Whodunit? Causal Analysis for Counterexamples*

Chao Wang[1], Zijiang Yang[2], Franjo Ivančić[1], and Aarti Gupta[1]

[1] NEC Laboratories America
4 Independence way, Princeton, NJ 08540, USA
{chaowang,ivancic,agupta}@nec-labs.com
[2] Department of Computer Science
Western Michigan University, Kalamazoo, MI 49008, USA
zijiang.yang@wmich.edu

Abstract. Although the counterexample returned by a model checker can help in reproducing the symptom related to a defect, a significant amount of effort is often required for the programmer to interpret it in order to locate the cause. In this paper, we provide an automated procedure to zoom in to potential software defects by analyzing a single concrete counterexample. Our analysis relies on extracting from the counterexample a syntactic-level proof of infeasibility, i.e., a minimal set of word-level predicates that contradict with each other. The procedure uses an efficient weakest pre-condition algorithm carried out on a single concrete execution path, which is significantly more scalable than other model checking based approaches. Unlike most of the existing methods, we do not need additional execution traces other than the buggy one. We use public-domain examples to demonstrate the effectiveness of our new algorithm.

1 Introduction

One of the major advantages of model checking [5, 22] is the production of a counterexample when verification fails. However, the counterexample only shows a symptom of the defect; users still need to spend a considerable amount of time scrutinizing the potentially lengthy trace in order to find the cause of the failure. In principle, an observable *failure* is caused by a *defect* in the code after the infection propagates through a sequence of relevant statements (also called the *infection chain* [26]). In this paper we present an efficient procedure for identifying this infection chain, i.e., the cause-effect segments from the given counterexample that eventually lead to a failure.

The problem of fault localization for software programs has been the attention of recent research. Testing based methods [16, 23] rely on availability of a good test suite; they compare a large set of failing executions with successful ones to find out points in the failing executions that may (statistically) be responsible for the failure. Usually, they assume that a large number of successful executions are available to be chosen as a comparison to failing executions.

* A *whodunit*, for "who done it?", is a plot-driven variety of detective story in which the reader is provided with clues from which the identity of the perpetrator of the crime may be deduced. Examples are the Sherlock Holmes stories by Conan Doyle.

S. Graf and W. Zhang (Eds.): ATVA 2006, LNCS 4218, pp. 82–95, 2006.

Model checking based methods [3, 11, 10] seek additional execution traces by deploying the same model checker again with additional constraints. A representative approach is the work by Groce *et al.* [10], which uses a SAT based bounded model checker to produce the counterexample, and then uses a pseudo-Boolean constraint solver (called PBS [2]) on a constrained version of the same bounded model checking instance to search for a "closest" successful execution trace. The difference between these two traces is considered as potential cause of failure. A drawback of model checking based method is their limited scalability in dealing with large systems or long counterexamples. Furthermore, the difference between a successful run and the counterexample does not always provide a good explanation of the failure.

Delta debugging as in [26, 6] uses automated testing to isolate relevant variables and values of the program by systematically narrowing the state difference between a passing run and a failing run. Note that this method also requires alternative runs in addition to the given counterexample. The method is based on *trial and error*, by assessing the outcome of altered executions to determine whether a change in the program state makes a difference in the test outcome. The alternative runs also determine the quality of results that Delta debugging can infer: a variable can be isolated as a failure cause only if its value differs in the two runs. This method is purely empirical, which is quite different from methods based on formal/static analysis. As is stated in [26], Delta debugging may require a large number of tests to find a difference that can no longer be narrowed.

A problem closely related to fault localization is program repair, which has been studied in [25, 15, 9]. They take the view that a system component may be responsible for a failure if replacing it by an alternative can make the system correct. The program repair problem is cast into a two-player reachability game on a finite-state machine extended from the system, by assuming any component can be replaced by an arbitrary function in terms of inputs and the system state. An algorithm that computes a winning strategy for the game effectively solves the program repair problem. However, program repair in general is significantly more costly than standard model checking.

In general, accurately locating the faulty code requires a complete specification of the system behavior (the same argument also holds for automated program repair). Unfortunately, such specifications are often missing in realistic software development settings. Without a complete specification, it is not possible to determine whether a particular line in the code is faulty or not. What can be done (a view shared by many previous works as well as this paper) is to locate portions of the program where a defect may reside, and to provide an explanation how a defect triggers the failure. In this paper, we try to identify the infection chain in the failed execution path, with the belief that the defect resides in one of the chain segments.

