Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Error Explanation with Distance Metrics

Alex Groce[1], **Sagar Chaki**[1], **Daniel Kroening**[2], **Ofer Strichman**[3]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA, e-mail: agroce,chaki@cs.cmu.edu
[2] ETH Zurich, Zurich, Switzerland, e-mail: kroening@cs.cmu.edu
[3] Technion, Haifa, Israel, e-mail: ofers@ie.technion.ac.il

The date of receipt and acceptance will be inserted by the editor

Abstract. In the event that a system does not satisfy a specification, a model checker will typically automatically produce a counterexample trace that shows a particular instance of the undesirable behavior. Unfortunately, the important steps that follow the discovery of a counterexample are generally not automated. The user must first decide if the counterexample shows genuinely erroneous behavior or is an artifact of improper specification or abstraction. In the event that the error is real, there remains the difficult task of understanding the error well enough to isolate and modify the faulty aspects of the system. This paper describes a (semi-)automated approach for assisting users in understanding and isolating errors in ANSI C programs. The approach, derived from David Lewis' counterfactual approach to causality, is based on distance metrics for program executions. Experimental results show that the power of the model checking engine can be used to provide assistance in understanding errors and to isolate faulty portions of the source code.

1 Introduction

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 [9], and locating an error is the most difficult aspect of the debugging process [62]. This paper describes the application of a technology traditionally used for *discovering* errors to the problem of *understanding and isolating* errors.

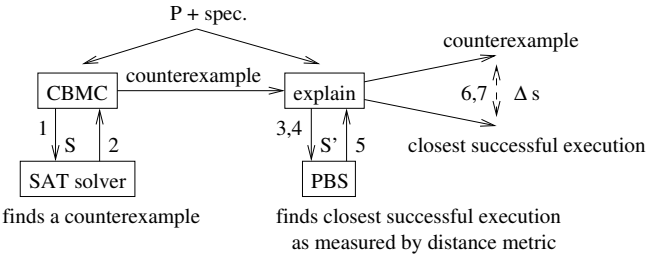

Fig. 1. Explaining an error using distance metrics

Error explanation describes 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. This is a psychological problem, and it is unlikely that formal proof of the superiority of any approach is possible. *Fault localization* is the more specific task of identifying the faulty core of a system, and is suitable for quantitative evaluation.

Model checking [19,49,21] 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 [20] is produced. This paper explains how a model checker can provide error explanation and fault localization information. For a program P, the process (Figure 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 Boolean satisfiability (SAT) problem, S. The satisfying assignments of S are finite *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 [5] 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 6). 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.

There are many possible approaches to error explanation. A basic notion shared by many researchers in this area [10,31,65] and many philosophers [59] is that to explain something 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) [31,52,66].

David Lewis [43] has proposed a theory of causality that provides a 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 logical that successful executions resembling a counterexample can be used to explain an error.

Following Hume [36,37,59] 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*) [44]. 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 [34,40].

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*.

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]:

Definition 1 (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). 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 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] Our causal dependence is actually Lewis' counterfactual dependence.

