Explaining Abstract Counterexamples

Sagar Chaki
School of Computer Science,
Carnegie Mellon University
chaki@cs.cmu.edu

Alex Groce
School of Computer Science,
Carnegie Mellon University
agroce@cs.cmu.edu

Ofer Strichman
Technion, Haifa, Israel
ofers@ie.technion.ac.il

ABSTRACT

When a program violates its specification a model checker produces a counterexample that shows an example of undesirable behavior. It is up to the user to understand the error, locate it, and fix the problem. Previous work introduced a technique for explaining and localizing errors based on finding the closest execution to a counterexample, with respect to a distance metric. That approach was applied only to *concrete executions* of programs. This paper extends and generalizes the approach by combining it with *predicate abstraction*. Using an abstract state-space increases scalability and *makes explanations more informative*. Differences between executions are presented in terms of predicates derived from the specification and program, rather than specific changes to variable values. Reasoning to the cause of an error from the fact that in the failing run `x < y`, but in the successful execution `x = y` is easier than reasoning from the information that in the failing run `y = 239`, but in the successful execution `y = 232`. An abstract explanation is *automatically generalized*.

Predicate abstraction has previously been used in model checking purely as a state-space reduction technique. However, an abstraction good enough to enable a model checking tool to find an error is also likely to be useful as an *automatically generated high-level description of a state space* — suitable for use by programmers. Results demonstrating the effectiveness of abstract explanations support this claim.

Keywords

model checking, predicate abstraction, fault localization

1. INTRODUCTION

Moving from a trace demonstrating that a program does not satisfy a specification to an understanding of what is wrong with the program (or the specification) and how to fix it is a difficult task. Recently, there has been a movement to apply the model checking [11] technology traditionally used to *find* errors to the problem of *understanding and isolating* errors.

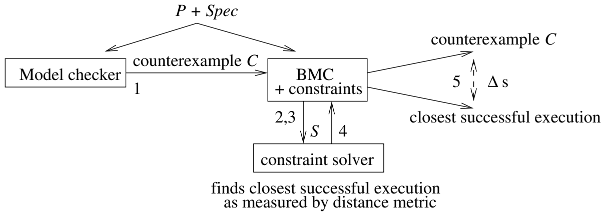


Figure 1: Error explanation with distance metrics.

Error explanation describes any automated approach to aiding a user in moving from a particular example of specification failure to an understanding of the *essence* of the failure and, perhaps, to a correction for the problem. *Fault localization* is the more specific task of identifying the faulty core of a system.

This paper describes an extension to the distance metric [24] based approach to error explanation and fault localization [14]. The explanation method may be summarized as follows (Figure 1):

1. Generate a counterexample C for a specification *Spec* of a program P using a model checker.

2. Use bounded model checking (BMC) [7] to unwind the transition relation of P to a finite bound[1] and produce a propositional formula S that represents exactly the executions of P that do *not* violate *Spec*.

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

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

5. Present the differences (Δs) between the successful execution and the counterexample as *explanation and localization* for the error.

The notion that these minimal differences serve to explain (in a causal sense) the error is justified by an adaptation of

[1] In earlier work [14], this value was typically the same as the bound used to produce the counterexample C.

FSE '04 Newport Beach, CA USA

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

Figure 2: minmax.c

```
Value changed:  input3#0 from 2147483615 to 0
Value changed:  most#4 from 2147483615 to 0
                file minmax.c line 8 func-
tion c::main
Value changed:  least#2 from 2147483615 to 0
                file minmax.c line 12 func-
tion c::main
Value changed:  least#3 from 2147483615 to 0
```

Figure 3: Concrete Δ values for minmax.c.

David Lewis' *counterfactual notion of causality* [21]. 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 depends not on the impossibility of $\neg c$ and e being simultaneously true in any possible world, but on what happens when we alter w *as little as possible* to remove the (possible) cause c. 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. The method for error explanation used in this paper is derived [14] by:

- Replacing Lewis' metrics for possible worlds with distance metrics for program executions.
- Finding a maximally similar execution in which the effect e does not hold, in order to *automatically produce* causes for an error.

