Locating Causes of Program Failures

Holger Cleve
Saarland University
Saarbrücken, Germany
cleve@cs.uni-sb.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-sb.de

ABSTRACT

Which is the defect that causes a software failure? By comparing the program states of a failing and a passing run, we can identify the *state differences* that cause the failure. However, these state differences can occur all over the program run. Therefore, we focus *in space* on those variables and values that are relevant for the failure, and *in time* on those moments where *cause transitions* occur—moments where new relevant variables begin being failure causes: "Initially, variable `argc` was 3; therefore, at `shell_sort()`, variable `a[2]` was 0, and therefore, the program failed." In our evaluation, cause transitions locate the failure-inducing defect twice as well as the best methods known so far.

Keywords: Automated debugging, program analysis, adaptive testing, tracing

1. INTRODUCTION

Some program fails. How did this happen? In principle, a failure is created by a *defect* in the code, which creates a fault or *infection*[1], in the program state which then propagates until it becomes an observable *failure.* Tracing back the infection chain from failure to defect is a hard task, because programmers must both

- *search in space* across a program state to find the infected variable(s)—often among thousands—, and

- *search in time* over millions of such program states to find the moment in time when the infection began—that is, the moment the defect was executed.

We want to ease this search as much as possible. In earlier work [16], we have shown how to *search in space*—by focusing on those variables that actually *cause* the failure. The idea is to focus on the *difference* between the program states of a run where the failure in question occurs, and the states of a run where the failure does not occur. Using an automatic strategy called Delta Debugging, we can

---

[1]The term "infection" was coined in [14]; we prefer it over "fault" because it applies exclusively to program state, and because it implies the idea of propagation.

systematically narrow these initial differences down to a small set of variables: "The GCC failure occurs if and only if `fld[0].rtx`, the second child of the `link` node, points to its parent"—a cycle in the abstract syntax tree. That is, variable `fld[0].rtx` is a *failure cause*: if `fld[0].rtx` is altered such that it no longer induces a cycle, the failure no longer occurs.

Unfortunately, finding causes in state is not enough. From input to failure, a program passes through thousands of states. In these, we can always isolate differences that cause the failure in one run and not in the other—simply because identical inputs and states would result in identical outcomes. Which are the moments in time we should focus upon?

In this paper, we show how to *search in time*—by focusing on *cause transitions:* moments in time where some variable ceases to be a failure cause and another variable begins. In the GCC example, such a cause transition is the moment where an earlier variable v caused the cycle to be created. From this moment on, v no longer has an effect on the failure, but the cycle has; the statement which created the cycle is likely the defect we are looking for.

A cause transition is where a cause originates—that is, it points to *program code* that causes the transition and hence the failure. Thus, a cause transition is a candidate for a code correction—and cause transitions can be isolated automatically, just like causes in the program state. But cause transitions are not only good locations for fixes—they actually locate *the defects that cause the failure.* In fact, we show that cause transitions are significantly better locators of defects than any other methods previously known.

This paper is organized as follows: We illustrate our techniques with a short example (Section 2). We then recall our earlier work on isolating failure causes from program states—that is, searching in space (Section 3). Section 4 shows how to find cause transitions—that is, searching in time. Section 5 demonstrates how the technique scales, creating a diagnosis for the GCC compiler. Section 6 discusses complexity and other practical issues. Our evaluation in Section 7 demonstrates the general effectiveness of our approach, comparing against related work. Section 8 closes with conclusion and consequences; an appendix summarizes formal definitions.

2. A SAMPLE FAILURE

Figure 1 shows our ongoing example. Ideally, the `sample` program sorts its arguments numerically and prints the sorted list, as in this run ($r_✔$):

```
$ sample 9 8 7
7 8 9
$ _
```

With certain arguments, `sample` fails (run $r_✗$):

```
$ sample 11 14
```

```
1    /* sample.c -- Sample C program to be debugged */
2
3    #include <stdio.h>
4    #include <stdlib.h>
5
6    static void shell_sort(int a[], int size)
7    {
8        int i, j;
9        int h = 1;
10       do {
11           h = h * 3 + 1;
12       } while (h <= size);
13       do {
14           h /= 3;
15           for (i = h; i < size; i++)
16           {
17               int v = a[i];
18               for (j = i; j >= h && a[j - h] > v; j -= h)
19                   a[j] = a[j - h];
20               if (i != j)
21                   a[j] = v;
22           }
23       } while (h != 1);
24   }
25
26   int main(int argc, char *argv[])
27   {
28       int  i = 0;
29       int *a = NULL;
30
31       a = (int *)malloc((argc - 1) * sizeof(int));
32       for (i = 0; i < argc - 1; i++)
33           a[i] = atoi(argv[i + 1]);
34
35       shell_sort(a, argc);
36
37       for (i = 0; i < argc - 1; i++)
38           printf("%d ", a[i]);
39       printf("\n");
40
41       free(a);
42       return 0;
43   }
```