The new causal analysis algorithm presented in this paper differs from previous works in that: (1) it does not require additional successful or failing executions other than the given counterexample; (2) it does not use expensive model checking or constraint solving algorithms. Instead, we use a path-based syntactic-level weakest precondition computation algorithm to aid the analysis. It produces a concise proof of infeasibility for the given counterexample, which is a minimal set of word-level predicates extracted from the failed execution that explains why the execution fails. Since

the pre-condition computations are cheap and are restricted to a single execution path (less chance to blow up), our method is significantly more scalable than other model checking based methods.

2 Motivating Examples

We provide two small examples to illustrate a shortcoming of some existing fault localization methods. The main assumption of the method in [10] is that, one can locate the defect by comparing a successful run with a buggy run. A similar assumption is also made in Delta debugging [26] although automatic testing is used to get alternative runs. The unique feature of [10] is defining a distance metric with respect to the given counterexample and then searching for a "closest" successful run with respect to that metric. Since a program is deterministic, the only change they make in searching for a successful run is the input values. By changing the input values and minimizing the difference caused by these changes, they try to find an execution trace that does not violate the property. In other words, they try to find ways to dodge the observable failure instead of fixing it.

```
   find_max (x1, x2, x3)
   {
1:  max = x1;
    ...
2:  if ( max <= x2 )
3:     max = x2 ;
    ...
4:  if ( max >= x3 )
5:     max = x3 ;
    ...
6:  assert ( max >= x1 ) ;
7:  assert ( max >= x2 ) ;
8:  assert ( max >= x3 ) ;
   }
```

(a) the maximum of three inputs;

```
   compute_diff (x1, x2)
   {
1:  if ( x1 != x2 ) {
2:     if ( x1 < x2 )
3:        diff = x1 - x2 ;
4:     else
5:        diff = x2 - x1 ;
    }
6:  else {
7:     diff = 0 ;
       ...
    }
8:  assert ( diff > 0 ) ;
   }
```

(b) the difference of two inputs;

Fig. 1. Two examples to illustrate fault localization algorithms

First, we note that it is not always possible to dodge the failure by merely changing input values. When a failure exists regardless of any particular input value, the algorithm in [10] fails since there is no valid solution for the constraint solver to optimize. Even if a successful run can be found, the difference between the two runs does not necessarily offer enough hints to locate the defect. This can be illustrated by the C program in Figure 1-(a), which is supposed to find the maximum of three inputs. The input `(0,1,0)` can trigger an execution that fails the assertion check at Line 7. The assertion failure is caused by Lines 4-5 where the conditional expression should have been different.

Table 1 lists the variable assignments at different execution steps for the original counterexample and a closest successful run. Each row in the table shows the names of variables or conditional expressions, their program locations (`max @3` corresponds to Line 3), their values, and the distance according to the metric in [10]. Since there are only two different assignments: `x2 @0` and `max @3`, it would classify Line 3 as cause of the failure. However, both Line 3 and Line 2 (the guard of Line 3) are correct, and the real error is in Lines 4-5.

Table 1. Counterexample and successful executions for `find_max`

variables/predicates	variable/predicate valuations in		distance
	counterexample	a successful run	
x1 @ 0	0	0	
x2 @ 0	1	0	1
x3 @ 0	0	0	
max @ 1	0	0	
(max<=x2) @ 2	true	true	
max @ 3	1	0	1
(max>=x3) @ 4	true	true	
max @ 5	0	0	
(max>=x1) @ 6	true	true	
(max>=x2) @ 7	false	true	

Our second example, Figure 1-(b), is a program to compute $|x1 - x2|$ when the two inputs have different values. There is a bug at Line 2 and the correct version should be `(x1 > x2)`. A counterexample can be produced with the input `(0,1)`, under which the program goes through Lines 1-3 and 8. Since there is no way to avoid the failure as long as `(x1 != x2)`, a closest successful run would be with the input `(0,0)`. The successful run goes through lines 6-8. As a result, all lines within the if-branch and else-branch are different between the two runs, and would be marked as potential causes of the failure.

In these two examples, the inaccuracy of the algorithm is due to its way of analyzing causality, which we believe is very different from the actual debugging practice by programmers. Given an execution trace exhibiting some erroneous behavior, a programmer will not keep changing the input values until the bug disappears. Instead, the programmer will keep the same input and try to find out *how this particular input value leads to the failure*. When there is an assertion check in the code, it often means that the program is expected to work at this location all the time, regardless of which path it has taken to reach here and regardless of the input values. Therefore, we choose to focus on the given counterexample and tackle the problem from a different angle; in particular, we want to explain why this particular run fails.