Previously [14], the counterexample and successful execution were both *concrete executions* produced by the bounded model checker CBMC [20] and the `explain` tool [15]. Δs between successful and failing runs were presented as changes at the level of the C type system, e.g. `x = 2147483615` vs. `x = 255`.

Many successful software model checking projects, such as SLAM, BLAST, and MAGIC [6, 17, 8] have been based on predicate abstraction [13] and counterexample-guided abstraction refinement (CEGAR) [10]. Rather than model checking a representation of the concrete state-space of a system, these tools check properties of conservative abstractions of programs, and refine the abstractions until either the program is shown to satisfy its specification or a counterexample is generated. The CEGAR framework for verifying a program P with specification *Spec* consists of three steps:

```
Control location deleted (step #5.5):
  10:  most = input2
  {most = [ $0 == input2 ]}
------------------------
Predicate changed (step #5):
  was:  most < least
  now:  least <= most
Predicate changed (step #5):
  was:  most < input3
  now:  input3 <= most
------------------------
Predicate changed (step #6):
  was:  most < least
  now:  least <= most
Action changed (step #6):
  was:  assertion_failure
------------------------
```

Figure 4: Abstract Δ values for minmax.c.

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

2. **Verify:** Check if $A(P) \models Spec$ holds. That is, determine whether the abstracted program satisfies the specification of P. If it does, P must also satisfy the specification, and the program is successfully verified. If $A(P)$ does not satisfy the specification, a counterexample C is generated. C may be *spurious*: not a valid execution of the concrete program P. If C is not spurious, P does not satisfy its specification.

3. **Refine:** If C is spurious, refine $A(P)$ in order to eliminate C, which represents behavior that does not agree with the actual program P. Return to step 1[2].

SLAM, BLAST, and MAGIC use abstraction because concrete state-spaces are often intractably large (or infinite). The reduced state-spaces produced by predicate abstraction have not, typically, been viewed as useful objects for human examination. They are artifacts of the verification process, used for refuting or proving a property of a system and then discarded. These automatically generated abstractions are usually more complex and less intuitive than those produced by humans, and the state-spaces are still generally too large to be presented directly to users.

Abstract error explanation, described in detail in Sections 3, 4, and 5 is a *selective* use of automatically generated predicate abstractions. Even though the abstracted program may not be useful or interesting to a user, the *differences* in predicate values between successful and faulty executions of a program may be very useful and interesting.

As a motivating example, consider the program in Figure 2. The uninitialized values `input1`, `input2`, and `input3` represent nondeterministically chosen inputs in both MAGIC and `explain`. Figure 3 shows the concrete explanation of the error produced by the `explain` tool [14]. Figure 4 shows the explanation produced by MAGIC using abstract Δs[3]. The abstract explanation is produced much

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

[3] Output is slightly simplified for readability.

more quickly[4] and highlights precisely the nature of the error. The counterexample executes the assignment on line 10 (which should change `least` rather than `most`). The most similar successful abstract execution has the same control flow, but does not execute this line, resulting in a change of predicate values — thus avoiding violating the assertion on line 13. The concrete explanation, in contrast, presents changes to input values that do not immediately indicate the nature of the problem. The control flow is unchanged, in part because of the distance metric's comparison of values from non-executed code.

Because software model checkers use conservative abstractions, a non-spurious counterexample *can* be produced from even a very coarse abstraction of a program. However, the search algorithms used typically have no bias in favor of non-spurious counterexamples, and will often first discover a spurious counterexample if an abstraction is too coarse. The difficulty of finding the needle of a non-spurious error path in a haystack of unrealistic behaviors is why predicate abstraction is used in place of the less expensive analysis of control flow alone. That the likelihood of generating a non-spurious counterexample increases as the abstraction more closely captures the real behavior of the system is a primary motivation behind the CEGAR approach. The success of recent software model checking projects has shown that in many cases a "good" abstraction can be found in the territory between the control flow graph (CFG) of a program and its full concrete state-space. This paper proposes that these "good" abstractions, which are provided "for free" by efficient verification tools, can also be used to improve program understanding and provide effective fault localization and debugging assistance. Results produced by an implementation of error explanation for the MAGIC tool [8] are presented in Section 7 as evidence of this claim.

