Error Explanation with Distance Metrics

Alex Groce[1]

Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213

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 an automated approach for assisting users in understanding and isolating errors in ANSI C programs. The approach 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. This paper describes the application of a technology traditionally used for *finding* errors to the problem of *understanding and isolating* errors.

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

Model checking [9] 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 [8] 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 SAT problem, S. The satisfying assignments of S are bounded *executions* of P that violate the specification (counterexamples).

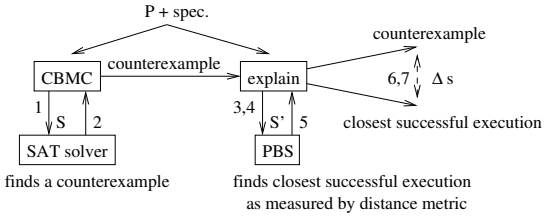

Fig. 1. Explaining an error using distance metrics.

2. CBMC uses a SAT solver to find a counterexample.
3. The `explain` tool produces a SAT problem, S'. The satisfying assignments of S' are executions of P that do *not* violate the specification.
4. `explain` uses the counterexample to add to S' 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.
5. `explain` uses the PBS solver to find a successful execution that is as close as possible to the counterexample.
6. The differences (Δs) between the successful execution and the counterexample are computed.
7. 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*.
8. If the explanation is unsatisfactory the user may add assumptions and return to step 1 (see Section 5).

There are many possible approaches to error explanation. A basic notion shared by many researchers in this area [5, 12, 24] and many philosophers [21] 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) [12, 18, 25].

David Lewis [16] has proposed a theory of causality that provides a justification for the second intuition if we assume explanation is the analysis of causal relationships. Following Hume 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*). 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. Distance metrics

between possible worlds are problematic, and Lewis' proposed criteria for such metrics have met with criticism [21].

Program executions are much more amenable to measurement and predication than possible worlds. If we replace possible worlds with program executions and events with propositions about those executions, a practically applicable definition 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. *$\exists$ an execution b . $\neg c(b) \wedge \neg e(b) \wedge (\forall b' \,.\, (\neg c(b') \wedge e(b')) \Rightarrow (d(a,b) < d(a,b')))$*

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.

This paper describes a distance metric that allows determination of causal dependencies and the implementation of that metric in a tool called `explain` that extends CBMC [1], a model checker for programs written in ANSI C. Note that the focus of the paper is not on computing causal dependence, which is only useful *after* forming a hypothesis about a possible cause c.

The basic approach, presented in Sections 3 and 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 3.4 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

Recent work has described proof-like and evidence-based counterexamples [7, 22]. Automatically generating assumptions for verification [10] can also be seen as a kind of error explanation. 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 MSR's SLAM model checker [5] and NASA's JPF model checker [12]. Jin, Ravi, and Somenzi proposed a game-like explanation (directed more at hardware than software systems) in which an

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

adversary tries to force the system into error [14]. Of these, only JPF uses a (weak) notion of distance between traces, and it cannot solve for nearest successful executions.

Zeller's delta debugging [25] extrapolates between failing and successful test cases to find more similar executions, with respect to inputs only. Delta-debugging for deriving cause-effect chains [24] 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.

Renieris and Reiss [18] describe an approach that is quite similar in spirit to the one described here, with the advantages and limitations of a testing rather than model checking basis. They use a distance metric to *select* a successful test run from among a given set rather than, as in this paper, to automatically *generate* a successful run that resembles a given failing run as much as is possible. Experimental results show that this makes their fault localization highly dependent on test case quality. Section 5.1 makes use of a quantitative method for evaluating fault localization approaches proposed by Renieris and Reiss.

This paper presents a new distance metric for program executions, based on David Lewis' counterfactual analysis of causality. The other original contributions are: a method for solving the optimization problem of finding the closest successful execution to a given failing execution and a new slicing technique which can remove irrelevant code that cannot be sliced away by previous static or dynamic slicing approaches.

3 Distance Metrics for Program Executions

A distance metric [20] for program executions is a function $d(a, b)$ (where a and b are executions of the same program) that satisfies certain 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:* $\forall a \,.\, \forall b \,.\, \forall c \,.\, d(a, b) + d(b, c) \geq d(a, c)$

3.1 Representing Program Executions

In order to compute distances between program executions, we need a single, well-defined representation for those executions. Bounded model checking (BMC) [6] 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 [15] 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) [3] 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.