3 Preliminaries

We provide some needed notations before introducing the definition of transforming statement and the notion of minimal proof of infeasibility, which are the foundation of

our counterexample causal analysis algorithm. We focus on the class of failures that can be captured using assertions. In a C program, for instance, `assert(!crash)` represents the property that `crash` should never be true at this program location. A counterexample is a particular execution path of the program that violates the assertion.

An *execution path* $\pi = s_1, s_2, \ldots$ is a sequence of simple program statements, each of which has one of the following types:

- assignment statement `s: v := e`, where v is a variable and e is an expression; we assume that the statement has no side-effects.
- branching statement `s: assume(c)`, where c is a predicate. It may come from statements like `if(c)...else` or successfully executing of `assert(c)`.

Given an execution path π, we use $\pi^i = s_i, \ldots$ to represent the suffix starting from $i \geq 1$; we also use $\pi^{i,j} = s_i, \ldots, s_j$ to represent the segment between i and j.

A *counterexample* is a tuple $\langle I, \pi^{1,n} \rangle$, where I is an input valuation and $\pi^{1,n}$ is the corresponding execution path leading to failure at s_n:`assert(c)`. A counterexample is a concrete execution of the program. Given a set I of initial values to input variables, the execution of a deterministic program is completely fixed. It is easy to map a counterexample back to an execution path π. Complex data structures and language constructs do not pose a problem, because everything is completely determined in a concrete trace. For pointers, the locations that they point to are fixed at every step; similarly for arrays, the indexes are also fully determined. Since a counterexample is of finite length, recursive functions and statements involving data in dynamically allocated memory can be rewritten into simple but equivalent statements.

The set of input variables of the program induces an *input space*, in which each particular input valuation corresponds to a point. In general, an execution path $\pi^{1,n}$ corresponds to more than one counterexamples, each of which maps to a distinct point in the input space. The input subspace related to $\pi^{1,n}$ can be represented by the *weakest pre-condition* of $\neg c$ with respect to $\pi^{1,n-1}$; that is, the weakest condition before $\pi^{1,n-1}$ that entails the failure at s_n. The definition of weakest pre-condition is given below, where we use $f(V/W)$ to denote the simultaneous substitution of W with V in function $f(W)$.

Definition 1 (cf.[8]). *Given $\pi^{i,j} = s_i, \ldots, s_j$ and a propositional formula ϕ, the weakest pre-condition of ϕ with respect to $\pi^{i,j}$, denoted by $WP(\pi^{i,j}, \phi)$, is defined as follows,*

- *For a statement `s: v = e`, $WP(s, \phi) = \phi(e/v)$;*
- *For a statement `s: assume(c)`, $WP(s, \phi) = \phi \wedge c$;*
- *For a sequence of statements `s1; s2`, $WP(s1 : s2, \phi) = WP(s1, WP(s2, \phi))$.*

Weakest pre-condition computation has been used in several recent predicate abstraction algorithms [20, 13, 14], where it is applied to an infeasible counterexample in the abstract model in order to find relevant predicates that can eliminate the trace in the refined model. However, in this paper the purpose of computing weakest pre-conditions is quite different, since the counterexample here is a feasible trace in the concrete program, as opposed to an infeasible trace in an abstract model.

We use this computation to find a minimal set of conditions for the program to stay on the same path without violating the assertion. The result is a set of predicates that should hold at each step of the path. By comparing how these predicates contradict with each other and with the given set of input values, we can locate part of the original code responsible for this particular assertion failure.

4 Analyzing the Infection Chain

Given a counterexample $\langle I, \pi^{1,n} \rangle$, we identify a set of statements in $\pi^{1,n}$ constituting the infection chain, i.e., cause-effect segments that lead eventually to a failure in s_n. We accomplish this by computing $WP(\pi^{1,n-1}, c)$. According to the definition, the weakest precondition over a path is a conjunction of predicates. That is,

$$WP(\pi^{i,j}, c) = c' \wedge (c'_1 \wedge c'_2 \ldots \wedge c'_k) \ ,$$

where c' is transformed from the given formula c through (possibly transitive) variable substitutions, and each c'_l is transformed from a condition in s_l: `assume(`c_l`)` such that $i \leq l \leq j$. More formally, given a formula ϕ, we use ϕ' to denote the formula in WP that is transformed from ϕ. The definition is transitive in that both $\phi' = \phi(e/v)$ and $\phi'(e_2/v_2)$ are *transformed formulae* from ϕ.