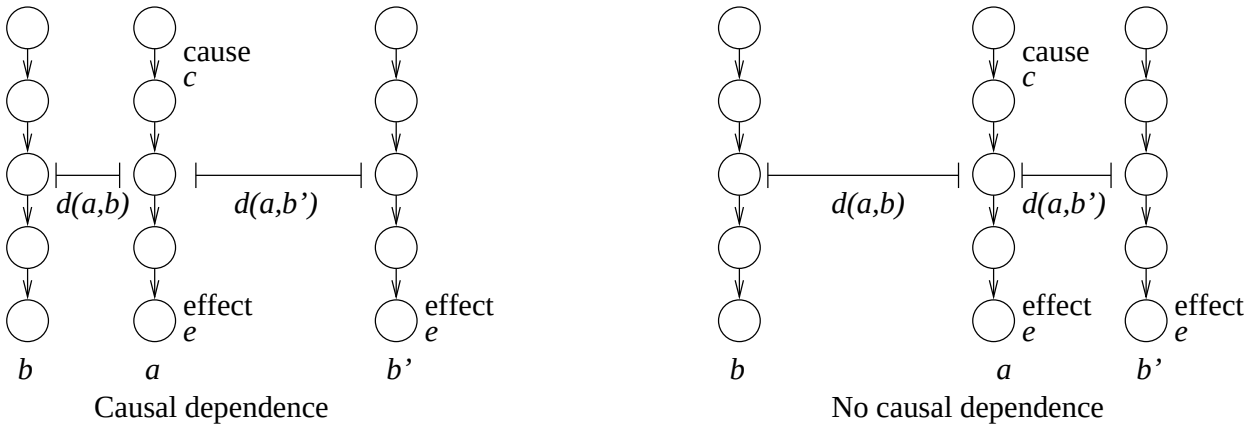

Fig. 2. Causal dependence

This article describes a distance metric that allows determination of causal dependencies and the implementation of that metric in a tool called `explain` [30] that extends CBMC [1], a model checker for programs written in ANSI C. The focus of the paper, 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[2]. 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.

The approach presented in this paper 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 our case studies, this is a result of the structure of the property, and can be fully automated; more generally, however, it is not possible to make use of a fully automated refinement, 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 of our examples required the addition of a non-automatable assumption. See Section 6.1 for the details of this occasional need for additional guidance.

[2] Computing causal dependence using two bounded model checking queries is described elsewhere [29].

The basic approach, presented in Section 4, 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 4.2 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.

2 Related Work

This paper is an extension of the TACAS 2004 paper [28] which originally presented error explanation based on distance metrics: we introduce further case study and experimental results and a new slicing method, shedding light on the need for user-introduced assumptions. The `explain` tool is described in a CAV 2004 paper [30].

Recent work by Chechik, Tan, and others has described proof-like and evidence-based counterexamples [17, 60]. Automatically generating assumptions for verification [23] can also be seen as a kind of error explanation: an assumption describes the conditions under which a system avoids error. These approaches appear to be unlikely to result in *succinct* explanations, as they may encode the complexity of the transition system; one measure of a useful explanation lies in how much it *reduces* the information the user must consider.

Error explanation facilities are now featured in Microsoft's SLAM [11] model checker [10] and NASA's Java PathFinder 2 (JPF) [63] model checker [31]. 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 [38]. Of these, only JPF uses a (weak) notion of distance between traces, and it cannot solve for nearest successful executions.

Sharygina and Peled [57] 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 [16] 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 [16,32] have proposed using these queries to provide feedback in the event that a property does not hold on a model.

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) [48] for explanation [58].

Analyses of causality from the field of artificial intelligence appear to rely on causal theories or more precise logical models of relationships between components than are available in model checking of software systems [27, 45,51], but may be applicable in some cases. The JADE system for diagnosing errors in Java programs makes use of model-based techniques [46]. 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. Wotawa has discussed the relationship between model-based debugging and program slicing [64].

Shapiro [56] introduced a technique for debugging logic programs that relies on interaction with a user as an oracle. Further developments based on this technique have reduced the number of user queries (in part by use of slicing) [41]. 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 [7].

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 [25] use the Daikon invariant detector [26] 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. The JPF implementation of error explanation also computes differences in invariants between sets of successful executions and counterexamples using Daikon. Program spectra [53,33] 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 [39] uses a visualization technique to illuminate (likely) faults statements in programs, as does χSlice [4].

Our work was partly inspired by the success of Andreas Zeller's delta debugging technique [66], 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 [18]. Delta-debugging for deriving cause-effect chains [65] 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. The AskIgor project [2] makes cause-effect chain debugging available via the web.

Renieris and Reiss [52] 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 6.3 makes use of a quantitative method for evaluating fault localization approaches proposed by Renieris and Reiss.

The "slicing" technique presented in Section 5 should be understood in the context of both work on program slicing [61,67,4] and some work on counterexample minimization [50,29]. 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).

Distance metrics can also be used to explain *abstract* counterexamples [15], in which Δs (deltas) are presented in terms of changes to *predicates* on program variables, rather than in terms of concrete values. The methodology presented in this paper is applied to the MAGIC [14] model checker, and the resulting gains in the generality of explanations are described. The distance metric used differs from that presented in this paper in that it does not rely on static single assignment. The resulting metric is possibly more intuitive than the one described in Section 3; however, the use of alignments sometimes results in serious performance problems and occasionally produces less satisfactory explanations.