Figure 1: The buggy *sample* program. This program sorts its arguments—that is, mostly.

```
0 11
$ _
```

Although the output is properly sorted, one of the arguments has been replaced by another number: in run $r_\mathbf{x}$, the argument 14 has been replaced by 0. The presence of 0 is a *failure*, which implies that there is some bug (or *defect*) in the program code. When the defect is executed, it makes the state faulty (or *infected*). As the execution progresses, this infection causes further infections, until the infection becomes visible as the failure.

To find the defect, we must trace back this *infection chain:* For each infection I', we must find the earlier infection I that causes I' as well as the failure (or find that there is no earlier infection, in which case we found the defect). This requires two proofs:

- To prove that I and I' are *infected,* we must show that they violate the specification.

- To prove that I *causes* I', we must show that the effect I' does not occur if the cause I does not occur.

In the absence of a detailed specification, we cannot determine whether a program state is infected or not. (Furthermore, a specification that covers every aspect of each program state as it occurs during the run is unlikely to exist.) However, we can focus on *causes.* In earlier work [16], summarized in Section 3, we have shown how to *search causes in space:* The basic idea is to identify the state difference between $r_\checkmark$ and $r_\mathbf{x}$ at some moment in time and to narrow down that difference to a relevant subset that causes the

Var	Value in $r_\checkmark$	in $r_\mathbf{x}$
argc	4	5
argv[0]	"./sample"	"./sample"
argv[1]	"9"	"11"
argv[2]	"8"	"14"
argv[3]	"7"	0x0 (NULL)
i'	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	4	3

Var	Value in $r_\checkmark$	in $r_\mathbf{x}$
i	3	2
a[0]	9	11
a[1]	8	14
a[2]	7	0
a[3]	1961	1961
a'[0]	9	11
a'[1]	8	14
a'[2]	7	0
a'[3]	1961	1961

Table 1: State differences between $r_\checkmark$ and $r_\mathbf{x}$. One of these differences causes *sample* to fail.

failure. In the sample program, for instance, we can examine the call of shell_sort() in Line 35 and find that $a[2]$ being zero causes the failure of $r_\mathbf{x}$.

The variable $a[2]$ is a cause, but not infected. Actually, the value of $a[2]$ should not influence the outcome at all, as $a[]$ should have only two elements. Variable $a[2]$ *does* affect the outcome, though. Therefore, it must be some other infection that makes $a[2]$ affect the outcome, and this can only be *size*, the number of arguments to be sorted. To correct the defect, Line 35 should be changed to shell_sort(a, argc - 1).

But how do we know that we should search at Line 35 in the first place? In terms of causality, Line 35 is a highly interesting place. Up to Line 35, variable *argc* causes the failure (because changing *argc*, and *argc* only, determines whether the failure occurs). From Line 35 on, it is $a[2]$ that causes the failure. This *cause transition* from *argc* to $a[2]$ means that the new cause $a[2]$ *originated* right at Line 35. Line 35 is thus a likely candidate for fixing the program.

Two more such cause transitions occur within shell_sort()—from $a[2]$ to v in Line 17 and from v to $a[0]$ in Line 21. Taken together, these locations are central for explaining how the failure came to be—but they are also *failure causes,* because if they had been executed differently, the failure-causing state would not have come to be. This is the key question of this paper: To which extent are such cause transitions not only failure causes, but actual defects?

3. SEARCHING IN SPACE

Let us first recall our earlier work [16] on isolating causes in program state. Given some moment in time, we can extract program states from both $r_\checkmark$ and $r_\mathbf{x}$, and *compare* these two states. (Technically, this is achieved by instrumenting an ordinary debugger such as GDB.) Table 1 lists the sample program states, as well as the differences, as obtained from both $r_\checkmark$ and $r_\mathbf{x}$ when Line 9 was reached. (a and i occur in shell_sort() and in main(); the shell_sort() instances are denoted as a' and i'.)