2. RELATED WORK

The most closely related work is the original presentation of error explanation based on distance metrics for program executions [14]. That work was inspired in part by the *transformation analysis* used by the JPF model checker's error explanation facilities [16]. Error explanation via model checking has also been presented for the SLAM model checker [5] and (in more hardware-oriented work) by Jin, Ravi, and Somenzi [19]. The latter techniques do not use a notion of most-similar executions.

In a larger context, Zeller's delta debugging [26], which extrapolates between failing and successful test cases to find more similar executions (with respect to inputs only), suggested the general notion of comparing executions with respect to atomic changes. Delta-debugging for deriving cause-effect chains [25] makes use of program state, but requires additional user choice of comparison points and guarantees neither minimality by a distance metric nor validity of execution traces.

Renieris and Reiss [22] use distance metrics to select similar failing and successful test cases to compare in order to perform fault localization. They also present a quantitative method for evaluating fault localization methods.

This paper extends the concrete distance metric approach [14] to handle abstract executions of programs and properties expressed in LTL, resulting in significant improvements in both the distance metric used and the expressiveness of explanations over earlier work.

3. ABSTRACT ERROR EXPLANATION

The explanation approach shown in Figure 1 can be used by a predicate abstraction and CEGAR based model checker, with suitable changes:

- S is produced by unwinding the transition relation of the *abstract* program $A(P)$ to a finite depth.

The first difference presents challenges when encoding the distance metric. Previous explanation metrics relied on a static single assignment (SSA) [3] encoding of execution. SSA provided a means to avoid the issue of *alignment*, i.e. which states of the successful execution should be compared to which states of the counterexample. In SSA, all executions are represented by a set of assignments to the same variables, and states are in a sense only implicit. SSA introduced a serious drawback, however: the distance metric was computed over values from *all possible control flow paths.* In some cases, a weak explanation was produced because the executions produced very similar values *in portions of the control flow not executed in either the successful execution or the counterexample.* The distance metric presented in Section 4 relies on alignments to avoid this counter-intuitive and questionable comparison over purely "hypothetical" values.

Another issue raised in unwinding the abstract transition relation is the choice of an unwinding depth. In the CBMC approach [14], the original counterexample is produced by bounded model checking, and the bound used to discover a counterexample can be reused in explanation. Furthermore, in CBMC this bound determined an upper limit for unrollings of loops, rather than a total number of steps. The depth used for abstract explanations limits the total number of steps in the successful execution. In practice, using a depth equal to the number of steps in the counterexample plus a small constant factor (to allow for previously untaken control branches) appears to suffice for most programs.

- Solutions to the optimization problem, corresponding to abstract executions, may represent spurious behaviors, requiring multiple iterations to find a non-spurious most-similar successful execution.

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

- The Δs are in terms of different control flow and *predicate values* rather than concrete variable values.

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

Finally, and most importantly, the changes necessary to avoid (or induce) error are presented as Δs of predicates of variables, rather than concrete values. The abstraction refinement process used to find a counterexample automatically, as a side-effect, produces a high-level model of the behavior of the program. With concrete explanations, the user must generalize to the logical causes of error from the overly specific values in the Δs. An abstract (but non-spurious) counterexample or successful execution, however, may represent *many* concrete behaviors of a program. The predicates necessary to find a non-spurious path will provide a description of the *logical* difference between these *sets* of concrete executions. As a simple example, a concrete Δ might indicate that in the counterexample `x` had the value of 47 and in the closest successful execution `x` had the value of 91[5]. An abstract Δ, on the other hand, might state that in the counterexample, `x < y` and in the successful execution, `x >= y`. It is easy to see which explanation is more likely to capture the underlying essence of the erroneous behavior. The abstract Δ not only generalizes the constraint on `x`, but introduces the information that this constraint is relative to `y`. This claim relies on the assumption that in order to find a non-spurious counterexample, refinement will typically have to produce an abstraction that *effectively captures important aspects of a program's behavior.*

4. A DISTANCE METRIC FOR ABSTRACT EXECUTIONS

The distance metric used for explanations is dependent on the representation of program executions. For the MAGIC tool, an execution is an ordered sequence of state-action pairs: $\{(s_0, \alpha_0), (s_1, \alpha_1), \ldots (s_n, \alpha_n)\}$. We will refer to a state-action pair as a step. Each state, s, is composed of a control location $c(s)$ and a predicate valuation $p(s)$. Predicate valuations are vectors of values for the predicates associated with a particular control location. In the MAGIC abstraction framework, different predicates may be tracked at different control locations [9]. For all states with the same control location, however, $p(s)$ will have the same size.