The `explain` tool has been extended to automatically generate and test hypotheses about causal dependence (as defined in Section 1), in order to provide some of the automatic generalization supplied by abstract explanation [29].

This paper 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 only interested in causal differences to further reduce the amount of information that must be understood by a user.

```
1  int main () {
2    int input1, input2, input3;       //input values
3    int least = input1;               //least#0
4    int most = input1;                //most#0
5    if (most < input2)                //guard#1
6      most = input2;                  //most#1,2
7    if (most < input3)                //guard#2
8      most = input3;                  //most#3,4
9    if (least > input2)               //guard#3
10     most = input2;                  //most#5,6 (ERROR!)
11   if (least > input3)               //guard#4
12     least = input3;                 //least#1,2
13   assert (least <= most);           //specification
14 }
```

Fig. 3. minmax.c

3 Distance Metrics for Program Executions

A distance metric [55] 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 Representing Program Executions

Bounded model checking (BMC) [13] 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 [42] 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 static single assignment (SSA) form [6] and loop unrolling. CBMC and `explain` handle the full set of ANSI C types, structures, and pointer operations including pointer arithmetic. CBMC only checks safety properties, although in principle BMC (and the `explain` approach) can handle full LTL [12][3].

Given the example program minmax.c (Figure 3), CBMC produces the constraints shown in Figure 4 (U is not needed, as minmax.c is loop-free)[4]. 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 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 using ϕ functions, as usual in SSA form: the constraint {-10}, for instance, assigns `most#2` to either `most#1` or `most#0`, depending on the conditional (`\guard#1`) for the assignment to `most#1` (the syntax is that of the C conditional expression). 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 [47], 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 5).

Figure 6 shows the counterexample from Figure 5 in terms of the SSA form assignments (the internal representation used by CBMC for an execution).

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

[3] Explanation for LTL properties has been implemented for error explanation in MAGIC [15].

[4] Output is slightly simplified for readability.

```
{-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
```

Fig. 4. 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

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
```

Fig. 5. Counterexample for minmax.c

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	

Fig. 6. Counterexample values for minmax.ce

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.

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 (and ϕ function) assignments.

3.2 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[5]. 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 2 (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 $b = \{v_0 = val_0^b, v_1 = val_1^b, \ldots, v_n = val_n^b\}$.

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

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, v_2, 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 $\{v_0, v_1, v_2, v_3, \ldots\}$ are {`input1#0`, `input2#0`, `input3#0`, `least#0`, `most#0`, ...}, execution a could be taken to be the counterexample (Figures 5 and 6), and execution b might be the most similar successful execution (see Figures 8 and 9).

This definition is equivalent to the Levenshtein distance [55] if we consider executions as strings where the alphabet elements are assignments and substitution is the only allowed operation[6]. The properties of inequality guarantee that d satisfies the four metric properties.

The metric d differs from the metrics often used in sequence comparison in that it does not make use of a notion of *alignment*. The SSA form based representation encodes an execution as a series of assignments. In contrast, the MAGIC implementation of error explanation represents an execution as a series of *states*, 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

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

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

all data elements when two states are aligned can impose a serious overhead on explanation [15].

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 [15] 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 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.

[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.

3.3 Choosing an Unwinding Depth

The definition of d presented above only considers executions with the same unwinding depth and therefore (due to SSA form) the same variable assignments. 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.

4 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 4.1 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 Section 5.

4.1 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:

```
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)
```

Fig. 7. Δs for minmax.c and the counterexample in Figure 5

$\neg\{1\}$: `least#2 > most#6.`

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

The distance to a given 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 7) by the `explain` tool.

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 to use pseudo-Boolean (0-1) constraints and use the pseudo-Boolean solver PBS [5] 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[8]. PBS accepts a SAT 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 can use a binary search to solve a pseudo-Boolean optimization problem, 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 5, 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 7) 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 8) 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.

[8] 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.

```
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
```

Fig. 8. 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	

Fig. 9. 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
```

Fig. 10. Δ values ($\Delta = 1$) for execution in Figure 8

The values of the Δs (Figure 10) 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 Section 5 (see Figure 16). This is an instance of 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.

