FALSE
P3_BCond=FALSE
P5_ACond=FALSE
Positive_RA_Alt_Thresh= 0, 0, 0, 0
PrA=FALSE
PrB=FALSE
Two_of_Three_Reports_Valid=FALSE
Up_Separation=0
need_downward_RA=FALSE
need_upward_RA=FALSE
Input_Alt_Layer_Value=0
Input_Climb_Inhibit=TRUE
Input_Cur_Vertical_Sep=65841
Input_Down_Separation=159
Input_High_Confidence=TRUE
Input_Other_Capability=2
Input_Other_RAC=0
Input_Other_Tracked_Alt=69497
Input_Own_Tracked_Alt=27606
Input_Own_Tracked_Alt_Rate=-31071 (11111111111111111000011010100001)
Input_Two_of_Three_Reports_Valid=FALSE
Input_Up_Separation=81309
tmp=FALSE

```
  tmp_1=0
  tmp_10=FALSE
  tmp_11=0
  tmp_12=FALSE
  tmp_13=FALSE
  tmp_14=1073741824
  tmp_15=FALSE
  tmp_2=FALSE
  tmp_3=FALSE
  tmp_4=0
  tmp_5=FALSE
  tmp_6=1073741824
  tmp_7=FALSE
  tmp_8=FALSE
  tmp_9=0
State 16 file tcasv1.c line 207 function c::main
----------------------------------------------------
  Cur_Vertical_Sep=65841
State 17 file tcasv1.c line 208 function c::main
----------------------------------------------------
  High_Confidence=TRUE
State 18 file tcasv1.c line 209 function c::main
----------------------------------------------------
  Two_of_Three_Reports_Valid=FALSE
State 19 file tcasv1.c line 210 function c::main
----------------------------------------------------
  Own_Tracked_Alt=27606
State 20 file tcasv1.c line 211 function c::main
----------------------------------------------------
  Own_Tracked_Alt_Rate=0
State 21 file tcasv1.c line 212 function c::main
----------------------------------------------------
  Other_Tracked_Alt=69497
State 22 file tcasv1.c line 213 function c::main
----------------------------------------------------
  Alt_Layer_Value=0
State 23 file tcasv1.c line 214 function c::main
----------------------------------------------------
```

```
  Up_Separation=81309
State 24 file tcasv1.c line 215 function c::main
----------------------------------------------------
  Down_Separation=159
State 25 file tcasv1.c line 216 function c::main
----------------------------------------------------
  Other_RAC=0
State 26 file tcasv1.c line 217 function c::main
----------------------------------------------------
  Other_Capability=2
State 27 file tcasv1.c line 218 function c::main
----------------------------------------------------
  Climb_Inhibit=TRUE
State 29 file tcasv1.c line 65 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 0, 0, 0
State 30 file tcasv1.c line 66 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 0, 0
State 31 file tcasv1.c line 67 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 0
State 32 file tcasv1.c line 68 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 740
State 33 file tcasv1.c line 133 function c::alt_sep_test
----------------------------------------------------
  ASTBeg=TRUE
State 34 file tcasv1.c line 135 function c::alt_sep_test
----------------------------------------------------
  enabled=TRUE
State 35 file tcasv1.c line 136 function c::alt_sep_test
----------------------------------------------------
  tcas_equipped=FALSE
State 36 file tcasv1.c line 137 function c::alt_sep_test
----------------------------------------------------
  intent_not_known=FALSE
State 37 file tcasv1.c line 139 function c::alt_sep_test
```

```
----------------------------------------------------
  alt_sep=0
State 39 file tcasv1.c line 143 function c::alt_sep_test
----------------------------------------------------
  ASTEn=TRUE
State 40 file tcasv1.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_1=81409
State 42 file tcasv1.c line 97 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::upward_preferred_1=1
State 44 file tcasv1.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_2=TRUE
State 47 file tcasv1.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_3=TRUE
State 50 file tcasv1.c line 73 function c::ALIM
----------------------------------------------------
  tmp_4=400
State 52 file tcasv1.c line 100 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::result_1=TRUE
State 59 file tcasv1.c line 106 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  tmp=TRUE
State 62 file tcasv1.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_7=TRUE
State 64 file tcasv1.c line 144 function c::alt_sep_test
----------------------------------------------------
  need_upward_RA=TRUE
State 65 file tcasv1.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_9=81409
State 67 file tcasv1.c line 115 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::upward_preferred_1=1
```

```
State 69 file tcasv1.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_10=TRUE
State 72 file tcasv1.c line 73 function c::ALIM
----------------------------------------------------
  tmp_11=400
State 74 file tcasv1.c line 118 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::result_1=FALSE
State 84 file tcasv1.c line 124 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  tmp_8=FALSE
State 89 file tcasv1.c line 145 function c::alt_sep_test
----------------------------------------------------
  need_downward_RA=FALSE
State 94 file tcasv1.c line 153 function c::alt_sep_test
----------------------------------------------------
  ASTUpRA=TRUE
State 95 file tcasv1.c line 154 function c::alt_sep_test
----------------------------------------------------
  alt_sep=1
State 101 file tcasv1.c line 166 function c::alt_sep_test
----------------------------------------------------
  r=1
State 103 file tcasv1.c line 229 function c::main
----------------------------------------------------
  PrA=FALSE
State 104 file tcasv1.c line 230 function c::main
----------------------------------------------------
  PrB=TRUE
State 105 file tcasv1.c line 232 function c::main
----------------------------------------------------
  Layer_Positive_RA_Alt_Thresh=400
State 106 file tcasv1.c line 236 function c::main
----------------------------------------------------
  P1_ACond=TRUE
State 107 file tcasv1.c line 239 function c::main
----------------------------------------------------
```

```
  P1_BCond=FALSE
State 110 file tcasv1.c line 248 function c::main
----------------------------------------------------
  P2_ACond=FALSE
State 112 file tcasv1.c line 255 function c::main
----------------------------------------------------
  P3_BCond=FALSE
State 114 file tcasv1.c line 260 function c::main
----------------------------------------------------
  P5_ACond=TRUE
Error explanation deltas:
Value changed:  Input_Down_Separation_1#0 from 400 to 159
Value changed:  Down_Separation#1 from 400 to 159
                file tcasv1.c line 215 function c::main
Value changed:  result_1#1 from TRUE to FALSE
                file tcasv1.c line 118 function c::Non_Crossing_Biased_Descend
Value changed:  result_1#3 from TRUE to FALSE
Value changed:  tmp_8#1 from TRUE to FALSE
                file tcasv1.c line 124 function c::Non_Crossing_Biased_Descend
Guard changed:  \guard#1 && tmp_8#1 (\guard#13) was TRUE
                file tcasv1.c line 145 function c::alt_sep_test
Value changed:  result_1#4 from TRUE to FALSE
Value changed:  tmp_8#2 from TRUE to FALSE
Value changed:  P1_ACond#1 from FALSE to TRUE
                file tcasv1.c line 236 function c::main
Value changed:  P3_BCond#1 from TRUE to FALSE
                file tcasv1.c line 255 function c::main
Minimizing deltas
Running decision procedure
Solving with PBS - Pseudo Boolean/CNF Solver and Optimizer
3613 variables, 10831 clauses
PBS checker: system is SATISFIABLE (distance 2)
Deltas after minimization:
Value changed:  Input_Down_Separation_1#0 from 400 to 159
Value changed:  P3_BCond#1 from TRUE to FALSE
                file tcasv1.c line 255 function c::main
CLOSEST SUCCESSFUL PATH FOUND
4.270u 0.120s 0:04.38 100.2%        0+0k 0+0io 1831pf+0w
```

Appendix D

TCAS Version #1 Counterexample (Post-Assumption)

```
file tcasv1a.c: Parsing
Converting
Starting Bounded Model Checking
Unwinding recursion iteration 0 (c::main)
...