Definition 2. *A* transforming statement *of ϕ is an assignment statement* `s: v = e` *such that variable v appears in the transitive support of formula ϕ.*

For example, statement `s1:x = y+1` is a transforming statement of ϕ:`(x > 0)`, since $WP(s1, \phi)$ produces ϕ':`(y+1 > 0)`; statement `s2:y = z*10` is also a transforming statement of ϕ, since $WP(s2, \phi')$ produces `(z*10+1 > 0)`. During weakest precondition computations, only assignment statements can transform an existing conjunct c into a new conjunct c'. Branching statements can only add new conjuncts to the existing formulae, but cannot transform them. Given an execution path $\pi^{i,j} = s_i, \ldots, s_j$, we use the subset $TS(\pi^{i,j}, c) \subseteq \{s_i, \ldots, s_j\}$ to denote the transforming statements for the predicate c. Transforming statements are the foundation of our causal analysis algorithm.

For a failed execution path $\pi^{1,n}$ where the statement s_n is `assert(c)`, the three pre-conditions, $WP(\pi^{1,n-1}, true)$, $WP(\pi^{1,n-1}, c)$, and $WP(\pi^{1,n-1}, \neg c)$, have the following relationships:

1. $WP(\pi^{1,n-1}, true) = WP(\pi^{1,n-1}, c) \vee WP(\pi^{1,n-1}, \neg c)$;
2. $WP(\pi^{1,n-1}, c) \wedge WP(\pi^{1,n-1}, \neg c) = \emptyset$;

This is illustrated by Figure 2. Also note that the three pre-conditions share a common subformula $(c'_1 \wedge \cdots \wedge c'_k)$, which is the same as $WP(\pi^{1,n-1}, true)$. We now introduce the notion of proof of infeasibility.

Theorem 1. *Given a counterexample $\langle I, \pi^{1,n} \rangle$, we have $I \subseteq WP(\pi^{1,n-1}, \neg c)$, meaning that*

$$I \wedge WP(\pi^{1,n-1}, c) = \emptyset \ .$$

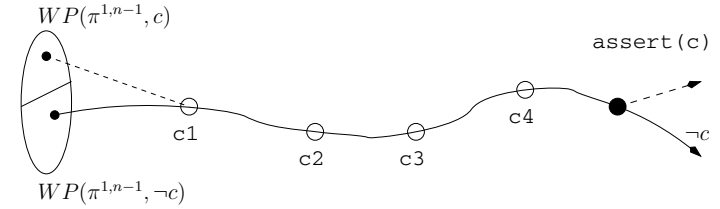


Fig. 2. Partitioning of the input subspace $WP(\pi^{1,n-1}, true)$

The input valuation I is a conjunction set of predicates $I = I_1 \wedge \ldots \wedge I_m$, where I_i, for instance, can be the valuation of an input variable `x = 10`. Given that

$$(I_1 \wedge \ldots \wedge I_m) \wedge c' \wedge (c'_1 \wedge \ldots \wedge c'_k) = \emptyset \ ,$$

there exist a minimal subset of conjuncts in I and a minimal subset of conjuncts in $WP(\pi^{1,n-1}, c)$, denoted by I_{sub} and WP_{sub}, respectively, such that $I_{sub} \wedge WP_{sub} = \emptyset$. The point here is that only some conjuncts are responsible for the empty intersection (which is the reason of the assertion failure). We call $I_{sub} \wedge WP_{sub}$ a minimal *proof of infeasibility*.

In general, one can find a minimal set of contradicting predicates as follows,

1. initialize $I_{sub} = I$ and $WP_{sub} = WP(\pi^{1,n-1}, c)$;
2. minimize WP_{sub} by dropping each conjunct c'_i in WP_{sub}, and then checking whether $I_{sub} \wedge WP_{sub} = \emptyset$: if the result remains empty, drop c'_i permanently; otherwise, add it back.
3. minimize I_{sub} by dropping each I_i in I_{sub}, and then checking whether $I_{sub} \wedge WP_{sub} = \emptyset$: if the result remains empty, drop I_i permanently; otherwise, add it back.

For this particular application, however, we note that WP_{sub} always contains c'. This is because other conjuncts c'_i come from `assume` statements and are all consistent with I, but c' comes from the failed assertion condition c. Therefore, we can skip the test for c' when minimizing WP_{sub}. It is often the case that c' contradicts to some other conjuncts in WP and I_{sub} is not needed in the proof of infeasibility. However, if WP does not have conflicting conjuncts by itself, then a minimal proof is of the form $I_{sub} \wedge c'$.