4.2 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.

More generally, this proof should hold for *any* metric which can be formulated in terms of a Levenshtein distance such that operations can be represented by mutually exclusive independent terms that can be conjoined (as with the atomic changes to the SSA form representation). Such a formulation should be possible for the non-SSA form metric used with abstract explanation [15]; however, the reduction to atomic terms in that case is considerably less natural, and the value of an explanation as a conjunction in terms of predicates on states and predicate values once alignment and position are taken into account is dubious.

```
1  int main () {
2    int input1, input2;
3    int x = 1, y = 1, z = 1;
4    if (input1 > 0) {
5      x += 5;
6      y += 6;
7      z += 4;
8    }
9    if (input2 > 0) {
10     x += 6;
11     y += 5;
12     z += 4;
13   }
14   assert ((x < 10) || (y < 10));
15 }
```

Fig. 11. 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
```

Fig. 12. Δ values for slice.c

5 Δ-Slicing

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 11 and 12. The change to `z` is necessary but also irrelevant to the assertion on line 14. Various static or dynamic slicing techniques [61] 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 *is* relevant. Dynamic slicing 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. Starting with differences rather than the full execution goes beyond the reductions provided by a dynamic slice.

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 6.1. 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 6.3) in that an explanation may be reduced to a very small set of Δs that clearly cannot contain a fault.

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 which 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.

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[9] for `x#3`, `x#3 == x#2 + 6` is replaced in the Δ-slicing constraints with the constant assignment `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 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 4.1.
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:

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

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

```
...
{-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
```

Fig. 13. 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
```

Fig. 14. Δ-slicing constraints for slice.c

– 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 13 shows some of the original constraints for slice.c. The modified constraints used for computing the Δ-slice are shown in Figure 14. The relaxation of the transition relation allows for a better solution to the optimization problem, the Δ-slice shown in Figure 15. 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 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
```

Fig. 15. Δ-slice for slice.c

```
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
```

Fig. 16. Δ-slice for minmax.c

Revisiting the original example program, we can apply Δ-slicing to the explanation in Figure 10 and obtain the smaller explanation shown in Figure 16.

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.

5.1 Explaining and Slicing in One Step

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 technique as *one-step* slicing as the execution and slice are computed at the same time. Before returning to the issue of which approach is best, we will demonstrate that solving the second optimization problem is indeed feasible.

5.1.1 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:

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

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

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

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).

5.1.2 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 to a user 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-phase 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, this will match the shadow value, which guarantees that all changes are a subset of a valid successful execution. Figure 17 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.

```
...
{-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
```

Fig. 17. One-step Δ-slicing constraints for slice.c

5.1.3 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 2 in Section 6.4 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 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 only makes sense 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 Section 6, 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

```
 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())));
```

Fig. 18. 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
```