size of program expression: 165 assignments
Generated 41 claims, 30 remaining
Passing to decision procedure
...
Running decision procedure
Solving with ZChaff version ZChaff 2003.6.16
7317 variables, 23488 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
Counterexample:
Initial State
----------------------------------------------------
  ASTBeg=FALSE
  ASTDownRA=FALSE
  ASTEn=FALSE
  ASTUnresRA=FALSE
  ASTUpRA=FALSE
  Alt_Layer_Value=0
```

```
  Climb_Inhibit=FALSE
  Cur_Vertical_Sep=0
  Down_Separation=0
  High_Confidence=FALSE
  Other_Capability=0
  Other_RAC=0
  Other_Tracked_Alt=0
  Own_Tracked_Alt=0
  Own_Tracked_Alt_Rate=0
  P1_ACond=FALSE
  P1_BCond=FALSE
  P2_ACond=FALSE
  P3_BCond=FALSE
  P5_ACond=FALSE
  Positive_RA_Alt_Thresh= 0, 0, 0, 0
  PrA=FALSE
  PrB=FALSE
  Two_of_Three_Reports_Valid=FALSE
  Up_Separation=0
State 1
----------------------------------------------------
  r=(assignment removed)
State 2
----------------------------------------------------
  Input_Cur_Vertical_Sep=686
State 3
----------------------------------------------------
  Input_High_Confidence=TRUE
State 4
----------------------------------------------------
  Input_Two_of_Three_Reports_Valid=FALSE
State 5
----------------------------------------------------
  Input_Own_Tracked_Alt=32759
State 6
----------------------------------------------------
  Input_Own_Tracked_Alt_Rate=-31071 (11111111111111111000011010100001)
State 7
```

```
----------------------------------------------------
  Input_Other_Tracked_Alt=32776
State 8
----------------------------------------------------
  Input_Alt_Layer_Value=3
State 9
----------------------------------------------------
  Input_Up_Separation=4
State 10
----------------------------------------------------
  Input_Down_Separation=740
State 11
----------------------------------------------------
  Input_Other_RAC=2
State 12
----------------------------------------------------
  Input_Other_Capability=2
State 13
----------------------------------------------------
  Input_Climb_Inhibit=FALSE
State 14
----------------------------------------------------
  Layer_Positive_RA_Alt_Thresh=(assignment removed)
State 30 file tcasv1a.c line 207 function c::main
----------------------------------------------------
  Cur_Vertical_Sep=686
State 31 file tcasv1a.c line 208 function c::main
----------------------------------------------------
  High_Confidence=TRUE
State 32 file tcasv1a.c line 209 function c::main
----------------------------------------------------
  Two_of_Three_Reports_Valid=FALSE
State 33 file tcasv1a.c line 210 function c::main
----------------------------------------------------
  Own_Tracked_Alt=32759
State 34 file tcasv1a.c line 211 function c::main
----------------------------------------------------
  Own_Tracked_Alt_Rate=0
```

```
State 35 file tcasv1a.c line 212 function c::main
----------------------------------------------------
  Other_Tracked_Alt=32776
State 36 file tcasv1a.c line 213 function c::main
----------------------------------------------------
  Alt_Layer_Value=3
State 37 file tcasv1a.c line 214 function c::main
----------------------------------------------------
  Up_Separation=4
State 38 file tcasv1a.c line 215 function c::main
----------------------------------------------------
  Down_Separation=740
State 39 file tcasv1a.c line 216 function c::main
----------------------------------------------------
  Other_RAC=2
State 40 file tcasv1a.c line 217 function c::main
----------------------------------------------------
  Other_Capability=2
State 41 file tcasv1a.c line 218 function c::main
----------------------------------------------------
  Climb_Inhibit=FALSE
State 43 file tcasv1a.c line 65 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 0, 0, 0
State 44 file tcasv1a.c line 66 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 0, 0
State 45 file tcasv1a.c line 67 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 0
State 46 file tcasv1a.c line 68 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 740
State 47
----------------------------------------------------
  enabled=(assignment removed)
State 48
----------------------------------------------------
```

```
  tcas_equipped=(assignment removed)
State 49
----------------------------------------------------
  intent_not_known=(assignment removed)
State 50
----------------------------------------------------
  need_upward_RA=FALSE
State 51
----------------------------------------------------
  need_downward_RA=FALSE
State 52
----------------------------------------------------
  alt_sep=(assignment removed)
State 53 file tcasv1a.c line 133 function c::alt_sep_test
----------------------------------------------------
  ASTBeg=TRUE
State 54 file tcasv1a.c line 135 function c::alt_sep_test
----------------------------------------------------
  enabled=TRUE
State 55 file tcasv1a.c line 136 function c::alt_sep_test
----------------------------------------------------
  tcas_equipped=FALSE
State 56 file tcasv1a.c line 137 function c::alt_sep_test
----------------------------------------------------
  intent_not_known=FALSE
State 57 file tcasv1a.c line 139 function c::alt_sep_test
----------------------------------------------------
  alt_sep=0
State 59 file tcasv1a.c line 143 function c::alt_sep_test
----------------------------------------------------
  ASTEn=TRUE
State 60
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::upward_preferred_1=0
State 61
----------------------------------------------------
  c::Non_Crossing_Biased_Climb::0::upward_crossing_situation_1=(assignment re-
moved)
```

```
State 62
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::result_1=FALSE
State 63 file tcasv1a.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_1=4
State 65 file tcasv1a.c line 97 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::upward_preferred_1=1
State 67 file tcasv1a.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_2=TRUE