The intuition behind this definition is that: given a concrete counterexample, our proof of infeasibility provides a succinct explanation about the cause of the assertion failure at s_n. The choice of computing a syntactic-level proof of infeasibility, as opposed to other forms including interpolation [19], is due to the need of eventually mapping the proof back to the source code program. In our case, the explanation can be mapped back to the source code by finding the transforming statements with respect to predicates in WP_{sub}.

5 The Causal Analysis Procedure

In this section we present the entire causal analysis procedure and then explain how it can be applied to the two working examples.

5.1 The Algorithm

Given a counterexample $\langle I, \pi^{1,n} \rangle$, we compute the weakest precondition $WP(\pi^{1,n-1}, c)$ by starting backward from c. (Recall that c comes from the failed `assert(c)` at s_n.) During this process, we also record in $TS(c)$ all transforming statements of c. At each pre-condition computation step, we check whether the intermediate result $WP(\pi^{i,n-1}, c)$ is empty. There are two possibilities:

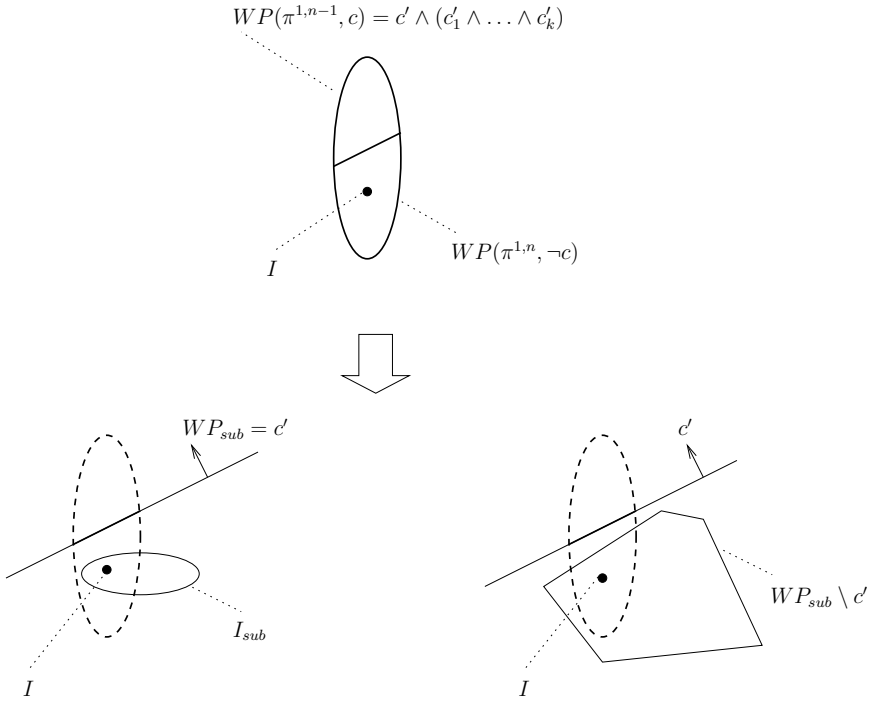
- there exists an index $1 \leq i < n$ such that $WP(\pi^{i,n-1}, c) = \emptyset$;
- no such index exists and the computation of $WP(\pi^{1,n-1}, c)$ completes.

We consider the first case as a special case, since it implies emptiness of $WP(\pi^{1,n-1}, c)$ and hence emptiness of its intersection with I.

1. In the first case, we take the set of conjuncts in $WP(\pi^{i,n-1}, c)$ right after it becomes empty and compute a minimal subset WP_{sub}. We consider all conjuncts in WP_{sub} as responsible for triggering the failure. In the source code, we mark only transforming statements in $\{s \mid s \in TS(\phi)$ such that $\phi' \in WP_{sub}\}$ as explanation of the failure.
2. In the second case, we take all conjuncts in I and $WP(\pi^{1,n-1}, c)$ and compute a minimal proof $I_{sub} \wedge WP_{sub}$. We consider I_{sub} and all conjuncts in WP_{sub} as responsible for triggering the failure. As is illustrated in Figure 3-(a), WP_{sub} has only one subformula in this case; that is, $WP_{sub} = c'$. In the source code, we mark only transforming statements in $TS(c)$ as explanation of the failure. The marked source code shows how I_{sub} leads to the failure at s_n:`assert(c)` through the execution of the transforming statements.