Fig. 19. 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
```

Fig. 20. Code for violated assertion

that are not forced by relevant changes, it is probably best to simply optimize the distance between the executions and trust that Δ-slicing will remove irrelevant behavior — once we have some context in which to define relevance.

An appealing compromise 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.

6 Case Studies and Evaluation

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 6.3.

6.1 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 [54] 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 [24] provided a partial specification that was able to detect faults in 5 of the 41 versions (CBMC's automatic array bounds checking detected 2 faults). In addition to these assertions, it was necessary to include some obvious assumptions on the inputs[10].

Variation #1 of the TCAS code differs from the correct version in a single line (Figure 18). A $\geq$ comparison in the correct code has been changed into a $>$ comparison on line 100. Figure 19 shows the result of applying `explain` to the counterexample generated by CBMC for this error (after Δ-slicing). The counterexample passes through 90 states before an assertion (shown in Figure 20) fails.

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. 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. 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 22) 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 only needs to observe 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. The original counterexample is still valid, as it clearly satisfies the assumption[11]. 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.

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`.

[10] 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).

[11] 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.

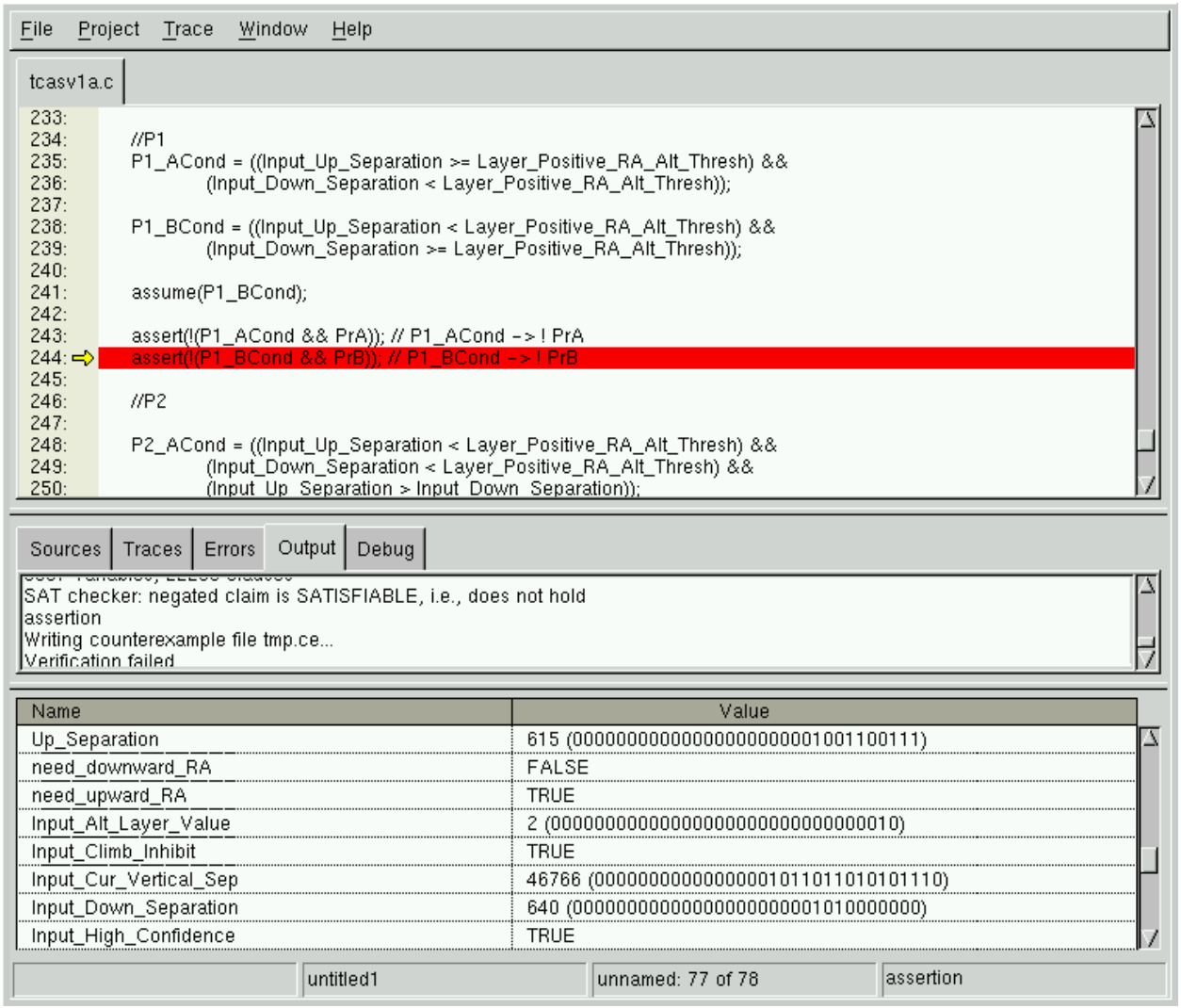

Fig. 21. Explaining tcasv1.c