State 70 file tcasv1a.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_3=TRUE
State 73 file tcasv1a.c line 73 function c::ALIM
----------------------------------------------------
  tmp_4=740
State 75 file tcasv1a.c line 100 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::result_1=TRUE
State 82 file tcasv1a.c line 106 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  tmp=TRUE
State 85 file tcasv1a.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_7=TRUE
State 87 file tcasv1a.c line 144 function c::alt_sep_test
----------------------------------------------------
  need_upward_RA=TRUE
State 88
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::upward_preferred_1=0
State 89
----------------------------------------------------
  c::Non_Crossing_Biased_Descend::0::upward_crossing_situation_1=(assignment re-
moved)
State 90
```

```
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::result_1=FALSE
State 91 file tcasv1a.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_9=4
State 93 file tcasv1a.c line 115 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::upward_preferred_1=0
State 101 file tcasv1a.c line 83 function c::Own_Above_Threat
----------------------------------------------------
  tmp_12=FALSE
State 109 file tcasv1a.c line 122 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::result_1=TRUE
State 110 file tcasv1a.c line 124 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  tmp_8=TRUE
State 113 file tcasv1a.c line 83 function c::Own_Above_Threat
----------------------------------------------------
  tmp_15=FALSE
State 115 file tcasv1a.c line 145 function c::alt_sep_test
----------------------------------------------------
  need_downward_RA=FALSE
State 120 file tcasv1a.c line 153 function c::alt_sep_test
----------------------------------------------------
  ASTUpRA=TRUE
State 121 file tcasv1a.c line 154 function c::alt_sep_test
----------------------------------------------------
  alt_sep=1
State 127 file tcasv1a.c line 166 function c::alt_sep_test
----------------------------------------------------
  r=1
State 129 file tcasv1a.c line 229 function c::main
----------------------------------------------------
  PrA=FALSE
State 130 file tcasv1a.c line 230 function c::main
----------------------------------------------------
  PrB=TRUE
```

```
State 131 file tcasv1a.c line 232 function c::main
----------------------------------------------------
  Layer_Positive_RA_Alt_Thresh=740
State 132 file tcasv1a.c line 236 function c::main
----------------------------------------------------
  P1_ACond=FALSE
State 133 file tcasv1a.c line 239 function c::main
----------------------------------------------------
  P1_BCond=TRUE
Failed assertion: assertion file tcasv1a.c line 244 function c::main
Symbols used in assertion:
  P1_BCond=TRUE
  PrB=TRUE
Total number of steps: 72
Sum of values: 171626
Writing counterexample file tcasv1.ce...
VERIFICATION FAILED
Runtime SAT: 0.538s
1.080u 0.000s 0:01.08 100.0%        0+0k 0+0io 1142pf+0w
```

Appendix E

TCAS Version #1 Second Explanation

```
Parsing tcasv1a.c
Converting
Checking tcasv1a
Starting Bounded Model Checking
Reading counterexample from file tcasv1a.ce...
...
size of program expression: 165 assignments
Generated 41 claims, 38 remaining
Passing to decision procedure
Running decision procedure
Solving with PBS - Pseudo Boolean/CNF Solver and Optimizer
8267 variables, 26535 clauses
PBS checker: system is SATISFIABLE (distance 25)
Counterexample:
Initial State
----------------------------------------------------
  ASTBeg=FALSE
  ASTDownRA=FALSE
  ASTEn=FALSE
  ASTUnresRA=FALSE
  ASTUpRA=FALSE
  Alt_Layer_Value=0
```

```
Climb_Inhibit=FALSE
Cur_Vertical_Sep=0
Down_Separation=0
High_Confidence=FALSE
Non_Crossing_Biased_Climb::0::result_1=FALSE
Non_Crossing_Biased_Climb::0::upward_preferred_1=0
Non_Crossing_Biased_Descend::0::result_1=FALSE
Non_Crossing_Biased_Descend::0::upward_preferred_1=0
Other_Capability=0
Other_RAC=0
Other_Tracked_Alt=0
Own_Tracked_Alt=0
Own_Tracked_Alt_Rate=0
P1_ACond=FALSE
P1_BCond=FALSE
P2_ACond=FALSE
P3_BCond=FALSE
P5_ACond=FALSE
Positive_RA_Alt_Thresh= 0, 0, 0, 0
PrA=FALSE
PrB=FALSE
Two_of_Three_Reports_Valid=FALSE
Up_Separation=0
need_downward_RA=FALSE
need_upward_RA=FALSE
Input_Alt_Layer_Value=1
Input_Climb_Inhibit=TRUE
Input_Cur_Vertical_Sep=30510
Input_Down_Separation=504
Input_High_Confidence=TRUE
Input_Other_Capability=2
Input_Other_RAC=0
Input_Other_Tracked_Alt=59429
Input_Own_Tracked_Alt=22708
Input_Own_Tracked_Alt_Rate=2209
Input_Two_of_Three_Reports_Valid=FALSE
Input_Up_Separation=451
tmp=FALSE
```

```
  tmp_1=0
  tmp_10=FALSE
  tmp_11=0
  tmp_12=FALSE
  tmp_13=FALSE
  tmp_14=1073741824
  tmp_15=FALSE
  tmp_2=FALSE
  tmp_3=FALSE
  tmp_4=0
  tmp_5=FALSE
  tmp_6=0
  tmp_7=TRUE
  tmp_8=FALSE
  tmp_9=0
State 16 file tcasv1a.c line 207 function c::main
----------------------------------------------------
  Cur_Vertical_Sep=30510
State 17 file tcasv1a.c line 208 function c::main
----------------------------------------------------
  High_Confidence=TRUE
State 18 file tcasv1a.c line 209 function c::main
----------------------------------------------------
  Two_of_Three_Reports_Valid=FALSE
State 19 file tcasv1a.c line 210 function c::main
----------------------------------------------------
  Own_Tracked_Alt=22708
State 20 file tcasv1a.c line 211 function c::main
----------------------------------------------------
  Own_Tracked_Alt_Rate=0
State 21 file tcasv1a.c line 212 function c::main
----------------------------------------------------
  Other_Tracked_Alt=59429
State 22 file tcasv1a.c line 213 function c::main
----------------------------------------------------
  Alt_Layer_Value=1
State 23 file tcasv1a.c line 214 function c::main
----------------------------------------------------
```

```
  Up_Separation=451
State 24 file tcasv1a.c line 215 function c::main
----------------------------------------------------