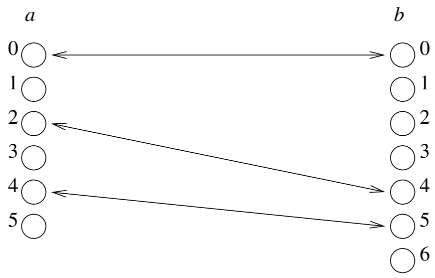
As an example, consider the control location at line 3 in Figure 2, `int least = input1`. In the abstraction used to generate the counterexample, for any state in which $c(s) = 3$ (using the line number to represent the unique control location), $|p(s)| = 3$. The components of $p(s)$ are values for distinct predicates. We write $p_i(s)$ to refer to the ith component of $p(s)$. $p_1(s)$ restricts the relationship of `input1` to `input3`. The possible values are: (`input1 < input3`), (`input3 < input1`), and (`input3 = input1`). The second and third components relate `input1` to `input2` and `input2` to `input3`.

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

4.1 Alignment

The distance metric will be based on comparison of states and actions. An obvious approach would be to compare the

[5]In practice, CBMC tends to produce values that are much larger and harder to follow than 47 and 91, e.g. 32 bit integers with most bits set to 1 if possible.



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

Figure 5: Alignments for executions.

ith step of a with the ith step of b. The two executions, however, may be of different lengths — if any changes in control flow are necessary to avoid the error, this will almost certainly be the case. In order to properly compare a and b, it is necessary to determine an *alignment* [24] mapping steps in a to steps in b. We will define alignment as a relation between elements of a and b, such that if $align(i, j)$, the ith step of a should be compared with the jth step of b:

DEFINITION 4.1 (ALIGNMENT, $align(i, j)$).

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

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

The conditions for alignment require that:

- Steps can only be aligned if they have matching control locations.
- Alignments are *unique*: each step in a is aligned with at most one step in b and vice-versa.
- Alignments preserve ordering: e.g., if i is aligned with j, no earlier step in a may align with a later step in b, and no later step in a may align with an earlier step in b. Visually, this means that alignments cannot cross.

See Figure 5 for an example of the consequences of these constraints. For a given a and b, there may be multiple alignments consistent with these conditions. In the presence of loops, there may be several steps in a or b (or both) with the same control location. There may also be steps in a or b that are not aligned with any step in the other execution. These steps are *unaligned*:

DEFINITION 4.2 (UNALIGNED, $unalign_{a/b}(i/j)$).

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

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

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

A step may be impossible to align — either because no control location in the other execution is matching, or, as in Figure 5, because certain other alignments preclude the conditions from holding. It is important to note, however, that the first condition only requires that *if two steps are aligned*, the conditions must hold. There is no requirement that *if the conditions hold*, two steps must be aligned. The empty relation is always a valid alignment.

4.2 The Distance Metric d

Given a and b we define the distance $d(a, b)$ based on the number of atomic changes (Δs) needed to transform a into b. The first set of Δs is possible alterations to predicate values. Δp is defined first over steps i and j of a and b. The sum of individual step Δps is then used to define the total Δp between two executions:

DEFINITION 4.3 $(\Delta p(i, j, v), \Delta p(a, b))$.

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

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

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

$\Delta p(i, j, v)$ is 1 iff step i and step j are aligned and have differing predicate values for the vth component of their predicate valuations. This comparison is always valid if i and j are aligned since this requires that they share a control location.

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

DEFINITION 4.4 $(\Delta\alpha(i, j), \Delta\alpha(a, b))$.

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

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

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

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

DEFINITION 4.5 $(\Delta c(a, b))$.