```
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 (\guard#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 (\guard#16)
                  was TRUE
                line 152 function c::alt_sep_test
Guard changed:  \guard#15 && !need_upward_RA#1 (\guard#17)
                  was FALSE
                line 152 function c::alt_sep_test
Guard changed:  \guard#17 && !need_downward_RA#1 (\guard#19)
                  was FALSE
                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
                line 230 function c::main
```

Fig. 22. Second explanation for variation #1 (after Δ-slicing)

The rest of the explanation shows how this value propagates to result in a correct choice of RA.

Figures 21 and 23 show the explanation process as it appears in the `explain` GUI. The error is highlighted in red (black), and all source lines appearing in the explanation are highlighted in orange (gray). Note that although the counterexample in the screenshots is actually for a different assertion violation (CBMC's initial settings determine which counterexample is produced), the localization information is unchanged.

For one of the five interesting[12] 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

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

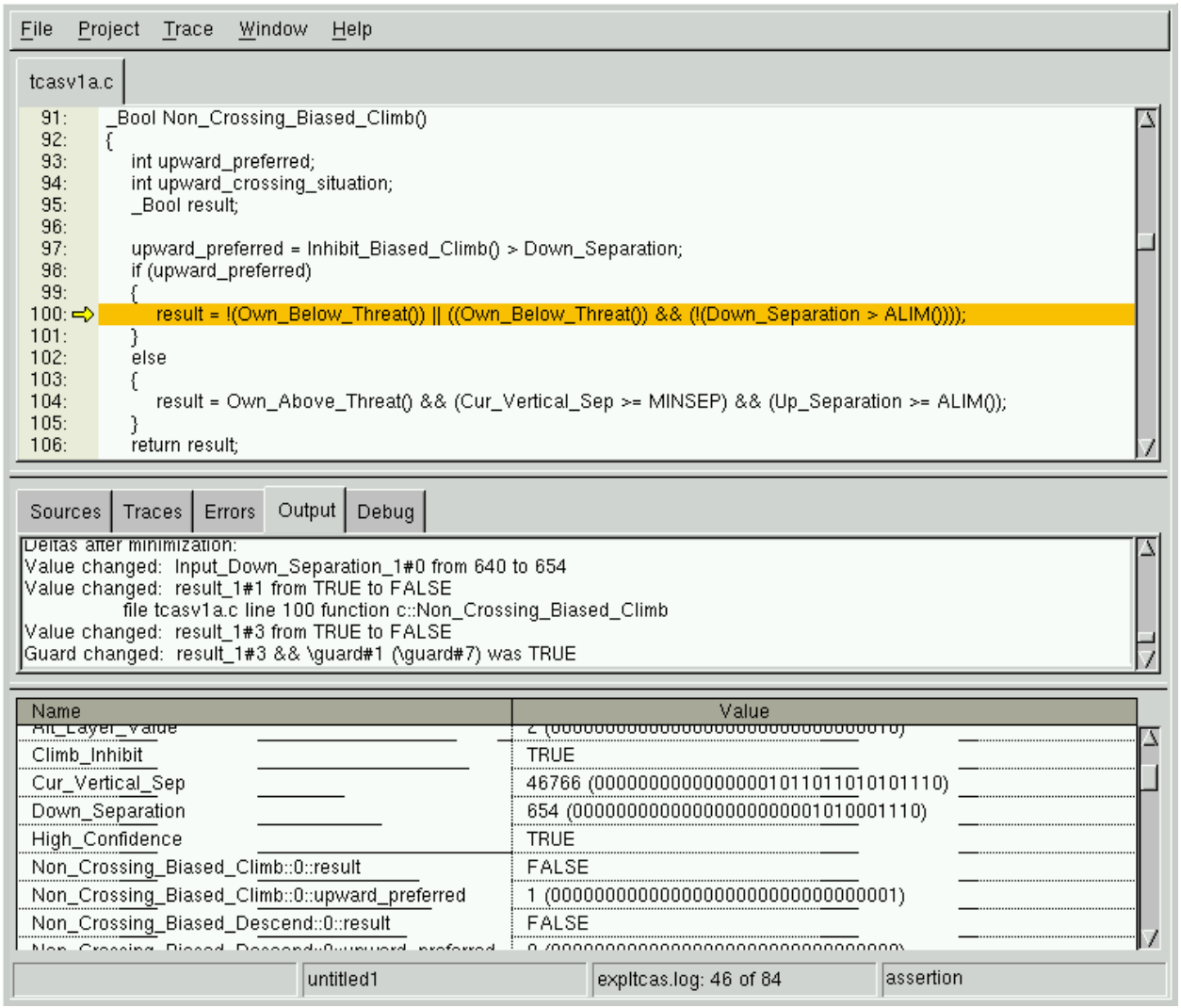

Fig. 23. Correctly locating the error in tcasv1.c

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 often meaningless) is implausible.

6.2 µC/OS-II Case Study

µC/OS-II [3] 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 24 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 25 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