Formally, this set of 12 differences is a failure cause: If we change the state of $r_\checkmark$ to the state in $r_\mathbf{x}$, we obtain the original failure. However, of all differences, only some may be *relevant* for the failure—that is, it may suffice to change only a *subset* of the variables to make the failure occur. For a precise diagnosis, we are interested in obtaining a subset of relevant variables that is as small as possible.

Delta Debugging [18] is a general procedure to obtain such a small subset. Given a set of differences (such as the differences between the program states in Figure 1), Delta Debugging determines a *relevant subset* in which each remaining difference is relevant for the failure to occur. To do so, Delta Debugging systematically and automatically *tests* subsets and narrows down the difference depending on the test outcome, as sketched in Figure 2. Overall,
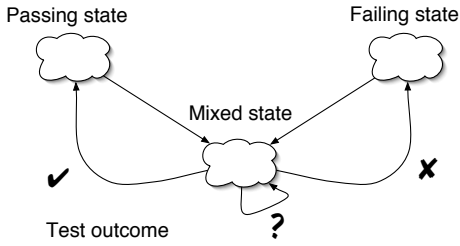
Figure 2: Narrowing down state differences. By assessing whether a mixed state results in a passing (✔), a failing (✘), or an unresolved (?) outcome, Delta Debugging isolates a relevant difference.

Delta Debugging behaves very much like a binary search.

Applied on the differences in Table 1, Delta Debugging would result in a first test that

- runs $r_✔$ up to Line 9,

- applies *half* of the differences on $r_✔$—that is, it sets *argc*, *argv*[1], *argv*[2], *argv*[3], *size*, and *i* to the values from $r_✘$—, and

- resumes execution and determines the outcome.

This test results in the same output as the original run; that is, the six differences applied were not relevant for the failure. With this experiment, Delta Debugging has narrowed down the failure-inducing difference to the remaining six differences. Repeating the search on this subset eventually reveals one single variable, *a*[2], whose zero value is failure-inducing: If, in $r_✔$, we set *a*[2] from 7 to 0, the output is `0 8 9`—the failure occurs. We can thus conclude that the zero being printed is caused by *a*[2]—which we can confirm further by changing *a*[2] in $r_✘$ from 0 to 7, and obtaining the output `7 11`. Thus, in Line 9, *a*[2] being zero causes the `sample` failure.

The idea of determining causes by experimenting with mixed program states (rather than by analyzing the program or its run, for instance) may seem strange at first. Yet, the technique has been shown to produce useful diagnoses for programs as large as the GNU compiler (GCC); it has also been implemented in the ASKIGOR public debugging server [1]. As detailed in [16], scaling up the general idea, as sketched here, requires capturing and comparing program states as *memory graphs* [19]. Also, Delta Debugging must do more than simple binary search; it needs to cope with interferences of multiple failure-inducing elements as well as with unresolved test outcomes [18].

4. SEARCHING IN TIME

When debugging, we are ultimately looking for the *defect that causes the failure*. This implies *searching in time* for the moment the defect has been executed and originated the infection chain.

In the absence of a correct program, we cannot tell whether a piece of code is a defect. What we can tell, though, is whether a piece of code is the origin of a cause-effect chain that leads to the failure. Such origins are can be tied to *variables.* Assume there is a point where a variable *A* ceases to be a failure cause, and a variable *B* begins. (These variables are isolated using Delta Debugging, as sketched in Section 3.) Such a *cause transition* from *A* to *B* is an origin of *B* as a failure cause. A cause transition thus is a good place to *break* the cause-effect chain and to fix the program—and since a cause transition may be a good fix, it may also indicate the actual defect.
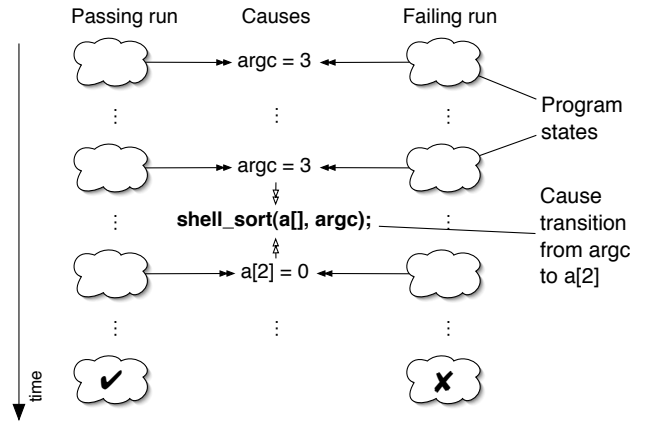

Figure 3: A cause transition. For each program step, we can narrow down variables that cause the failure. When these variables change, we have a cause transition—a potential defect.