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

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

DEFINITION 4.6 (DISTANCE, $d(a, b)$).

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

W_p, W_α, and W_c may reasonably vary, depending on the user's interest in similarity of observable actions, predicate values, and control locations. However, in accordance with the principle that it is best to compare steps whenever possible, it is suggested that W_c be chosen such that it is greater than the maximum possible $\Delta p + \Delta\alpha$ for a single step. In our experimental results, we have uniformly used $W_p = 1$, $W_\alpha = 1$, and $W_c = max(|p(s^a)|) + 2$. With positive values for these weights, d satisfies the standard *nonnegative*, *zero*, *symmetry*, and *triangle inequality* properties of a distance metric [24]. The use of an alignment-based metric, such as d, is not intrinsically tied to abstraction. For any notion of executions based on explicit steps and states, Δp (really Δs: changes to state components other than control location) could be defined over the appropriate elements.

5. FINDING A SUCCESSFUL EXECUTION

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

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

2. For a fixed counterexample a, add to S Boolean variables for all possible alignments of a and b. For each $i < |a|$ and $j < max(|b|)$[6], a variable for $align(i, j)$ may be introduced. Rather than adding all $|a| \times max(|b|)$ variables, we observe that $align(i, j)$ can only be 1 if it is possible for $c(s_i^a)$ to equal $c(s_j^b)$. In many cases, the unwinding of the transition relation will show that this condition cannot be satisfied. The constraints given in Definition 4.1 are also only introduced for alignments not ruled out by the transition relation.

3. Add Δ-*variables* for each possible difference between a and b. For each $i < |a|$, $j < max(|b|)$, and $v < |p(s_i^a)|$, a variable is introduced for: $unalign_a(i)$, $unalign_b(j)$, $\Delta p(i, j, v)$, and $\Delta\alpha(i, j)$. The values are constrained in accordance with the definitions in Section 4. When $j \geq |b|$ (because the successful execution is shorter than the unwinding depth), the associated $align$ variable is forced to be 0, ensuring that Δs variables for steps not in b are also 0.

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

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

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

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

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

Using a shallower unwinding depth may make it possible to find b even if one of these conditions holds. An execution that, if extended, will always become a counterexample will be considered successful if it does not reach an error state before $max(|b|)$ steps.

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

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

In our implementation, we use the pseudo-Boolean solver PBS [2], which combines a fast SAT procedure with special techniques for 0-1 ILP, to efficiently perform step 5. Returning to the motivating example (Figure 2), we observe that it requires 421 Boolean variables to represent the transition relation for $A(P)$ up to a depth of 12 steps (this is sufficient to encode all possible executions of the program). An additional 110 variables are required to represent the possible alignments and Δs. The full SAT instance has 531 variables and 1,841 clauses. The CBMC representation, even without the overhead of alignment variables, has 1,759 variables and 5,747 clauses.

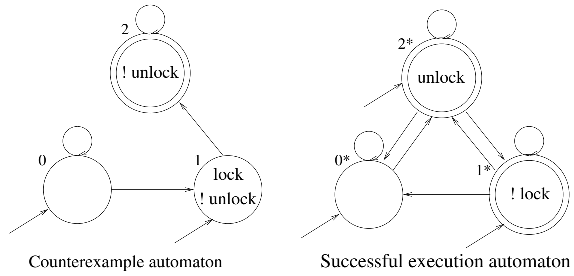
The output shown in Figure 4 is produced by examining the values of the Δ and alignment values in the solution to the ILP problem. In this case, there is one unaligned control location in a, at line 10 (the location of the error). The (aligned) control locations in a and b that follow this change in control flow (lines 11 and 13) differ in predicate values, because the assignment of `input2` to `most` has been performed in a but not in b. The counterexample's final action is an `assertion_failure`, while in b the assertion holds.

6. EXPLAINING LTL PROPERTIES

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

For reachability properties, successful execution is guaranteed by adding constraints such that no error state can appear in b. For an LTL property $A\phi$, a successful execution is a *counterexample* to $A\neg\phi$. A counterexample to $A\phi$ demonstrates that ϕ does not hold for all paths. A counterexample to $A\neg\phi$ demonstrates that ϕ *can hold* for some path. This is not guaranteed to be true (any more than for reachability it is guaranteed that an execution exists that does not reach an error state).