The result in the first case is a stronger condition for explaining the failure—an empty $WP(\pi^{i,n}, c)$ means that any execution path with the same suffix $(s_i, ..., s_{n-1})$ would fail at s_n. As is illustrated in Figure 3-(b), the relevant input subspace in this case becomes $WP(\pi^{i,n-1}, true)$, which is large than $WP(\pi^{1,n-1}, true)$ in general. (In the figure, with a little abuse of notation, we have used $WP_{sub} \setminus c'$ to represent the removal of c' from the set of conjuncts in WP_{sub}.) By focusing on WP_{sub} only, we can explain the cause of failure common to all these execution paths.

Our algorithm aims at explaining why the given execution path fails by focusing on the infection chain (i.e., set of transforming statements) leading to the failure. We do not attempt to answer the question *which segment in the infection chain contains the faulty code* or *how to fix the bug by changing a particular segment*. We believe that the latter two problems in general require a relatively complete specification of the intended program behavior in order for them to be solved effectively. Unfortunately, complete specifications are often missing in realistic software development settings.



(a) When $I_{sub} \wedge WP_{sub} = \emptyset$

(b) $WP_{sub} = \emptyset$ before reaching s_1

Fig. 3. The minimal proof of infeasibility. WP_{sub} consists of a subset of conjuncts of $WP(\pi^{i,n-1}, c)$, and thus $WP(\pi^{i,n-1}, c) \subseteq WP_{sub}$. Similarly, $I \subseteq I_{sub}$.

5.2 The Working Examples

We now demonstrate that our new method can produce better results than existing algorithms. We first apply the new algorithm to `find_max` in Figure 1-(a). We start the weakest pre-condition computation with the failed assertion condition `(max >= x2)`. The sequence of intermediate results are listed in Table 2, where the first column gives the line numbers, the second column gives the subformulae whose conjunction is WP, the third column indicates whether the statement belongs to $TS(max \geq x2)$; for instance, a "yes" for Line 5 means that s_5 `:max = x3` is a transforming statement of the predicate $(max \geq x2)$. The last column shows whether the weakest pre-condition is an empty set.

Table 2 shows that the weakest pre-condition becomes empty only after the intersection with initial input values `x1=0, x2=1, x3=0`. The minimal subset is

$$(x2 = 1) \wedge (x3 = 0) \wedge (x3 \geq x2)$$

In the source code, we highlight all transforming statements of predicates in $TS(max \geq x2)$ as responsible for the failure. Thus, Line 5 is marked as explanation of the

Table 2. Analyzing the cause of failure in `find_max`

Line	Predicates in the WP	in $TS(max \geq x2)$	empty WP?
7	(max≥x2)		
6	max≥x1, (max≥x2)		
5	x3≥x1, (x3≥x2)	yes	
4	max≥x3, x3≥x1, (x3≥x2)		
3	x2≥x3, x3≥x1, (x3≥x2)		
2	max≤x2, x2≥x3, x3≥x1, (x3≥x2)		
1	x1≤x2, x2≥x3, x3≥x1, (x3≥x2)		
0	0≤1, 1≥0, 0≥0, (0≥1)		empty

Table 3. Analyzing the cause of failure in `compute_diff`

Line	Predicates in WP	in $TS(diff > 0)$	empty WP
8	(diff>0)		
3	(x1-x2>0)	yes	
2	x1<x2, (x1-x2>0)		empty
1	x1≠x2, x1<x2, (x1-x2>0)		empty
0	0≠1, 0<1, (0-1>0)		empty

failure cause; this is significantly more accurate than the algorithm of [10] (which instead would mark Line 3).

Next, we apply our algorithm to `compute_diff` in Figure 1-(b). We start weakest precondition computation with the failed assertion condition `(diff > 0)`. The sequence of intermediate results are given in Table 3. The statement in Line 3 transforms the initial predicate into `(x1-x2 > 0)`, which then contradicts to `(x1 < x2)`, the new predicate added at Line 2. Since $WP(\pi^{i,n-1}, (diff > 0)) = \emptyset$, we compute the minimal proof of infeasibility at this point. The result is as follows,

$$(x1 < x2) \wedge (x1 - x2 > 0) \ .$$

In the source code, we mark all transforming statements in $TS(diff > 0)$ and $TS(x1 < x2)$, as well as the source statement of c'_1, which is `assume(x1 < x2)`. Thus, our algorithm reports Lines 2-3 of Figure 1-(b) as the failure cause. The fact that weakest pre-condition becomes empty in the middle of a counterexample strongly indicates that the error may happen in the common suffix. In contrast, the algorithm in [10] is ineffective on this example since the first `else`-branch is the only possible successful run; as a result, it would mark the code in both branches.