How do we locate such transitions? The process is sketched in Figure 3: before the call to `shell_sort()`, Delta Debugging isolates *argc* as a failure cause. Afterwards, *a*[2] is the failure cause. In order to find the moment of that cause transition, we apply Delta Debugging in the middle of the interval. Then we repeat the process for the two sub-intervals, effectively narrowing down the transitions until we find only *direct* transitions from one moment to the next. These direct transitions are associated with the statement executed at this point, resulting in Line 35, the `shell_sort()` call.

The actual algorithm which finds direct cause transitions from two runs is named *cts* for *cause transitions*; it is formally defined in the Appendix. To understand how *cts* works, we illustrate it on the `sample` program.

Table 2 summarizes the execution of *cts*. The runs $r_✔$ and $r_✘$ execute a sequence of 54 and 38 statements, respectively; Column 1 shows the sequence number of the executed statement, Column 2 contains line number and code. Not every line is executed in both runs, though; coverage is indicated by "•" in the $r_✔$ and $r_✘$ columns, respectively.

1. To find an interval of matches to start with, we determine relevant variables at the first matching point and the last point where the failure has not occurred.

2. At the first executed line of both runs (Step 1), the value of variable *argc* can be determined to be relevant for the failure. Obtaining the relevant variables at the last executed line before printing zero (Step 44) yields *a*[0], which contains zero. So, there was a cause transition between *argc* in step 1 and *a*[0] in Step 44, since *a*[0] did not exist in Step 1; *argc* is no longer relevant in Step 44, even though it holds the same value as in the beginning.

3. To narrow down this cause transition, *cts* searches a matching point between Steps 1 and 44. Our implementation prefers function calls to other statements, so we end up in Step 11. Applying Delta Debugging now isolates *a*[2] as relevant. Thus, we have a transition from *argc* to *a*[2], and a transition from *a*[2] to *a*[0]. Again, these transitions must be narrowed down.

4. At Step 26, again *a*[2] is isolated, so there is no need to search between Steps 11 and 26. Between Steps 26 and 44 lies Step 35, where *v* is relevant, refining the cause transition from *a*[2] to *a*[0] into two cause transitions to and from *v*.

Step	Line	Code	$r_{\checkmark}$	$r_{\times}$	Vars	*cts* step
1	28	`int i=0;`	●	●	argc=3	2
⋮						
6	32	`for(i=0;i<argc-1;i++)`	●	●	argc=3	11
7	33	`a[i]=atoi(argv[i+1]);`	●	●		
8	32	`for(i=0;i<argc-1;i++)`	●	●	argc=3	12
9	33	`a[i]=atoi(argv[i+1]);`	●		?	
10	32	`for(i=0;i<argc-1;i++)`	●		?	
11	35	`shell_sort(a,argc);`	●	●	a[2]=0	3
⋮						
26	20	`if(i!=j)`	●	●	a[2]=0	4
27	21	`a[j]=v;`	●			
28	15	`for(i=h;i<size;i++)`	●	●	a[2]=0	9
29	17	`int v=a[i];`	●	●	a[2]=0	10
30	18	`for(j=i;j>=h&&a[...`	●	●	v=0	8
⋮						
35	20	`if(i!=j)`	●	●	v=0	5
36	21	`a[j]=v;`	●	●	v=0	7
37	15	`for(i=h;i<size;i++)`	●	●	a[0]=0	6
⋮						
44	37	`for(i=0;i<argc-1;i++)`	●	●	a[0]=0	1
45	38	`printf("%d ",a[i]);`	●	●		

Table 2: Locating direct cause transitions

5. Continuing the process, we find three direct cause transitions, highlighting how the failure came to be. In Table 2, Column "Vars" shows the relevant variables, and Column "*cts* step" shows the order of search points where relevant states were isolated. Direct cause transitions are shown by dividing lines.

6. Note the cause transition between Steps 8 and 11: the execution traces have no matching points in between, thus the transition cannot be exactly located at one executed line of code. Setting *argc* to the value of the failing run alone causes the control flow of the resumed run to change, being different from both original runs. Our implementation thus locates the beginning of the cause transition at the earlier matched point.

Overall, we obtain cause transitions from *argc* to *a*[2] in Lines 32–35 (Steps 8–11), from *a*[2] to *v* in Line 17 (Step 29), and from *v* to *a*[0] in Line 21 (Step 36). All of these locations are potential places to fix such that the cause-effect chain is broken.