LTL model checking in MAGIC uses the standard approach in which a Büchi automaton for the negation of the property is constructed [12]. Counterexamples are executions in the product automaton (the product of the model and the Büchi automaton for the negation of the property) that contain a cycle that passes through an accepting state. Accepting states in the product automaton are projected from the Büchi automaton. In order to check for successful executions, MAGIC unwinds a Büchi automaton for the property (rather than its negation) along with the transition relation and adds constraints requiring a cycle through an accepting condition to appear in the execution.



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

Figure 6: Successful executions for LTL properties.

The use of a Büchi condition adds an additional possibility to the list of reasons given in step 5 of the procedure in Section 5 for inability to find a successful execution b: the unwinding depth may be insufficient to allow a cycle through an accepting state. For reachability properties, a "spurious"[7] successful execution can sometimes be produced by lowering the unwinding depth. For LTL properties, the unwinding depth may need to be *increased* in order to find a successful execution, but any b that is discovered will represent an infinite behavior, and thus be immune to extension to error[8].

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

6.1 Example of LTL Explanation

As an example of the notion of successful execution used when explaining LTL counterexamples, consider the property $(A)G(lock \Rightarrow F(unlock))$, which requires that on all paths, at all steps, locking requires eventually unlocking. The Büchi automaton on the left in Figure 6 accepts counterexamples to this property: executions in which a lock is acquired (state 1) but not released (state 2). Because state 2 is the only accepting state, counterexamples must have a cycle through program states that do not unlock, and can only reach this state after having locked at least once without unlocking (because state 2 is not an initial state of the automaton).

The automaton on the right accepts *successful executions*:

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

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

```
1   int process () {
2     int x, y, z;
3     z = 0;
4     Lock ();
5     if (x == 0) {
6       if (y == 0)
7         z = 1;
8     }
9     if (y != 0) {
10      z = y;
11    }
12    if (x != 0) {
13      z = 2;
14      Unlock ();
15    }
16    else if (z > 0) {
17      z = 3;
18      Unlock ();
19    }
20  }
21  int main () {
22    while (1) {
23      process ();
24    }
25  }
```

Figure 7: locks.c

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

Figure 8: Abstract Δ values for locks.c.

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

The code in Figure 7 produces a non-spurious counterexample when checked against the LTL formula $(A)G(lock \Rightarrow F(unlock))$ (after 4 iterations of abstraction refinement). It is possible to exit the body of `process` without making a call to `Unlock`[9].

The explanation in Figure 8 describes the conditions in which the error is present: in the counterexample, `y` is `< 1` but *not* equal to 0. In the successful execution, `y > 0`. The error is avoided because the assignment of `y` to `z` on line 10 now ensures that `z` will satisfy the condition at line 16, creating a cycle in which `process` unlocks. The change in `y` has focused our attention on the real problem with this code: the programmer has neglected to take negative values of `y` into account, assuming that `y != 0` implies `y > 0`.

7. EXPERIMENTAL RESULTS

We applied the MAGIC implementation of error explanation to several faulty programs. Table 1 summarizes the results. The results strongly support the claim that finding a non-spurious successful execution will require very few iterations. No benchmark required the addition of even one blocking clause to prevent a spurious successful execution, even though in two cases considerable refinement was required to produce a non-spurious counterexample. Results not shown in the table also supported this claim, although iterations were observed for a few artificial (and unrealistic) examples with LTL properties. The example in Section 6.1 required 4 abstraction refinement iterations and no blocking clauses.

[9]Recall that in MAGIC, uninitialized variables such as `x` and `y` are used to represent nondeterministic choice — perhaps the results of system calls or user input.