  Down_Separation=504
State 25 file tcasv1a.c line 216 function c::main
----------------------------------------------------
  Other_RAC=0
State 26 file tcasv1a.c line 217 function c::main
----------------------------------------------------
  Other_Capability=2
State 27 file tcasv1a.c line 218 function c::main
----------------------------------------------------
  Climb_Inhibit=TRUE
State 29 file tcasv1a.c line 65 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 0, 0, 0
State 30 file tcasv1a.c line 66 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 0, 0
State 31 file tcasv1a.c line 67 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 0
State 32 file tcasv1a.c line 68 function c::initialize
----------------------------------------------------
  Positive_RA_Alt_Thresh= 400, 500, 640, 740
State 33 file tcasv1a.c line 133 function c::alt_sep_test
----------------------------------------------------
  ASTBeg=TRUE
State 34 file tcasv1a.c line 135 function c::alt_sep_test
----------------------------------------------------
  enabled=TRUE
State 35 file tcasv1a.c line 136 function c::alt_sep_test
----------------------------------------------------
  tcas_equipped=FALSE
State 36 file tcasv1a.c line 137 function c::alt_sep_test
----------------------------------------------------
  intent_not_known=FALSE
State 37 file tcasv1a.c line 139 function c::alt_sep_test
```

```
----------------------------------------------------
  alt_sep=0
State 39 file tcasv1a.c line 143 function c::alt_sep_test
----------------------------------------------------
  ASTEn=TRUE
State 40 file tcasv1a.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_1=551
State 42 file tcasv1a.c line 97 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::upward_preferred_1=1
State 44 file tcasv1a.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_2=TRUE
State 47 file tcasv1a.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_3=TRUE
State 50 file tcasv1a.c line 73 function c::ALIM
----------------------------------------------------
  tmp_4=500
State 52 file tcasv1a.c line 100 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  Non_Crossing_Biased_Climb::0::result_1=FALSE
State 59 file tcasv1a.c line 106 function c::Non_Crossing_Biased_Climb
----------------------------------------------------
  tmp=FALSE
State 64 file tcasv1a.c line 144 function c::alt_sep_test
----------------------------------------------------
  need_upward_RA=FALSE
State 65 file tcasv1a.c line 88 function c::Inhibit_Biased_Climb
----------------------------------------------------
  tmp_9=551
State 67 file tcasv1a.c line 115 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::upward_preferred_1=1
State 69 file tcasv1a.c line 78 function c::Own_Below_Threat
----------------------------------------------------
  tmp_10=TRUE
```

```
State 72 file tcasv1a.c line 73 function c::ALIM
----------------------------------------------------
  tmp_11=500
State 74 file tcasv1a.c line 118 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  Non_Crossing_Biased_Descend::0::result_1=TRUE
State 84 file tcasv1a.c line 124 function c::Non_Crossing_Biased_Descend
----------------------------------------------------
  tmp_8=TRUE
State 87 file tcasv1a.c line 83 function c::Own_Above_Threat
----------------------------------------------------
  tmp_15=FALSE
State 89 file tcasv1a.c line 145 function c::alt_sep_test
----------------------------------------------------
  need_downward_RA=FALSE
State 99 file tcasv1a.c line 161 function c::alt_sep_test
----------------------------------------------------
  ASTUnresRA=TRUE
State 100 file tcasv1a.c line 162 function c::alt_sep_test
----------------------------------------------------
  alt_sep=0
State 101 file tcasv1a.c line 166 function c::alt_sep_test
----------------------------------------------------
  r=0
State 103 file tcasv1a.c line 229 function c::main
----------------------------------------------------
  PrA=FALSE
State 104 file tcasv1a.c line 230 function c::main
----------------------------------------------------
  PrB=FALSE
State 105 file tcasv1a.c line 232 function c::main
----------------------------------------------------
  Layer_Positive_RA_Alt_Thresh=500
State 106 file tcasv1a.c line 236 function c::main
----------------------------------------------------
  P1_ACond=FALSE
State 107 file tcasv1a.c line 239 function c::main
----------------------------------------------------
```

```
  P1_BCond=TRUE
State 111 file tcasv1a.c line 250 function c::main
----------------------------------------------------
  P2_ACond=FALSE
State 113 file tcasv1a.c line 257 function c::main
----------------------------------------------------
  P3_BCond=FALSE
State 115 file tcasv1a.c line 262 function c::main
----------------------------------------------------
  P5_ACond=FALSE
Error explanation deltas:
Value changed:  Input_Down_Separation_1#0 from 500 to 504
Value changed:  Down_Separation#1 from 500 to 504
                file tcasv1a.c line 215 function c::main
Value changed:  result_1#1 from TRUE to FALSE
                file tcasv1a.c line 100 function c::Non_Crossing_Biased_Climb
Value changed:  result_1#3 from TRUE to FALSE
Value changed:  tmp#1 from TRUE to FALSE
                file tcasv1a.c line 106 function c::Non_Crossing_Biased_Climb
Guard changed:  \guard#1 && tmp#1 (\guard#7) was TRUE
                file tcasv1a.c line 144 function c::alt_sep_test
Value changed:  tmp_7#0 from FALSE to TRUE
Value changed:  need_upward_RA_1#1 from TRUE to FALSE
                file tcasv1a.c line 144 function c::alt_sep_test
Guard changed:  \guard#15 && need_upward_RA_1#1 (\guard#16) was TRUE
                file tcasv1a.c line 152 function c::alt_sep_test
Guard changed:  \guard#15 && !need_upward_RA_1#1 (\guard#17) was FALSE
                file tcasv1a.c line 152 function c::alt_sep_test
Guard changed:  \guard#17 && !need_downward_RA_1#1 (\guard#19) was FALSE