The *cts* algorithm has been implemented as part of the ASKIGOR public debugging server. The cause transitions for `sample` and the involved variables are reported as a cause-effect chain (Figure 4).

5. CASE STUDY: THE GCC FAILURE

The `sample` program is a relatively small program, such that the question may arise whether the approach is feasible for larger programs, too. To explore scaleability, we applied the technique to locate a failure cause in the GNU compiler (GCC).

Consider the *fail.c* program in Figure 5. This program is interesting in one aspect: It causes the GNU C compiler (GCC) to crash—at least, when using version 2.95.2 on Intel-Linux with optimization enabled:

```
$ gcc −O fail.c
gcc: Internal compiler error:
    program cc1 got fatal signal 11
$ _
```
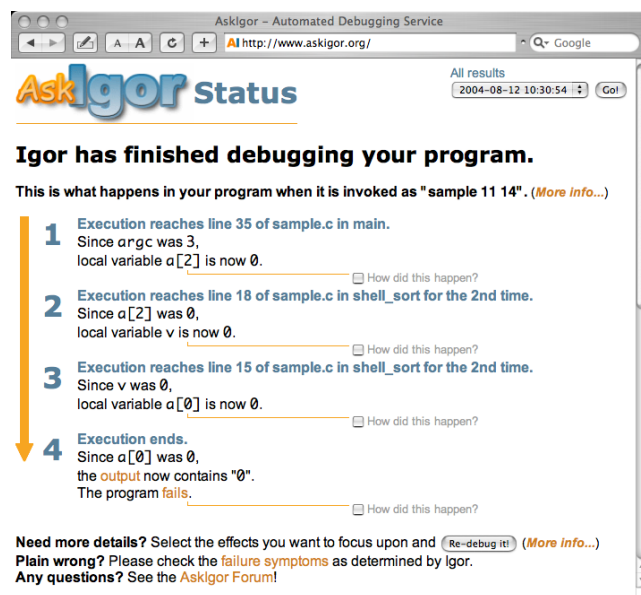

Figure 4: ASKIGOR with a diagnosis for *sample*

In earlier work, we had used this example twice to determine failure causes:

- Applying Delta Debugging on the input (i.e. the code *fail.c*), it turned out that if the code + 1.0 was omitted, the program compiled just fine—that is, the input + 1.0 causes the failure [18].

- Applying Delta Debugging on the program states of a passing run (i.e. without + 1.0) and a failing run (i.e. with + 1.0) returned a cycle in the abstract syntax tree as failure cause [16].

To isolate potential causes in the program code, we applied our technique to isolate the cause transitions for the GCC failure. Overall, our algorithm identified 10 such transitions, listed in Table 3.

Again, these transitions summarize how the failure came to be—as a cause-effect chain from input to failure. The failure cause propagates through the GCC execution in four major blocks:

1. Initially, the file name (*fail.c*) is the failure cause—called with *pass.c*, the alternate input file without + 1.0, the error does not occur. This argument is finally passed to the GCC lexer (Transitions 1–3).

2. In the lexical and syntactical analysis (Transitions 4–6), it is the actual difference in file content which becomes a failure cause—that is, the characters + 1.0.

```
1   double mult(double z[], int n)
2   {
3       int i, j;
4       i = 0;
5       for (j = 0; j < n; j++) {
6           i = i + j + 1;
7           z[i] = z[i] * (z[0] + 1.0);
8       }
9       return z[n];
10  }
```

Figure 5: The *fail.c* program that crashes GCC.

#	Location	Cause transition to variable
0	⟨Start⟩	`argv[3]`
1	toplev.c:4755	`name`
2	toplev.c:2909	`dump_base_name`
3	c-lex.c:187	`finput→_IO_buf_base`
4	c-lex.c:1213	`nextchar`
5	c-lex.c:1213	`yyssa[41]`
6	c-typeck.c:3615	`yyssa[42]`

7	c-lex.c:1213	`last_insn→fld[1].rtx`
		`→fld[1].rtx→fld[3].rtx`
		`→fld[1].rtx.code`

8	c-decl.c:1213	`sequence_result[2]`
		`→fld[0].rtvec`
		`→elem[0].rtx→fld[1].rtx`
		`→fld[1].rtx→fld[1].rtx`
		`→fld[1].rtx→fld[1].rtx`
		`→fld[1].rtx→fld[1].rtx`
		`→fld[3].rtx→fld[1].rtx.code`

9	combine.c:4271	`x→fld[0].rtx→fld[0].rtx`

Table 3: Cause transitions in GCC

3. The difference in file content becomes a difference in the abstract syntax tree, where + 1.0 induces `fld[1].rtx` to hold an additional node (`fld[1].rtx.code` is `PLUS`) in the failing run (Transitions 7–8). Thus, the + in the input has caused a `PLUS` node, created at Transition 8.