7.1 Benchmarks

The examples presented in Table 1 are taken from several sources. The smaller benchmarks were taken from the regression tests for MAGIC (small fragments of Linux kernel code with seeded errors). Additional benchmarks were taken from the C source code of `OpenSSL-0.9.6c`, with seeded errors. In particular, SSL-1 and SSL-2 are from faulty versions of the initial handshake protocol. Section 7.3 presents the SSL-1 explanation (Figure 9) in greater detail as a case study. The final benchmark is the source code for the μC/OS-II [1] real-time multitasking kernel (RTOS) for microprocessors and microcontrollers. The error explained was original to the source code, rather than added for our experiments.

7.2 Evaluation of Fault Localization

The results presented in this section make use of the scoring function (based on program dependency graphs [18]) for evaluating fault localization techniques proposed by Renieris and Reiss [22]. The evaluation method assumes that a correct version of the program is available. 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. A node in the pdg is considered *faulty* if it does not match the correct program. 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. The score assigned reflects how much of a program

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

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

Table 1: Experimental results for MAGIC examples.

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. The pdgs used to evaluate results were computed by CodeSurfer [4].

7.3 SSL Explanation

The specification for the SSL handshake protocol requires that if the `get_client_hello` action is performed, then a `send_server_hello` must be performed, or the server call must return a value of -1. The fault introduced at line 1213 (Figure 10) allows a re-assignment of the return value `ret` (and presents another opportunity for a successful client hello action). In the correct code, the assignment at line 1213 is not present. The counterexample for this property contains 29 states and actions that a user must sort through in order to understand the error. Error explanation produces a successful execution that differs in two actions, two predicates, and two control locations (Δs in Figure 9). The key to the error is indicated as being the faulty assignment at line 1213: if this call fails as the first call did (causing the branch at line 1212 to be taken), the specification is not violated. In the counterexample, the server call succeeds, having failed the first time, and the server returns success without having responded to the received client hello. In the successful execution, the second attempt to get a client hello also fails, and the value of `ret` correctly indicates failure. The error has been localized to line 1213, and the precise conditions under which the faulty assignment will result in erroneous behavior are indicated.

```
Error explanation deltas:
Action changed (step #25):
  was:  get_client_hello
  now:  {ret = [ $0 == -1 ]}
------------------------
Control location deleted (step #26.26):
  1213:  ret = ssl3_get_client_hello ( s  )
  {ret = [ $0 == 1 ]}
------------------------
Predicate changed (step #26):
  was:  ret == 1
  now:  ret == -1
------------------------
Predicate changed (step #27):
  was:  ret == 1
  now:  ret == -1
Action changed (step #27):
  was:  return { $0 == 1 }
  now:  return { $0 == -1 }
------------------------
Control location inserted (step #28):
  final location
------------------------
```

Figure 9: Abstract Δ values for SSL-1.

```
1210  s->shutdown = 0;
1211  ret = ssl3_get_client_hello(s);
1212  if (ret <= 0)
1213    ret = ssl3_get_client_hello(s);
1214    goto end;
1215
1216  got_new_session = 1;
1217  s->state = 8496;
1218  s->init_num = 0;
```

Figure 10: SSL-1 code fragment.

8. CONCLUSIONS AND FUTURE WORK

Any conclusion about the utility of abstract explanation beyond the existential claim that for *some programs and errors* it works very well would be premature. The scoring method proposed by Renieris and Reiss provides a quantitative means for comparing fault localizations; unfortunately, in the absence of competing tools and methods that apply to the same programs and errors, the raw scores are difficult to assess.

8.1 Is Abstract Superior to Concrete?

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

8.2 Is Concrete Superior to Abstract?

The result for μC/OS-II in Table 1 is startling: the explanation is of no value for localization! Inspection shows that the unwinding depth allows the system to avoid the consequences of a missing return statement by delaying the calls that expose the error. The counterexample fails almost immediately after taking the branch guarding the location of the missing return. Because the counterexample fails immediately after error, any successful execution will be forced to insert new control locations. The distance metric, unfortunately, ensures that it is "better" to introduce irrelevant steps that delay the unlock call that exposes the error than to avoid the branch that ensures failure (which requires that even more new control locations be added and forces a costly unalignment after the branch). CBMC [20] and `explain` [15], in contrast, produce the optimal explanation, which avoids taking the branch guarding the missing return location. With static single assignment [3], the change for the untaken branch is represented by a pair of Δs (one for the condition and one for the control flow change) and there is no need to insert new control locations. For errors best explained purely in terms of control flow, concrete explanation is just as expressive as abstract execution.