6 Further Discussion

6.1 About Delta-Debugging

The notion of cause transition in [26, 6] is similar to transforming statements in our method. A cause transition points to the connecting points of execution path where a

change of the previous state would lead the execution to a different branch. To find the defect, the method in [26, 6] traces forward in the program to identify the chain of cause transitions, by running an additional set of tests. Their idea of empirically comparing state difference between successful and failing runs is significantly different from ours; the notion of minimal proof of infeasibility is not used. In our method, the set of predicates produced by weakest pre-condition computations at each individual program location represents an *abstract* program state.

As is pointed out in [6], for each infection $\mathcal{F}'$, there is either an earlier infection $\mathcal{F}$ that causes $\mathcal{F}'$, or no earlier infection—in which case $\mathcal{F}'$ is the defect. Therefore, given the observable failure, tracing back the infection chain requires two proofs:

1. to prove that $\mathcal{F}$ and $\mathcal{F}'$ are "infected";
2. to prove that $\mathcal{F}$ causes $\mathcal{F}'$.

Without a complete specification, in general it is not possible to determine whether a program state is infected (and therefore not possible to determine where the very first infection is). However, we note that the actual defect ought to be in one of the chain segments. By our definition, each transforming statement of the failed assertion condition (i.e., the last infection) is a proof that a previous infection $\mathcal{F}$ causes $\mathcal{F}'$.

6.2 About Distance Metric in Explain [10]

Compared to the `explain` tool in [10], our method answers the question why a specific execution path fails, instead of *how the failure can be avoided*. In general, it is hard to answer the latter question in a useful way unless one has a complete specification. The reason is that there can be multiple ways of avoiding a particular failure, each of which corresponds to a different program intent. (Program intent in principle can only be provided by human.)

For example, the failed property in our first working example is, "all runs that go through Lines 1-8 should pass the assertion check at Line 9." This is only a partial specification of the program behavior. When being represented in linear temporal logic, this property is of the form $\mathbf{G}(P \rightarrow Q)$, where

- P: the execution actually goes through lines 1-8;
- Q: the execution fails assertion check at line 9.

A counterexample of this property is an execution on which $(P \wedge \neg Q)$ holds. One can avoid this particular failure by satisfying $\neg(P \wedge \neg Q)$ under the same input condition, which is the same as

$$(\neg c'_1 \vee \ldots \vee \neg c'_k) \vee c' \quad .$$

Unfortunately, any one of the disjunctive subformulae entails the entire formula. Note that in our causal analysis, we focus only on c' (i.e., the assertion check at s_n should pass) by assuming that P holds.

More Related Work: The property $P \rightarrow Q$ has also been studied in the context of vacuity detection in [4, 18, 21], where $P \rightarrow Q$ is said to be vacuously satisfied whenever P is false. This is because Q is often the property that the user intends to check, while P is only a pre-condition. We believe that the same argument also applies to counterexample explanation or fault localization.

6.3 About Dynamic Slicing

Our method is also different from dynamic slicing [17, 1, 12], which is a variant of program slicing with the restriction to an execution path. Although dynamic slicing often gives more accurate data dependencies between variables than normal static analysis, it is inferior to our weakest pre-condition based causal analysis in explaining cause of failed assertions. Consider the following example,

```
1:   x1 = 10;
2:   x3 = 5;
3:   ...
4:   x2 = 0 ;
5:   if ( x1 == 10) {
6:     x2 = x2 * x3 ;
7:     ...
8:   }
9:   else {
10:    x2 = x2 * x4;
11:  }
12:  assert( x2 != 0 );
```

If only Lines 1-8 and 12 are executed, dynamic slicing with respect to line 12 can remove the irrelevant variable $x4$, which could not have been removed by static program slicing without knowing which path will be executed. However, it could not remove variable $x1$ since whether line 6 gets executed or not depends on the condition `(x1==10)`. In contrast, our analysis algorithm would remove `(x1==10)` because it is not in the minimal proof of infeasibility (lines 4, 6, and 12).

7 Experiments

In this section, we apply our procedure to public benchmark programs in the Siemens suite [24]. The Siemens suite provides a set of C programs, each of which has a number of test vectors as well as a correct version of the program. The examples we used in this study are from the TCAS (Traffic Collision Avoidance System) example, which is a model of the aircraft conflict detection system. The assertion checks (or properties) used in our experiment originated from a previous study using symbolic execution [7].