4. In Transition 9, the failure cause moves from the additional `PLUS` node to a *cycle* in the abstract syntax tree. We have

$$x→fld[0].rtx→fld[0].rtx = x$$

meaning that the node at `*x` is its own grandchild! This cycle ultimately causes an endless recursion and thus the GCC crash.

In our earlier work [16], we had also identified the cycle as the ultimate failure cause, and assumed that an experienced GCC programmer would be able to distinguish infections from non-infections. Therefore, an experienced programmer would have immediately focused on the GCC cycle.

Under the assumption that cause transitions indicate defects, a programmer less familiar with GCC could start his investigation at all listed cause transitions—starting with the transitions closest to the failure. At combine.c:4271, the location of the last transition, we find a single statement

```
return x;
```

This line is not likely to be a defect. Let us take a look at the direct origin of x, in combine.c:4013–4019, listed in Figure 6.

This place is where the infection originates: The call to the function `apply_distributive_law()` is wrong. This function transforms code using the rule

$$(MULT \ (PLUS \ a \ b) \ c) \Rightarrow (PLUS \ (MULT \ a \ c_1)(MULT \ b \ c_2))$$

Unfortunately, in the `apply_distributive_law()` call in Figure 6, the third and fourth arguments c_1 and c_2 share a common grandchild (the macro `XEXP(x, 1)` translates into the expression `x→fld[1].rtx`), which leads to the cycle in the abstract syntax tree. To fix the problem, one should call the function with a *copy* of the grandchild—and this is how the error was fixed in GCC 2.95.3.

At this point, one may wonder why cause transitions did not single out the call to `apply_distributive_law()` as a cause

```
case MULT:
 /* If we have (mult (plus A B) C), apply the distributive
    law and then the inverse distributive law to see if
    things simplify.  This occurs mostly in addresses,
    often when unrolling loops.  */

 if (GET_CODE (XEXP (x, 0)) == PLUS)
   {
     x = apply_distributive_law
       (gen_binary (PLUS, mode,
                    gen_binary (MULT, mode,
                                XEXP (XEXP (x, 0), 0),
                                XEXP (x, 1)),
                    gen_binary (MULT, mode,
                                XEXP (XEXP (x, 0), 1),
                                XEXP (x, 1))));

     if (GET_CODE (x) != MULT)
       return x;
   }
 break;
```

Figure 6: The GCC defect in combine.c

transition. The answer is simple: This piece of code is executed only during the failing run. Therefore, we have no state to compare against, and therefore, we cannot narrow down the cause transition any further. Line 4271, however, has been executed in both runs, and thus we are able to isolate the failure-inducing state at this location.

Overall, to locate the defect, the programmer had to follow just one backwards dependency from the last isolated cause transition. In numbers, this translates into just 2 lines out of 338,000 lines of GCC code. Even if we assume the programmer examines all 9 transitions and all direct dependencies, the effort to locate the GCC defect is minimal.

6. COMPLEXITY AND OTHER ISSUES

Finding causes and cause transitions by automated experimentation can require a large number of test runs:

Searching in space. In the best case, Delta Debugging needs $2s \log k$ test runs to isolate s failure-inducing variables from k state differences. The (pathological) worst case is $k^2 + 3k$; In practice, though, Delta Debugging is much more logarithmic than linear.

Searching in time. This is a simple binary search over n program steps, repeated for each cause transition. For m cause transitions, we thus need $m \log n$ runs of Delta Debugging.[2]

Since applications can have a large number of fine-grained cause transitions, a practical implementation would simply limit the number of cause transitions to be sought, or just run as long as the available execution time permits.