Although MAGIC is capable of model-checking the TCAS examples [23] used in the original presentation of distance metric based explanation [14], it fails to produce explanations for the errors discovered. The TCAS counterexamples are very lengthy and require many alignment variables. To produce non-spurious executions, numerous predicates must be introduced at most control locations in the program, although the values on which the predicates are based are only assigned to at the beginning of execution. The SSA unwinding used by CBMC only has to produce constraints for these inputs at possible assignment or branching points. Because the TCAS code is essentially a computation of a function with a very small range (3 values) from a large set of unaltered inputs, CBMC and `explain`, despite using full 32-bit integers in place of abstract values, produce a much simpler 0-1 ILP problem than MAGIC.

8.3 Choosing a Distance Metric

It is probably incorrect to ascribe these differences to concrete vs. abstract explanation. A tool using SSA with abstract assignments would likely match or improve upon the results produced by concrete explanation[10]. To our knowledge, no tool supporting SSA and predicate abstraction currently exists[11]. For the time being, for some programs, CBMC and `explain` may be the best model checking tools for error explanation. It may be that the counter-intuitive SSA-based metric is, in fact, better for some errors than the alignment-based metric used for abstract explanations in this paper. Ideally, the choice of an SSA or alignment based distance metric is orthogonal to the use of an abstract state-space.

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

8.4 Conclusions

It is unlikely, explanation being at heart a *psychological* notion, that any one approach to error explanation can be *proven* to be optimal or even "correct." That said, the distance metric approach to explanation [14] is based on David Lewis' intuitively appealing notion of causality [21] and provides an effectively computable notion of explanation. The method was first applied to concrete executions of programs, using a somewhat counter-intuitive distance metric influenced by hypothetical values computed by unexecuted code. The approach can be generalized to apply to abstract executions, use a more intuitive distance metric, and explain Linear Temporal Logic property violations. Experimental results demonstrate the utility of abstract explanation, but also indicate that the original SSA-based metric and tool have some advantages over the implementation of abstract explanation for MAGIC.

The most interesting lesson to be drawn from abstract explanation is that the predicate abstractions introduced to model checking in order to combat the state-space explosion problem are also useful for improving program understanding. The fact that each abstract execution potentially represents many counterexamples or successful executions provides an automatic generalization to the logical causes of an error. It should be possible to exploit this generalization of program behaviors (or the production of a set of predicates that are relevant to a given property, etc.) for other program understanding techniques. Rather than viewing the abstract state-spaces automatically produced by software model checkers as disposable artifacts of verification, we must at least consider the possibility that *the abstractions themselves are valuable by-products that can be mined for information.*

8.5 Future Work

Numerous directions for future error explanation research are open. The TCAS and μC/OS-II results indicate that predicate abstraction plus SSA-form BMC might be a fruitful combination for error explanation. Abstraction makes the presence of irrelevant Δs in an explanation less likely but does not fully eliminate the need for causally-aware slicing. Adapting the Δ-slicing method [14] used with concrete explanations to an alignment-based distance metric poses interesting challenges. A more extensive empirical study of explanation approaches and distance metrics is in order, as are user studies to discover how genuinely useful explanations are for debugging.

9. REFERENCES

[1] `http://www.ucos-ii.com/`.

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

[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.

[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] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.

[7] 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.

[8] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering*, pages 385–395, 2003.

[9] S. Chaki, E. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 19–34, 2003.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, pages 154–169, 2000.

[11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[12] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, 1995.

[13] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, pages 72–83, 1997.

[14] A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.

[15] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with `explain`. In *Computer-Aided Verification*, 2004. To appear.

[16] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[18] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *International Conference of Software Engineering*, pages 392–411, 1992.

[19] 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.

[20] 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, 2004.

[21] D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.

[22] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, pages 30–39, 2003.

[23] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.

[24] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison Wesley, 1983.

[25] A. Zeller. Isolating cause-effect chains from computer programs. In *Foundations of Software Engineering*, pages 1–10, 2002.

[26] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.