                file tcasv1a.c line 156 function c::alt_sep_test
Value changed:  ASTUnresRA#3 from FALSE to TRUE
Value changed:  ASTUpRA#2 from TRUE to FALSE
Value changed:  alt_sep_1#7 from 1 to 0
Value changed:  ASTUnresRA#4 from FALSE to TRUE
Value changed:  ASTUpRA#3 from TRUE to FALSE
Value changed:  alt_sep_1#8 from 1 to 0
Value changed:  ASTUnresRA#5 from FALSE to TRUE
Value changed:  ASTUpRA#4 from TRUE to FALSE
```

```
Value changed:  result_1#4 from TRUE to FALSE
Value changed:  alt_sep_1#9 from 1 to 0
Value changed:  need_upward_RA_1#2 from TRUE to FALSE
Value changed:  tmp#2 from TRUE to FALSE
Value changed:  r_1#1 from 1 to 0
                file tcasv1a.c line 166 function c::alt_sep_test
Value changed:  PrB#1 from TRUE to FALSE
                file tcasv1a.c line 230 function c::main
Minimizing deltas
Running decision procedure
Solving with PBS - Pseudo Boolean/CNF Solver and Optimizer
4694 variables, 14618 clauses
PBS checker: system is SATISFIABLE (distance 14)
Deltas after minimization:
Value changed:  Input_Down_Separation_1#0 from 500 to 504
Value changed:  Down_Separation#1 from 500 to 504
                file tcasv1a.c line 215 function c::main
Value changed:  result_1#1 from TRUE to FALSE
                file tcasv1a.c line 100 function c::Non_Crossing_Biased_Climb
Value changed:  result_1#3 from TRUE to FALSE
Value changed:  tmp#1 from TRUE to FALSE
                file tcasv1a.c line 106 function c::Non_Crossing_Biased_Climb
Guard changed:  \guard#1 && tmp#1 (\guard#7) was TRUE
                file tcasv1a.c line 144 function c::alt_sep_test
Value changed:  need_upward_RA_1#1 from TRUE to FALSE
                file tcasv1a.c line 144 function c::alt_sep_test
Guard changed:  \guard#15 && need_upward_RA_1#1 (\guard#16) was TRUE
                file tcasv1a.c line 152 function c::alt_sep_test
Guard changed:  \guard#15 && !need_upward_RA_1#1 (\guard#17) was FALSE
                file tcasv1a.c line 152 function c::alt_sep_test
Guard changed:  \guard#17 && !need_downward_RA_1#1 (\guard#19) was FALSE
                file tcasv1a.c line 156 function c::alt_sep_test
Value changed:  ASTUpRA#2 from TRUE to FALSE
Value changed:  ASTUpRA#3 from TRUE to FALSE
Value changed:  ASTUpRA#4 from TRUE to FALSE
Value changed:  PrB#1 from TRUE to FALSE
                file tcasv1a.c line 230 function c::main
CLOSEST SUCCESSFUL PATH FOUND
```

```
4.070u 0.100s 0:04.17 100.0%        0+0k 0+0io 1831pf+0w
```

Appendix F

μC/OS-II Counterexample

```
Parsing microcos.c
Converting
Checking microcos
Starting Bounded Model Checking
...
Unwinding loop iteration 1
size of program expression: 2578 assignments
Generated 95 claims, 33 remaining
Passing to decision procedure
...
Running decision procedure
Solving with ZChaff version ZChaff 2003.6.16
235263 variables, 566940 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
Counterexample:
Initial State
----------------------------------------------------
  LOCK=0
  OSCPUUsage=0 (00000000)
  OSCtxSwCtr=0
  OSEventFreeList=NULL
  OSEventTO::pevent_1=NULL
  OSEventTO::pevent_2=NULL
  OSEventTO::pevent_3=NULL
```

```
bitx_1=0 (00000000)
bitx_2=0 (00000000)
bitx_3=0 (00000000)
bitx_4=0 (00000000)
bity_1=0 (00000000)
bity_2=0 (00000000)
bity_3=0 (00000000)
bity_4=0 (00000000)
prio_1=0 (00000000)
prio_2=0 (00000000)
prio_3=0 (00000000)
prio_4=0 (00000000)
ptcb_1=NULL
ptcb_2=NULL
ptcb_3=NULL
ptcb_4=NULL
x_1=0 (00000000)
x_2=0 (00000000)
x_3=0 (00000000)
x_4=0 (00000000)
OSEventTaskRdy::0::y_1=0 (00000000)
OSEventTaskRdy::0::y_2=0 (00000000)
OSEventTaskRdy::0::y_3=0 (00000000)
OSEventTaskRdy::0::y_4=0 (00000000)
OSEventTaskRdy::msg_1=NULL
msg_2=NULL
msg_3=NULL
msg_4=NULL
msk_1=1 (00000001)
msk_2=2 (00000010)
msk_3=4 (00000100)
msk_4=4 (00000100)
OSEventTaskRdy::pevent_1=NULL
OSEventTaskRdy::pevent_2=NULL
OSEventTaskRdy::pevent_3=NULL
pevent_4=NULL
OSEventTaskWait::pevent_1=NULL
OSEventTaskWait::pevent_2=NULL
```

```
OSEventTaskWait::pevent_3=NULL
OSEventTbl=  NULL,  0, 0 , 0, 0, 0 ,  NULL,  0, 0 , 0, 0, 0
OSIdleCtr=0
OSIdleCtrMax=0
OSIdleCtrRun=0
OSIntNesting=0 (00000000)
OSLockNesting=0 (00000000)
OSMapTbl= 1, 2, 4, 8, 16, 32, 64, 128
OSMboxAccept::0::msg_1=NULL
OSMboxAccept::pevent_1=NULL
OSMboxCreate::msg_1=NULL
OSMboxPend::0::msg_1=NULL
OSMboxPend::err_1=&error_1
OSMboxPend::pevent_1=NULL
OSMboxPend::timeout_1=10
OSMboxPost::msg_1=NULL
OSMboxPost::pevent_1=NULL
OSMboxQuery::0::i_1=0 (00000000)
OSMboxQuery::0::pdest_1=NULL
OSMboxQuery::0::psrc_1=NULL
OSMboxQuery::0::tmp_1=NULL
OSMboxQuery::0::tmp___0_1=NULL
OSMboxQuery::pdata_1=&mboxData_1
OSMboxQuery::pevent_1=NULL
OSPrioCur=0 (00000000)
OSPrioHighRdy=0 (00000000)
OSQAccept::0::msg_1=NULL
OSQAccept::0::pq_1=NULL
OSQAccept::0::tmp_1=NULL
OSQAccept::pevent_1=NULL
size=10
start=NULL
OSQFlush::0::pq_1=NULL
OSQFlush::pevent_1=NULL
OSQPend::0::msg_1=NULL
OSQPend::0::pq_1=NULL
OSQPend::0::tmp_1=NULL
OSQPend::0::tmp___0_1=NULL
```

```
OSQPend::err_1=&error_1
OSQPend::pevent_1=NULL
OSQPend::timeout_1=10
OSQPost::0::pq_1=NULL
OSQPost::0::tmp_1=NULL
OSQPost::msg_1=NULL
OSQPost::pevent_1=NULL
OSQPostFront::0::pq_1=NULL
OSQPostFront::msg_1=NULL
OSQPostFront::pevent_1=NULL
OSQQuery::0::i_1=0 (00000000)
OSQQuery::0::pdest_1=NULL
OSQQuery::0::pq_1=NULL
OSQQuery::0::psrc_1=NULL
OSQQuery::0::tmp_1=NULL
OSQQuery::0::tmp___0_1=NULL
OSQQuery::pdata_1=&qdata_1
OSQQuery::pevent_1=NULL
OSRdyGrp=0 (00000000)
OSRdyTbl= 0, 0
OSRunning=0 (00000000)
OSSched::0::y_1=0 (00000000)
OSSched::0::y_2=0 (00000000)
OSSched::0::y_3=0 (00000000)
OSSched::0::y_4=0 (00000000)
y_5=0 (00000000)
y_6=0 (00000000)
y_7=0 (00000000)
0::cnt_1=0
OSSemAccept::pevent_1=NULL
OSSemCreate::cnt_1=1
OSSemPend::err_1=&error_1
OSSemPend::pevent_1=NULL
OSSemPend::timeout_1=10
OSSemPost::pevent_1=NULL
OSSemQuery::0::i_1=0 (00000000)
OSSemQuery::0::pdest_1=NULL
OSSemQuery::0::psrc_1=NULL
```

```
OSSemQuery::0::tmp_1=NULL
OSSemQuery::0::tmp___0_1=NULL
OSSemQuery::pdata_1=&semData_1