Other practical issues we faced in our implementation, in particular for the GCC case study, included:

Accessing state. We currently instrument the GNU debugger (GDB) to access the state, which is painfully slow: The entire GCC

---

[2]Unfortunately, a pure binary search does not always suffice. In a cause-effect chain, all reported causes must cause all later causes as well as the failure. This can lead to tricky situations: Assume we have isolated a cause c_1 and a later cause c_2, and these two form a cause-effect chain, meaning that c_1 causes c_2 as well as the failure. Now, *cts* isolates a new cause c between c_1 and c_2; again, c causes all later causes (c_2) as well as the failure. But does c_1 cause c, too? In case c_1 has no effect on c, we have to *re-isolate* c_1 such that the new c_1 causes c as well as c_2.

diagnosis takes about 12 hours to compute.[3] One can think of much better ways to access and compare program states directly.

Capturing accurate states. Capturing and transferring the state of a C program to another is a tricky business [19]. For instance, we had to implement several heuristics to determine what type of element a pointer points to, and at how many it points. When such heuristics fail, the state cannot be transferred, and we cannot determine a relevant state difference.

Incomparable states. When control flow reaches different points in $r_\checkmark$ and $r_\times$, the resulting states are not comparable—simply because the set of local variables is different. To determine when the control flows of $r_\checkmark$ and $r_\times$ diverge and converge requires some effort.

So far, we can isolate causes and cause transitions from programs whose state is well-defined—that is, typical stand-alone C programs without much external state or user interaction. We are currently porting our techniques to languages with managed memory (such as JAVA and C#) and expect much relief.

7. EVALUATION: THE SIEMENS SUITE

The sample and GCC examples show that cause transitions can help to locate the failure-inducing defect in a very precise manner. However, these results do not generalize: They do not necessarily show an advance beyond the state of the art, nor do they show a more general usefulness. Therefore, we conducted an evaluation using a larger number of programs with known defects, which had already been used to evaluate other defect localization techniques.

7.1 Object of Analysis

Our object of evaluation is the *Siemens test suite* [7], as modified by Rothermel and Harrold [13] as well as Renieris and Reiss [12]. This test suite consists of seven C programs with 170 to 560 lines of code, as well as 132 variations of these seven programs, each with exactly one manually injected defect. Defects may span multiple statements or even functions. Several defects are created by omitting code, or relaxing or tightening control conditions. Each of the seven program families comes with a test suite that exposes the defect in each of the faulty versions.

7.2 Earlier Results and Related Work

In the past, the Siemens test suite has been used to determine the effectiveness of methods that locate defects from alternate test runs.

Coverage. One suggested method to locate defects is to compare the coverage or *spectra* of passing and failing test runs. The idea is that code executed in failing runs only is more likely to lead to the defect than code that is executed in all runs. Harrold et al. found that failing runs tend to have unusual coverage [5]; such information can also be visualized to assist programmers [8]. However, applied to the Siemens suite, summarizing the coverage of failing runs either by intersection or union yields no useful results [12].

Slicing. *Program slicing* [15] yields the set of statements that potentially may have influenced the state at a given statement; *Dynamic slicing* [2, 9] does so for a specific run. The Siemens test suite so far has not been subject to dynamic slicing. A dynamic backward slice from the program output, though,

typically contains *all* executed statements, simply because an executed statement that does not influence the output would be an anomaly. In this evaluation, we thus assume that a dynamic slice is equal to the coverage; in particular, a difference or *dice* between slices should yield the same results as the difference between coverage.

Dynamic invariants. Rather than focusing on coverage, one may also attempt to summarize the properties of *data* as found in the passing runs. The approach of Ernst et al. [3] determines *dynamic invariants* from a number of (passing) runs, thus summarizing their common properties. The idea is that failing runs might violate some of these invariants, thus highlighting data anomalies that might lead to a defect. However, the study of Pytlik et al. [11] showed no success of this approach when being applied to the Siemens suite.

Explicit specification. Predicting a defect location becomes much easier if a specification of correct *internal* behavior is given (in contrast to a test case, which checks only *external* behavior). This has been explored by Groce [4], who used model checking for locating defects in a subset of the Siemens suite, and which produced good results on these subsets. However, we do not assume the existence of any specifications besides the test.

Nearest neighbor. The most successful method so far to predict defects in the Siemens suite is the *nearest neighbor* approach by Renieris and Reiss [12]. Rather than attempting to summarize the properties of multiple passing runs, this generic approach selects the single passing run that is *closest* (in coverage, or some other property) to the failing run and focuses on the differences between these two runs alone. Applied to coverage in the Siemens test suite, this method showed moderate success to predict defects.

One should note that results obtained from the Siemens test suite do not generalize to arbitrary programs. It is likely that the methods described perform much better for other programs with a greater separation of concerns. In the context of this paper, though, we use the Siemens test suite as a *benchmark:* Cause transitions must prove to be better locators of software defects than the best method known, which is Renieris and Reiss' nearest neighbor heuristic.