Given the example program minmax.c (Figure 2), CBMC produces the constraints shown in Figure 3 (U is not needed, as minmax.c is loop-free)[2]. The renamed variables describe unique assignment points: `most#2` denotes the second possible assignment to `most`. CBMC assigns uninitialized (`#0`) values nondeterministically — thus `input1, input2,` and `input3` will be unconstrained 32 bit integer values. The assumption on line 5 limits executions to those in which these values are non-negative (constraints {-15} and {-16} encode this requirement). The `\guard` variables encode the control flow of the program (`\guard1` is the value of the conditional on line 6, 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: the constraint {-10}, for instance, assigns `most#3` to either `most#2` or `most#1`, depending on the conditional for the assignment to `most#2` (the syntax is that of the C conditional expression). `most#3`, therefore, is the value assigned to `most` at the point before the execution of line 8 of minmax.c. A solution to this set of constraints is an erroneous execution of minmax.c: a counterexample.

CBMC generates CNF clauses representing the conjunction of ({-1}∧{-2}∧ ... {-16}) with the negation of the claim (¬{1}). CBMC calls ZChaff [17], 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 4).

Figure 4 includes both an easier-to-read summary of the full counterexample generated by CBMC and the more detailed internal representation consisting of the set of all values assigned to the variables appearing in the constraints.

For given loop bounds, all executions of a program can be represented as sets of assignments to the variables appearing in the constraints. Moreover, all

[2] Output is slightly simplified for readability.

```
1  int main () {
2    int input1, input2, input3;       //input values
3    int least = input1;
4    int most = input1;
5    assume ((input1 >= 0) && (input2 >= 0) && (input3 >= 0));
6    if (most < input2)                //guard1
7      most = input2;
8    if (most < input3)                //guard2
9      most = input3;
10   if (least > input2)               //guard3
11     most = input2;                  //ERROR:  should be ‘‘least = input2’’
12   if (least > input3)               //guard4
13     least = input3;
14   assert (least <= most);           //specification
15 }
```

Fig. 2. minmax.c

```
{-16} \guard#0 => input1#0 >= 0 && input2#0 >= 0 && input3#0 >= 0
{-15} \guard#0 == TRUE
{-14} least#1 == input1#0
{-13} most#1 == input1#0
{-12} \guard#1 == (most#1 < input2#0 && \guard#0)
{-11} most#2 == input2#0
{-10} most#3 == (\guard#1 && \guard#0 ? most#2 : most#1)
{-9}  \guard#2 == (most#3 < input3#0 && \guard#0)
{-8}  most#4 == input3#0
{-7}  most#5 == (\guard#2 && \guard#0 ? most#4 : most#3)
{-6}  \guard#3 == (least#1 > input2#0 && \guard#0)
{-5}  most#6 == input2#0
{-4}  most#7 == (\guard#3 && \guard#0 ? most#6 : most#5)
{-3}  \guard#4 == (least#1 > input3#0 && \guard#0)
{-2}  least#2 == input3#0
{-1}  least#3 == (\guard#4 && \guard#0 ? least#2 : least#1)
|-------------------------
{1}  \guard#0 => least#3 <= most#7
```

Fig. 3. Constraints generated for minmax.c

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[3]. This guarantees that any two executions will be represented by constraints on the same variables. $d(a, b)$ is equal to the number of variables to which a and b assign different values.

Definition 2 (distance between two executions, $d(a, b)$). *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

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

This definition is equivalent to the Levenshtein distance [20] if we consider executions as strings where the alphabet elements are assignments and substitution is the only allowed operation. The properties of inequality guarantee that d satisfies the four metric properties.

The representation for executions presented here has the advantage of combining precision and relative simplicity, and results in a very clean distance metric. All of the pitfalls involved in trying to align executions with different control

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

```
Initial State
------------------------------------------------------
  input1=2147483618 (01111111111111111111111111100010)
  input2=1073741792 (00111111111111111111111111100000)
  input3=2147483615 (01111111111111111111111111011111)
State 1
------------------------------------------------------
  least=2147483618 (01111111111111111111111111100010)
State 2
------------------------------------------------------
  most=2147483618 (01111111111111111111111111100010)
State 9 file minmax.c line 11 function c::main
------------------------------------------------------
  most=1073741792 (00111111111111111111111111100000)
State 11 file minmax.c line 13 function c::main
------------------------------------------------------
  least=2147483615 (01111111111111111111111111011111)
Failed assertion: assertion file minmax.c line 14 function c::main
```

input1#0 = 2147483618	most#2 = 1073741792	most#6 = 1073741792
least#1 = 2147483618	most#3 = 2147483618	most#7 = 1073741792
most#1 = 2147483618	\guard#2 = FALSE	\guard#4 = TRUE
input2#0 = 1073741792	most#4 = 2147483615	least#2 = 2147483615
input3#0 = 2147483615	most#5 = 2147483618	least#3 = 2147483615
\guard#1 = FALSE	\guard#3 = TRUE	