```
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;
```

Fig. 24. Code structure for μC/OS-II error

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

Fig. 25. Explanation for μC/OS-II error

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`.

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 236,013 variables and 568,989 clauses, with 69 variables appearing in the pseudo-Boolean constraint.

6.3 Evaluation of Fault Localization

Renieris and Reiss [52] propose a scoring function for evaluating error localization techniques based on program dependency graphs (PDGs) [35]. 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 only contain 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 [22].

Renieris and Reiss report fault localization results for the entire Siemens suite [52]. 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 suitable manner 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.

Table 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 [8] 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. For versions #1, #11, #31, and #41, the orig-

	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

Table 1. Scores for localization techniques. Explanation execution times in seconds. Best results in boldface. FAIL indicates memory exhaustion (> 768MB used). * indicates alternative scoring method.

inal 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. 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. These 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 an 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 [31] 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 was 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 [52]. 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 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 [52]. Run-times are not reported for these methods, as they are (roughly) equivalent to the time needed to run the various test cases, and therefore not suitable for comparison to the model checking approaches.

The last two columns provide a baseline: scores and times for the counterexamples generated by the CBMC model checker.

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 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 mod-

```
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
```

Fig. 26. One-step slicing report for TCAS variation #1

ified 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 1), the counterexample itself receives a score of 0.00[13], 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).

6.4 Evaluation of One-Step Slicing

Table 2 shows the poor results obtained by applying one-step slicing (Section 5.1) 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.

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. Consider, for example, the explanation produced for the TCAS variation #1 by one-step slicing (Figure 26).

The code shown in Figure 27 is used to determine if a Resolution Advisory is computed by TCAS: the properties for TCAS are 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 20). The change in this explanation results in the `if` branch

[13] 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.