OSSemQuery::pevent_1=NULL
OSStatRdy=0 (00000000)
OSTCBCur=NULL
OSTCBFreeList=NULL
OSTCBHighRdy=NULL
OSTCBList=NULL
OSTCBPrioTbl= NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL
OSTaskCtr=0 (00000000)
OSTime=0
OSUnMapTbl= 0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0,
1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3,
0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1,
0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0,
1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 7, 0, 1, 0, 2,
0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0,
2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1,
0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 6, 0, 1, 0, 2, 0, 1, 0, 3, 0,
1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5,
0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0,
3, 0, 1, 0, 2, 0, 1, 0
mboxData= NULL,  0, 255 , 0
qdata= NULL, 0, 0,  0, 255 , 0
semData= 0,  0, 255 , 0
OSIntExitY=0 (00000000)
OSMemFreeList=NULL
OSMemTbl=  NULL, NULL, 0, 0, 0 ,  NULL, NULL, 0, 0, 0
OSQFreeList=NULL
OSQTbl=  NULL, NULL, NULL, NULL, NULL, 0, 0 ,  NULL, NULL, NULL,
NULL, NULL, 0, 0   OSTCBTbl=  NULL, NULL, NULL, 0, 0, 0, NULL,
NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0,
0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL,
NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0
,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0,
```

```
0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL,
0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL,
NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0,
NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL,
NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,
NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0,
0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0,
0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL,
NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL,
NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0,
0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0
OSTaskIdleStk= 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
OSTaskStatStk= 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
  tmp=0
State 1 file microcos.c line 2940 function c::main
----------------------------------------------------
  error=0 (00000000)
State 4 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 5 file microcos.c line 1898 function c::OSSemCreate
----------------------------------------------------
  OSSemCreate::0::pevent_1=NULL
State 9 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 11 file microcos.c line 1916 function c::OSSemCreate
----------------------------------------------------
  sem=NULL
State 15 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
```

```
State 16 file microcos.c line 1013 function c::OSMboxCreate
----------------------------------------------------
  OSMboxCreate::0::pevent_1=NULL
State 20 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 22 file microcos.c line 1031 function c::OSMboxCreate
----------------------------------------------------
  mbox=NULL
State 26 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 27 file microcos.c line 1471 function c::OSQCreate
----------------------------------------------------
  OSQCreate::0::pevent_1=NULL
State 31 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 33 file microcos.c line 1526 function c::OSQCreate
----------------------------------------------------
  queue=NULL
State 36 file microcos.c line 2946 function c::main
----------------------------------------------------
  choice=0
State 41 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 45 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 46 file microcos.c line 1931 function c::OSSemPend
----------------------------------------------------
  error=1 (00000001)
State 54 file microcos.c line 1954 function c::OSSemPend
----------------------------------------------------
  invalid_object_4= INVALID-255 + -1, INVALID-255 + -1, INVALID-255
  + -1, 4294967295, 4294967295, 4294967295, INVALID-255 + -1,
  INVALID-255 + -1, INVALID-255 + -1, INVALID-255 + -1, 4294967295, 1,
```

```
  255, 255, 255, 255, 255, 255
State 55 file microcos.c line 1956 function c::OSSemPend
----------------------------------------------------
  invalid_object_6= INVALID-255 + -1, INVALID-255 + -1, INVALID-255
  + -1, 4294967295, 4294967295, 4294967295, INVALID-255 + -1,
  INVALID-255 + -1, INVALID-255 + -1, INVALID-255 + -1, 10, 255, 255,
  255, 255, 255, 255, 255
State 57 file microcos.c line 419 function c::OSEventTaskWait
----------------------------------------------------
  invalid_object_7= INVALID-255 + -1, INVALID-255 + -1, INVALID-255
  + -1, 4294967295, 4294967295, 4294967295, INVALID-255 + -1,
  INVALID-255 + -1, NULL, INVALID-255 + -1, 4294967295, 255, 255, 255,
  255, 255, 255, 255
State 58 file microcos.c line 422 function c::OSEventTaskWait
----------------------------------------------------
  OSRdyTbl= 0, 0
State 61 file microcos.c line 430 function c::OSEventTaskWait
----------------------------------------------------
  invalid_object_13= INVALID-255 + -1,  0, 0 , 4294967295, 255, 255
State 62 file microcos.c line 432 function c::OSEventTaskWait
----------------------------------------------------
  invalid_object_18= INVALID-255 + -1,  255, 255 , 4294967295, 255, 0
Failed assertion: assertion file microcos.c line 582 function c::OS_EXIT_CRITICAL
Other Variables
----------------------------------------------------
(omitted in interests of space:  invalid objects)
...
Writing counterexample file ucos.ce...
Runtime SAT: 10.79s
VERIFICATION FAILED
```

Appendix G

μC/OS-II Explanation

```
Parsing microcos.c
Converting
Checking microcos
Starting Bounded Model Checking
Reading counterexample from file ucos.ce...
...
Unwinding loop iteration 1
size of program expression: 2578 assignments
Generated 95 claims, 33 remaining
Passing to decision procedure
...