Fig. 4. Counterexample for minmax.c

input1#0Δ == (input1#0 != 2147483618)	most#4Δ == (most#4 != 2147483615)
least#1Δ == (least#1 != 2147483618)	most#5Δ == (most#5 != 2147483618)
most#1Δ == (most#1 != 2147483618)	\guard#3Δ == (\guard#3 != TRUE)
input2#0Δ == (input2#0 != 1073741792)	most#6Δ == (most#6 != 1073741792)
input3#0Δ == (input3#0 != 2147483615)	most#7Δ == (most#7 != 1073741792)
\guard#1Δ == (\guard#1 != FALSE)	\guard#4Δ == (\guard#4 != TRUE)
most#2Δ == (most#2 != 1073741792)	least#2Δ == (least#2 != 2147483615)
most#3Δ == (most#3 != 2147483618)	least#3Δ == (least#3 != 2147483615)
\guard#2Δ == (\guard#2 != FALSE)	

Fig. 5. Δs for minmax.c and the counterexample in Figure 4

flow for purposes of comparison are avoided by the use of SSA. Obviously, the details of the SSA encoding may need to be hidden from non-expert users (the CBMC GUI provides this service). Any gains in the direct presentability of the representation (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.

3.3 Combining the Metric and Constraints

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 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. All of the Δ functions necessary to compute the distance are added as new constraints (Figure 5) by the `explain` tool.

These constraints do not affect satisfiability; correct values can always be assigned for the Δs. These values are used to encode the optimization problem. For a fixed a, $d(a, b) = n$ can directly be encoded as a constraint by encoding that exactly n of the Δs be set to 1 in the CNF. However, it is more efficient

```
Initial State ------------------------------------------------------
input1=2147483618 (01111111111111111111111111100010)
input2=1073741792 (00111111111111111111111111100000)
input3=0 (00000000000000000000000000000000)
State 1
------------------------------------------------------
  least=2147483618 (01111111111111111111111111100010)
State 2
------------------------------------------------------
  most=2147483618 (01111111111111111111111111100010)
State 9 file minmax.c line 11 function c::main
------------------------------------------------------
  most=1073741792 (00111111111111111111111111100000)
State 11 file minmax.c line 13 function c::main
------------------------------------------------------
  least=0 (00000000000000000000000000000000)
Error explanation deltas:
Value changed:  input3#0 from 2147483615 to 0
Value changed:  most#4 from 2147483615 to 0
                file minmax.c line 9 function c::main
Value changed:  least#2 from 2147483615 to 0
                file minmax.c line 13 function c::main
Value changed:  least#3 from 2147483615 to 0
```

Fig. 6. Closest successful execution and Δ values for minmax.c

to use pseudo-Boolean (PB) constraints and replace ZChaff with the PB-solver PBS [2]. With PBS we can express and solve for the conditions $d(a, b) = n$, $d(a, b) < n$, $d(a, b) \geq n$, etc., and, more importantly, directly solve optimization constraints, minimizing or maximizing $d(a, b)$.

From the counterexample shown in Figure 4, we can generate an execution (1) with minimal distance from the counterexample and (2) in which the assertion on line 14 is not violated. Constraints {-1}-{-16} are conjuncted with the Δ constraints (Figure 5) and the unnegated verification claim {1}. The PB constraints express an optimization problem of minimizing the sum of the Δs. The result is an execution (Figure 6) in which a change in the value of `input3` results in `least <= most` being `true` at line 14. This solution is not unique. In general, there may be a very large class of executions that have the same distance from a counterexample.

The values of the Δs allow us to examine precisely the points at which the two executions differ. The first change is the different value for `input3`. 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 assignment to `most` at line 9. Because the condition on line 8 is still not satisfied (no guards change values — control flow is the same as in the counterexample), the value of `most` which reaches line 10 (`most#5`) is not changed. The final assignment to `most` is also unchanged (and incorrect). The only change which reaches the assertion is the alteration to the value of `least`, which is now correct because `input2` is no longer minimal. `input2` cannot be assigned to `least` due to the faulty code on line 11.