```
_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;
```

Fig. 27. Code for determining if RA is computed

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. 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.

7 Conclusions and Future Work

No single "best" approach for error explanation can be formally defined, as the problem is inherently to some extent psychological. David Lewis' approach to causality is both intuitively appealing and readily translated into mathematical terms, and therefore offers a practical means for deriving concise explanations of program errors. A distance metric informed by Lewis' approach makes it possible to generate provably-most-similar successful executions by translating metric constraints into pseudo-Boolean optimality problems. Experimental results indicate that such executions are quite useful for localization and explanation.

There are a number of interesting avenues for future research. An in-depth look at interactive explanation in practice and further empirical evaluation is certainly in order. It might be fruitful to use static slicing to improve the distance metric, as suggested in Section 5.1.3. Combining the SSA form based metric with predicate abstraction appears likely to result in better explanations than either the current CBMC or MAGIC approaches [15].

Acknowledgments: The authors would like to thank Edmund Clarke, Reid Simmons, David Garlan, Willem Visser, Manos Renieris, Fadi Aloul, Andreas Zeller, Dim-

	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 2. Scores with one-step slicing

itra Giannakopoulou, and Flavio Lerda for their valuable assistance.

References

1. `http://www.cs.cmu.edu/~modelcheck/cbmc/`.
2. `http://www.askigor.com`.
3. `http://www.ucos-ii.com/`.
4. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering*, pages 143–151, Toulouse, France, October 1995.
5. F. Aloul, A. Ramani, I. Markov, and K. 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.
6. B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages*, pages 1–11, San Diego, CA, January 1988.
7. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In *Logic Based Program Synthesis and Tranformation, 12th International Workshop*, September 2002.
8. P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering*, Paris, France, July 2001.
9. T. Ball and S. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.
10. T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages*, pages 97–105, New Orleans, LA, January 2003.
11. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, Toronto, Canada, May 2001.
12. A. Biere, C. Artho, and V. 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.
13. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Amsterdam, The Netherlands, March 1999.
14. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.
15. S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering*, Newport Beach, CA, November 2004. To appear.
16. W. Chan. Temporal-logic queries. In *Computer-Aided Verification*, pages 450–463, Chicago, IL, July 2000.
17. M. Chechik and A. Gurfinkel. Proof-like counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 160–175, Warsaw, Poland, April 2003.
18. J. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220, Rome, Italy, July 2002.
19. E. 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.
20. E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*, pages 427–432, San Francisco, CA, June 1995.
21. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
22. H. Cleve and A. Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, St. Louis, MO, May 2005. To appear.
23. J. Cobleigh, D. Giannakopoulou, and C. Păsăreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346, Warsaw, Poland, April 2003.
24. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. 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.
25. N. Dodoo, A. Donovan, L. Lin, and M. Ernst. Selecting predicates for implications in program analysis. `http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.ps`, 2000.
26. M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, May 1999.

27. D. Galles and J. Pearl. Axioms of causal relevance. *Artificial Intelligence*, 97(1-2):9–43, 1997.
28. A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, Barcelona, Spain, March-April 2004.
29. A. Groce and D. Kroening. Making the most of BMC counterexamples. In *Workshop on Bounded Model Checking*, Boston, MA, July 2004. To appear.
30. A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with `explain`. In *Computer-Aided Verification*, Boston, MA, July 2004. To appear.
31. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, Portland, OR, May 2003.
32. A. Gurfinkel, B. Devereux, and M. Chechik. Model exploration with temporal logic query checking. In *Foundations of Software Engineering*, pages 139–148, Charleston, SC, November 2002.
33. M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.
34. P. Horwich. *Asymmetries in Time*, pages 167–176. 1987.
35. S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, May 1992.
36. D. Hume. *A Treatise of Human Nature*. London, 1739.
37. D. Hume. *An Enquiry Concerning Human Understanding*. London, 1748.
38. H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–458, Grenoble, France, April 2002.
39. J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, pages 467–477, Orlando, FL, May 2002.
40. J. Kim. Causes and counterfactuals. *Journal of Philosophy*, 70:570–572, 1973.
41. G. Kókai, L. Harmath, and T. Gyimóthy. Algorithmic debugging and testing of Prolog programs. In *Workshop on Logic Programming Environments*, pages 14–21, July 1997.
42. D. Kroening, E. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Barcelona, Spain, March-April 2004.
43. D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.
44. D. Lewis. *Counterfactuals*. Harvard University Press, 1973 [revised printing 1986].
45. P. Lucas. Analysis of notions of diagnosis. *Artificial Intelligence*, 105(1-2):295–343, 1998.
46. C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Model-based debugging of Java programs. In *Workshop on Automatic Debugging*, Munich, Germany, August 2000.
47. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.
48. P. Nayak and B. Williams. Fast context switching in real-time propositional reasoning. In *National Conference on Artificial Intelligence*, pages 50–56, Providence, RI, July 1997.
49. J. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *International Symposium on Programming*, pages 337–351, Torino, Italy, April 1982.
50. K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, Barcelona, Spain, March-April 2004.
51. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
52. M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, pages 30–39, Montreal, Canada, October 2003.
53. T. Reps, T. Ball, M. Das, and J. 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.
54. G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.
55. D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison Wesley, 1983.
56. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
57. N. Sharygina and D. Peled. A combined testing and verification approach for software reliability. In *Formal Methods Europe*, pages 611–628, Berlin, Germany, March 2001.
58. R. Simmons and C. Pecheur. Automating model checking for autonomous systems. In *AAAI Spring Symposium on Real-Time Autonomous Systems*, 2000.
59. E. Sosa and M. Tooley, editors. *Causation*. Oxford University Press, 1993.
60. L. Tan and R. Cleaveland. Evidence-based model checking. In *Computer-Aided Verification*, pages 455–470, Copenhagen, Denmark, July 2002.
61. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
62. I. Vesey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.
63. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.
64. F. Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002.
65. A. Zeller. Isolating cause-effect chains from computer programs. In *Foundations of Software Engineering*, pages 1–10, Charleston, SC, November 2002.
66. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
67. X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *International Conference on Software Engineering*, pages 319–329, Portland, OR, May 2003.