Running decision procedure
Solving with PBS - Pseudo Boolean/CNF Solver and Optimizer
236064 variables, 568886 clauses
PBS checker: system is SATISFIABLE (distance 3)
Counterexample:
Initial State
----------------------------------------------------
  LOCK=0
  OSCPUUsage=0 (00000000)
  OSCtxSwCtr=0
  OSEventFreeList=NULL
  OSEventTO::pevent_1=NULL
  OSEventTO::pevent_2=NULL
```

```
OSEventTO::pevent_3=NULL
bitx_1=255 (11111111)
bitx_2=14 (00001110)
bitx_3=176 (10110000)
bitx_4=97 (01100001)
bity_1=188 (10111100)
bity_2=53 (00110101)
bity_3=202 (11001010)
bity_4=188 (10111100)
prio_1=42 (00101010)
prio_2=11 (00001011)
prio_3=189 (10111101)
prio_4=252 (11111100)
ptcb_1=INVALID-66 + -514965513
ptcb_2=INVALID-246 + -141955093
ptcb_3=INVALID-79 + -526139393
ptcb_4=INVALID-31 + -741503359
x_1=3 (00000011)
x_2=121 (01111001)
x_3=64 (01000000)
x_4=53 (00110101)
OSEventTaskRdy::0::y_1=189 (10111101)
OSEventTaskRdy::0::y_2=131 (10000011)
OSEventTaskRdy::0::y_3=211 (11010011)
OSEventTaskRdy::0::y_4=184 (10111000)
OSEventTaskRdy::msg_1=NULL
msg_2=NULL
msg_3=NULL
msg_4=NULL
msk_1=1 (00000001)
msk_2=2 (00000010)
msk_3=4 (00000100)
msk_4=4 (00000100)
OSEventTaskRdy::pevent_1=NULL
OSEventTaskRdy::pevent_2=NULL
OSEventTaskRdy::pevent_3=NULL
pevent_4=NULL
OSEventTaskWait::pevent_1=NULL
```

```
OSEventTaskWait::pevent_2=NULL
OSEventTaskWait::pevent_3=NULL
OSEventTbl=  NULL,  0, 0 , 0, 0, 0 ,  NULL,  0, 0 , 0, 0, 0
OSIdleCtr=0
OSIdleCtrMax=0
OSIdleCtrRun=0
OSIntNesting=0 (00000000)
OSLockNesting=0 (00000000)
OSMapTbl= 1, 2, 4, 8, 16, 32, 64, 128
OSMboxAccept::0::msg_1=INVALID-253 + -285382688
OSMboxAccept::pevent_1=NULL
OSMboxCreate::msg_1=NULL
OSMboxPend::0::msg_1=INVALID-98 + -215585008
OSMboxPend::err_1=&error_1
OSMboxPend::pevent_1=NULL
OSMboxPend::timeout_1=10
OSMboxPost::msg_1=NULL
OSMboxPost::pevent_1=NULL
OSMboxQuery::0::i_1=211 (11010011)
OSMboxQuery::0::pdest_1=INVALID-81 + -1308916935
OSMboxQuery::0::psrc_1=INVALID-225 + 383761919
OSMboxQuery::0::tmp_1=INVALID-147 + -139510865
OSMboxQuery::0::tmp___0_1=INVALID-246 + -572272207
OSMboxQuery::pdata_1=&mboxData_1
OSMboxQuery::pevent_1=NULL
OSPrioCur=0 (00000000)
OSPrioHighRdy=0 (00000000)
OSQAccept::0::msg_1=INVALID-138 + 1890745416
OSQAccept::0::pq_1=INVALID-111 + 171729768
OSQAccept::0::tmp_1=INVALID-176 + 2060098078
OSQAccept::pevent_1=NULL
size=10
start=NULL
OSQFlush::0::pq_1=INVALID-190 + -117380951
OSQFlush::pevent_1=NULL
OSQPend::0::msg_1=INVALID-230 + -73863301
OSQPend::0::pq_1=INVALID-239 + 2130176765
OSQPend::0::tmp_1=INVALID-58 + -804330987
```

```
OSQPend::0::tmp___0_1=INVALID-126 + 188701213
OSQPend::err_1=&error_1
OSQPend::pevent_1=NULL
OSQPend::timeout_1=10
OSQPost::0::pq_1=INVALID-243 + -676386849
OSQPost::0::tmp_1=INVALID-209 + -92675154
OSQPost::msg_1=NULL
OSQPost::pevent_1=NULL
OSQPostFront::0::pq_1=INVALID-240 + 1036066654
OSQPostFront::msg_1=NULL
OSQPostFront::pevent_1=NULL
OSQQuery::0::i_1=127 (01111111)
OSQQuery::0::pdest_1=INVALID-147 + 559185748
OSQQuery::0::pq_1=INVALID-80 + -1795095728
OSQQuery::0::psrc_1=INVALID-254 + -2130439717
OSQQuery::0::tmp_1=INVALID-64 + 166475285
OSQQuery::0::tmp___0_1=INVALID-159 + -1244365570
OSQQuery::pdata_1=&qdata_1
OSQQuery::pevent_1=NULL
OSRdyGrp=0 (00000000)
OSRdyTbl= 0, 0
OSRunning=0 (00000000)
OSSched::0::y_1=90 (01011010)
OSSched::0::y_2=243 (11110011)
OSSched::0::y_3=249 (11111001)
OSSched::0::y_4=178 (10110010)
y_5=233 (11101001)
y_6=30 (00011110)
y_7=166 (10100110)
0::cnt_1=18350168 (00000001000110000000000001011000)
OSSemAccept::pevent_1=NULL
OSSemCreate::cnt_1=1
OSSemPend::err_1=&error_1
OSSemPend::pevent_1=NULL
OSSemPend::timeout_1=10
OSSemPost::pevent_1=NULL
OSSemQuery::0::i_1=1 (00000001)
OSSemQuery::0::pdest_1=INVALID-135 + 29246984
```

```
OSSemQuery::0::psrc_1=INVALID-17 + 1990264193
OSSemQuery::0::tmp_1=INVALID-154 + 1447418904
OSSemQuery::0::tmp___0_1=INVALID-202 + -2098456782
OSSemQuery::pdata_1=&semData_1
OSSemQuery::pevent_1=NULL
OSStatRdy=0 (00000000)
OSTCBCur=NULL
OSTCBFreeList=NULL
OSTCBHighRdy=NULL
OSTCBList=NULL
OSTCBPrioTbl= NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL
OSTaskCtr=0 (00000000)
OSTime=0
OSUnMapTbl= 0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0,
1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3,
0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1,
0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0,
1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 7, 0, 1, 0, 2,
0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0,
2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1,
0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 6, 0, 1, 0, 2, 0, 1, 0, 3, 0,
1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5,
0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0,
3, 0, 1, 0, 2, 0, 1, 0
mboxData= INVALID-249 + -929194441,  60, 255 , 75
qdata= INVALID-197 + -537327008, 936476665, 168413167,  43, 255 , 66
semData= 1094223367,  47, 255 , 129
OSIntExitY=0 (00000000)
OSMemFreeList=NULL
OSMemTbl=  NULL, NULL, 0, 0, 0 ,  NULL, NULL, 0, 0, 0
OSQFreeList=NULL
OSQTbl=  NULL, NULL, NULL, NULL, NULL, 0, 0 ,  NULL, NULL, NULL,
NULL, NULL, 0, 0
OSTCBTbl=  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0,
0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL,
NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0,
```

```
NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL,
NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,
NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0,
0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0,
0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL,
NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL,
NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0,
0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0 ,  NULL,
NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0
,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL, 0, 0, 0, 0,
0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL,
0, 0, 0, 0, 0, 0, 0, 0 ,  NULL, NULL, NULL, 0, 0, 0, NULL, NULL,
NULL, NULL, 0, 0, 0, 0, 0, 0, 0, 0
OSTaskIdleStk= 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
OSTaskStatStk= 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
  tmp=820047884 (00110000111000001111000000001100)
State 1 file microcos.c line 2940 function c::main
----------------------------------------------------
  error=0 (00000000)
State 4 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 5 file microcos.c line 1898 function c::OSSemCreate
----------------------------------------------------
  OSSemCreate::0::pevent_1=NULL
State 9 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 11 file microcos.c line 1916 function c::OSSemCreate
----------------------------------------------------
  sem=NULL
State 15 file microcos.c line 578 function c::OS_ENTER_CRITICAL
```

```
----------------------------------------------------
  LOCK=1
State 16 file microcos.c line 1013 function c::OSMboxCreate
----------------------------------------------------
  OSMboxCreate::0::pevent_1=NULL
State 20 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 22 file microcos.c line 1031 function c::OSMboxCreate
----------------------------------------------------
  mbox=NULL
State 26 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 27 file microcos.c line 1471 function c::OSQCreate
----------------------------------------------------
  OSQCreate::0::pevent_1=NULL
State 31 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 33 file microcos.c line 1526 function c::OSQCreate
----------------------------------------------------
  queue=NULL
State 36 file microcos.c line 2946 function c::main
----------------------------------------------------
  choice=0
State 41 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 54 file microcos.c line 1954 function c::OSSemPend
----------------------------------------------------
  invalid_object_4= INVALID-255 + -1, INVALID-255 + -1, INVALID-255
  + -1, 4294967295, 4294967295, 4294967295, INVALID-255 + -1,
  INVALID-255 + -1, INVALID-255 + -1, INVALID-255 + -1, 4294967295,
  161, 255, 255, 255, 255, 255, 255
State 55 file microcos.c line 1956 function c::OSSemPend
----------------------------------------------------
  invalid_object_6= INVALID-255 + -1, INVALID-255 + -1, INVALID-255
```

```
  + -1, 4294967295, 4294967295, 4294967295, INVALID-255 + -1,
  INVALID-255 + -1, INVALID-255 + -1, INVALID-255 + -1, 10, 255, 255,
  255, 255, 255, 255, 255
State 57 file microcos.c line 419 function c::OSEventTaskWait
----------------------------------------------------
  invalid_object_7= INVALID-255 + -1, INVALID-255 + -1, INVALID-255
  + -1, 4294967295, 4294967295, 4294967295, INVALID-255 + -1,
  INVALID-255 + -1, NULL, INVALID-255 + -1, 4294967295, 255, 255, 255,
  255, 255, 255, 255
State 58 file microcos.c line 422 function c::OSEventTaskWait
----------------------------------------------------
  OSRdyTbl= 0, 0
State 61 file microcos.c line 430 function c::OSEventTaskWait
----------------------------------------------------
  invalid_object_13= INVALID-255 + -1,  8, 30 , 4294967295, 255, 255
State 62 file microcos.c line 432 function c::OSEventTaskWait
----------------------------------------------------
  invalid_object_18= INVALID-255 + -1,  255, 255 , 4294967295, 255, 166
State 65 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 69 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 71 file microcos.c line 644 function c::OSSched
----------------------------------------------------
  OSSched::0::y_1=0 (00000000)
State 72 file microcos.c line 646 function c::OSSched
----------------------------------------------------
  OSPrioHighRdy=0 (00000000)
State 79 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 82 file microcos.c line 578 function c::OS_ENTER_CRITICAL
----------------------------------------------------
  LOCK=1
State 85 file microcos.c line 444 function c::OSEventTO
----------------------------------------------------
```

```
  invalid_object_22= INVALID-255 + -1,  153, 1 , 4294967295, 255, 255
State 87 file microcos.c line 448 function c::OSEventTO
----------------------------------------------------
  invalid_object_29= INVALID-255 + -1,  255, 255 , 4294967295, 255, 175
State 88 file microcos.c line 451 function c::OSEventTO
----------------------------------------------------
  invalid_object_32= INVALID-255 + -1, INVALID-255 + -1,
  INVALID-255 + -1, 4294967295, 4294967295, 4294967295, INVALID-255 +
  -1, INVALID-255 + -1, INVALID-255 + -1, INVALID-255 + -1,
  4294967295, 0, 255, 255, 255, 255, 255, 255
State 89 file microcos.c line 453 function c::OSEventTO
----------------------------------------------------
  invalid_object_33= INVALID-255 + -1, INVALID-255 + -1,
  INVALID-255 + -1, 4294967295, 4294967295, 4294967295, INVALID-255 +
  -1, INVALID-255 + -1, NULL, INVALID-255 + -1, 4294967295, 255, 255,
  255, 255, 255, 255, 255
State 92 file microcos.c line 583 function c::OS_EXIT_CRITICAL
----------------------------------------------------
  LOCK=0
State 93 file microcos.c line 1972 function c::OSSemPend
----------------------------------------------------
  error=10 (00001010)
Other Variables
----------------------------------------------------
(omitted in interests of space:  invalid objects)
...
Error explanation deltas:
Guard changed:  !(invalid_object#0.OSEventType == 3) &&
                \guard#1 (\guard#2) was TRUE
                file microcos.c line 1927 function c::OSSemPend
Value changed:  LOCK#9 from 0 to 1
Value changed:  error_1#3 from 1 to 0
CLOSEST SUCCESSFUL PATH FOUND
```