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

In the example, δ is the predicate `(input3#0 != 0)` $\vee$ `(most#4 != 0)` $\vee$ `(least#2 != 0)` $\vee$ `(least#3 != 0)`. Finding the closest successful execution also produces a predicate c on which the error is causally dependent[4]. δ can be used as a starting point for hypotheses about a more general cause for the error.

4 Δ-Slicing

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

Consider the program and Δ values in Figure 7. The change to `c` is necessary but also irrelevant to the assertion on line 10. In this case, various static or dynamic slicing techniques [23] would suffice to remove the unimportant variables. 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.

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

[4] Note that the proof also holds for other successful executions: minimal distance minimizes the number of terms in δ.

```
 1  int main () {
 2    int input1, input2;
 3    int a = 1, b = 1, c = 1;
 4    if (input1 > 0) {
 5      a += 5; b += 6; c += 4;
 6    }
 7    if (input2 > 0) {
 8      a += 6; b += 5; c += 4;
 9    }
 10   assert ((a < 10) || (b < 10));
 11 }
Value changed:  input2#0 from 1073741824 to 0
Guard changed:  input2#0 > 0 && \guard#0 (\guard#2) was TRUE
                file slice.c line 7 function c::main
Value changed:  a#5 from 12 to 6
Value changed:  b#5 from 12 to 7
Value changed:  c#5 from 9 to 5
```

Fig. 7. slice.c and Δ values

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 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 this "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. For values that did not change ($\Delta(i) = 0$) the constant constraint $v_i = val_i^a$ is used. 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.

In slice.c, for example, the constraint `c#5 == (input2#0 > 0 && \guard#0 ? c#4 : c#3)` is replaced with `((c#5 == 9) || ((c#5 == 5) && (c#5 == (input2#0 > 0 && \guard#0 ? c#4 : c#3))))`, and so forth. The relaxation of

```
Value changed:  input2#0 from 1073741824 to 0
Guard changed:  input2#0 > 0 && \guard#0 (\guard#2) was TRUE
                file slice.c line 7 function c::main
Value changed:  b#5 from 12 to 7
```

Fig. 8. Δ-slice for slice.c

the transition relation allows for a better solution to the optimization problem, the Δ-slice shown in Figure 8. Another slice would replace `b` with `a`. It is only necessary to observe a change in *either* `a` or `b` to satisfy the assertion. Previous dynamic slicing techniques do not appear to provide this kind of information.

5 Case Study: TCAS

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 [19] 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.

Renieris and Reiss made use of the entire Siemens suite [18]. 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. 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. The TCAS component, however, is suitable for model checking with almost no modification. A previous study of the component using symbolic execution [11] provided a partial specification that was able to detect faults in 5 of the 41 versions (CBMC's automatic array bounds checking detected 2). In addition to these assertions, it was necessary to include some obvious assumptions on the inputs.

Variation #1 of the TCAS code differs from the correct version in a single line (Figure 9). A $\geq$ comparison in the correct code has been changed into a $>$ comparison on line 100. Figure 10 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 fails.

```
 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. 9. 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
                file tcasv1.c line 255 function c::main
    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. 10. First explanation for variation #1 (after Δ-slicing), code for violated assertion