A faulty TCAS version differs from the correct one in Line 100, where the relational operator $>$ is used when it should be $\geq$.

```
  result = !( Own_Below_Thread()) || ((Own_Below_Threat())
           && (!(Down_Separation >= ALIM()))) ;   // correct
---
  result = !( Own_Below_Thread()) || ((Own_Below_Threat())
           && (!(Down_Separation >  ALIM()))) ;   // buggy
```

The counterexample used in our study has been generated from a software model checker, and it has a length of 90. (The counterexample may also come from other software testing tools—our causal analysis procedure would be equally applicable as long

as the counterexample is a fully determined execution path.) When our causal analysis procedure is used, the weakest pre-condition becomes empty right after the computation passes Line 100. This gives a succinct explanation of the actual failure down the stretch.

We compare our results with the previous results in [10]. Given the same counterexample, the initial explanation by this previous algorithm was not particularly useful. In fact, their tool dodged the failure by making the antecedent of the implication false. As is stated in [10], to coerce it into reporting a more meaningful explanation, they had to manually add some additional constraints (e.g. the antecedent should not be true). After that, their tool reports a similar result as ours. We argue, however, that this kind of manual intervention requires the user to have a deep understanding of the counterexample as well as the software program. In contrast, our method does not need additional hints from the programmer, but still achieves the same accuracy as [10] combined with manually provided assumptions.

8 Conclusions

We have addressed the problem of locating the failure cause of a program given a concrete counterexample trace, and demonstrated the effectiveness of our approach using several examples. Our automated procedure relies on the minimal proof of infeasibility to generate succinct failure explanations. Since the computations are performed at the syntactic level and are restricted to a single concrete path, there is no foreseeable difficulty in applying it to long counterexamples in large production-quality software. As future work, we will pursue a more detailed experimental study of the proposed technique and comparison with existing tools.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[2] F. A. Aloul, B. D. Sierawski, and K. A. Sakallah. Satometer: How much have we searched? In *Proceedings of the Design Automation Conference*, pages 737–742, New Orleans, LA, June 2002.

[3] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Symposium on Principles of Programming Languages (POPL'03)*, pages 97–105, January 2003.

[4] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Computer Aided Verification (CAV'97)*, pages 279–290. Springer, 1997. LNCS 1254.

[5] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.

[6] H. Cleve and A. Zeller. Locating causes of program failures. In *ACM/IEEE International Conference on Software Engineering*, 2005.

[7] 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.

[8] E. Dijkstra. *A Discipline of Programming*. Pretice Hall, NJ, 1976.

[9] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to c. In *Computer Aided Verification (CAV'06)*. Springer, 2006. LNCS series.

[10] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 2005.

[11] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking of Software: 10th International SPIN Workshop*, pages 121–135. Springer-Verlag, May 2003. LNCS 2648.

[12] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *7th European Software Engineering Conference (ESEC/FSE'99)*, pages 303–321. Springer, 1999. LNCS 1687.

[13] H. Jain, F. Ivančić, A. Gupta, and M. Ganai. Localization and register sharing for predicate abstraction. In *Tools and Algorithms for the Construction and Abnalysis of Systems (TACAS'05)*, pages 394–409. Springer, 2005. LNCS 3440.

[14] H. Jain, F. Ivančić, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Computer Aided Verification (CAV'06)*. Springer, 2006. LNCS series.

[15] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV '05)*, pages 226–238. Springer, 2005. LNCS 3576.

[16] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ACM/IEEE International Conference on Software Engineering*, 2002.

[17] B. Korel and J. W. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[18] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 82–96, Berlin, September 1999. Springer-Verlag. LNCS 1703.

[19] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17, April 2003. LNCS 2619.

[20] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification (CAV'00)*, pages 435–449. Springer, 2000. LNCS 1855.

[21] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Computer Aided Verification (CAV'02)*, pages 485–499. Springer-Verlag, July 2002. LNCS 2404.

[22] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth Annual Symposium on Programming*, 1981.

[23] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, October 2003.

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

[25] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *Correct Hardware Design and Verification Methods (CHARME '05)*, pages 35–49. Springer, 2005. LNCS 3725.

[26] A. Zeller. Isolating cause-effect chains from computer programs. In *Symposium on the Foundations of Software Engineering (FSE'02)*, pages 1–10, November 2002.