```
Value changed:  Input_Down_Separation_1#0 from 500 to 504
Value changed:  Down_Separation#1 from 500 to 504
                file tcasv1a.c line 215 function c::main
Value changed:  result_1#1 from TRUE to FALSE
                file tcasv1a.c line 100 function c::Non_Crossing_Biased_Climb
Value changed:  result_1#3 from TRUE to FALSE
Value changed:  tmp#1 from TRUE to FALSE
                file tcasv1a.c line 106 function c::Non_Crossing_Biased_Climb
Guard changed:  \guard#1 && tmp#1 (\guard#7) was TRUE
                file tcasv1a.c line 144 function c::alt_sep_test
Value changed:  need_upward_RA_1#1 from TRUE to FALSE
                file tcasv1a.c line 144 function c::alt_sep_test
Guard changed:  \guard#15 && need_upward_RA_1#1 (\guard#16) was TRUE
                file tcasv1a.c line 152 function c::alt_sep_test
Guard changed:  \guard#15 && !need_upward_RA_1#1 (\guard#17) was FALSE
                file tcasv1a.c line 152 function c::alt_sep_test
Guard changed:  \guard#17 && !need_downward_RA_1#1 (\guard#19) was FALSE
                file tcasv1a.c line 156 function c::alt_sep_test
Value changed:  ASTUpRA#2 from TRUE to FALSE
Value changed:  ASTUpRA#3 from TRUE to FALSE
Value changed:  ASTUpRA#4 from TRUE to FALSE
Value changed:  PrB#1 from TRUE to FALSE
                file tcasv1a.c line 230 function c::main
```

Fig. 11. Second explanation for variation #1 (after Δ-slicing)

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 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[5]. After model checking the program again we can reapply `explain`. The new explanation (Figure 11) is far more useful.

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

[5] This implication antecedent solution can presumably be automated.

Var.	exp	slice	time	assm	slice	time	JPF	time	(R&R) n-c	(R&R) n-s	CBMC
#1	0.51	0.00	4	0.90	**0.91**	4	0.87	1521	0.00/0.00	0.58/0.58	0.41
#11	0.36	0.00	5	0.88	0.93	7	0.93	5673	0.00/**0.95**	0.00/**0.95**	0.51
#31	0.76	0.00	4	0.89	**0.93**	7	FAIL	-	0.00/0.00	0.00/0.00	0.46
#40	0.75	**0.88**	6	-	-	-	0.87	30,482	0.83/0.83	0.77/0.77	0.35
#41	0.68	0.00	8	0.84	0.88	5	0.30	34	0.56/0.60	**0.92/0.92**	0.38

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

For one of the five interesting[6] variations (#40), a useful explanation is produced without any added assumptions. Variations #11 and #31 also require assumptions about the antecedent of an implication in an assertion. The final variation, #41, requires an antecedent assumption and an assumption requiring that TCAS is enabled (the successful execution finally produced differs from the counterexample to such an extent that changing inputs so as to disable TCAS is a closer solution).

5.1 Evaluation of Fault Localization

Renieris and Reiss[18] propose a scoring function for evaluating error localization techniques based on program dependency graphs [13]. 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.

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 [4] generated the pdgs and code provided by Manos Renieris computed the scores for the error reports. The second and third columns show scores given to reports provided by `explain` without using added

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

assumptions, before and after Δ-slicing. For #1, #11, #31, and #41, a fault is included as an effect of a change in input, but removed by slicing. Columns five and six show `explain` results after adding appropriate assumptions, if needed. 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 (based on subsets[7] of the full JPF reports) is therefore of dubious value. Columns eleven and twelve show minimum and maximum scores for two methods given by Renieris and Reiss [18]. After introducing assumptions and slicing, 0.88 was the lowest score for an `explain` report. Ignoring pre-assumption inclusions of faults, Δ-slicing always resulted in improved scores.

In other experiments, `explain` produced a 3 line (correct) localization of a 58 step locking-protocol counterexample for a 2991 line portion of a real-time OS microkernel in 158 seconds.

6 Future Work and Conclusions

There are a number of interesting avenues for future research. The current analysis of one failing run can be extended to the problem of "n different counterexamples with m different explanations." Another extension in progress applies the basic technique when using predicate abstraction and for LTL path+cycle counterexamples. An in-depth look at interactive explanation in practice and discussion of using `explain` to produce *maximally* different counterexamples (for comparison of similarities) are also beyond the scope of this paper.

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.

Acknowledgments: I would like to thank Ofer Strichman, Willem Visser, Daniel Kroening, Manos Renieris, Fadi Aloul, Andreas Zeller, and Dimitra Giannakopoulou for their assistance.

References

1. `http://www.cs.cmu.edu/~modelcheck/cbmc/`.
2. 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, 2002.

[7] specifically $only(pos) \cup only(neg) \cup cause(pos) \cup cause(neg) \cup (all(neg) \backslash all(pos)) \cup (all(pos) \backslash all(neg))$ for transitions and transforms [12]

3. B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages*, pages 1–11, 1988.
4. P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering*, 2001.
5. T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages*, pages 97–105, 2003.
6. 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, 1999.
7. M. Chechik and A. Gurfinkel. Proof-like counter-examples. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 160–175, 2003.
8. 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, 1995.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
10. 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, 2003.
11. 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, 2001.
12. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
13. S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *International Conference of Software Engineering*, pages 392–411, 1992.
14. 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, 2002.
15. 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*, 2004. To appear.
16. D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.
18. M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, 2003.
19. G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.
20. D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison Wesley, 1983.
21. E. Sosa and M. Tooley, editors. *Causation*. Oxford University Press, 1993.
22. L. Tan and R. Cleaveland. Evidence-based model checking. In *Computer-Aided Verification*, pages 455–470, 2002.
23. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
24. A. Zeller. Isolating cause-effect chains from computer programs. In *Foundations of Software Engineering*, pages 1–10, 2002.
25. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.