7.3 Variables and Measures

Independent variables. We manipulated one independent variable: the used defect locator (Reiss and Renieris' "NN/Perm" versus cause transitions "CT").

Dependent variables. To determine the quality of a defect locator, Renieris and Reiss have introduced the concept of a *score* [12], indicating the fraction of the code that can be ignored when searching for a defect.

If the score is $S = 0.95 = 95\%$, then the programmer can ignore 95% of the code; she has to examine only 5% to locate the defect. If $S = 0\%$, the programmer must examine *all* of the code—the report is useless. The higher the score, the better the defect locator.

The score computation [12] can be summarized as follows:

1. We have two versions of a program: A failing version $p_\times$ and a passing version $p_\checkmark$ where the defect has been fixed.

---

[3]All times measured on a 3 Ghz Pentium PC.

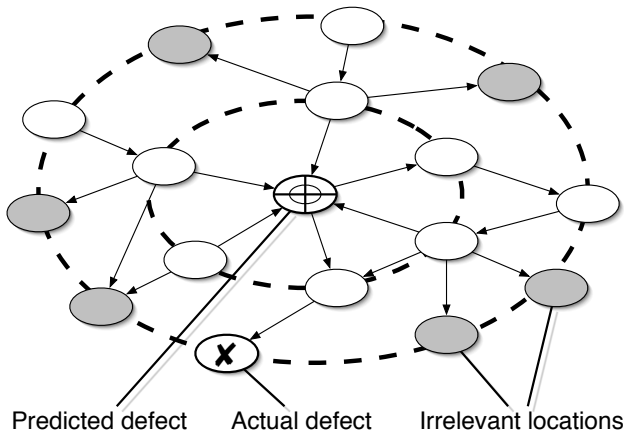**Predicted defect**   **Actual defect**   **Irrelevant locations**

Figure 7: Distance between predicted and actual defect. To find the actual defect, which is two dependencies away, the programmer has to examine up to 16 locations in the code.

2. We construct a program dependence graph (PDG) [6] for $p_\times$, a graph which contains a node for each statement in the program, and edges for data and control dependencies between these statements.

3. We mark all nodes in the PDG as "defect" if they have been fixed in $p_\checkmark$.

4. A defect locator, working on $p_\times$, reports a set $R = \{n_1, n_2, \dots\}$ of nodes n_i in the PDG as being likely defect locations. In the CT case, we used the location of each occurring cause transition as such a likely defect; in the PDG, we mark these nodes as "blamed".

   In the GCC example from Section 5, we would blame the locations listed in Table 3.

5. Let $k(n, e)$ be the set of nodes that are reachable from n within the distance e. For each blamed node $n \in R$, we determine the distance $d(n)$ to the nearest defect. Then, $k(n, d(n))$ reflects the maximum number of locations in the program a programmer has to examine, starting with n and increasing the distance until the actual defect is found (Figure 7). Note that if $d(n) = 0$ holds, we have a perfect match; only $k(n, 0) = \{n\}$ needs to be examined.

   Again, in the GCC example from Section 5, we would start with the blamed nodes and find the defect at a distance of 1, as combine.c:4013–4019 is reachable via one dependency from the blamed location.

6. From all blamed nodes R, we now determine some node $m \in R$ that is closest to a defect (i.e., $d(m) \le d(n)$ holds for all $n \in R$), and we determine the set of nodes N that had to be examined up to that distance:

$$N = \bigcup_{n \in R} k(n, d(m))$$

   The idea is that the programmer does a breadth-first search across the PDG, starting with the blamed nodes, and increasing the distance until a defect is found.

   In the GCC example, N would include all nodes at a distance of 1 from the 9 blamed locations.

7. The fewer nodes one must examine when searching for the defect, the better the quality of the defect locator. This *score* can be expressed as a fraction of the PDG:

$$S = 1 - \frac{|N|}{|\text{PDG}|}$$

7.4 Improved Strategies

A cause transition at a blamed location b can impact the failure only in two ways:

- If the state at b is infected, it is *caused* by the defect. That is, the defect d is in the backward slice of b or $b \leftarrow^* d$. ("←" and "→" are backward and forward dependencies in the PDG, respectively.)

- If the state at b is not infected, it must nonetheless *cause* the infection to reach the failure. In this case, there must be some future location n at which the infection of d and the effect of b meet ($b \rightarrow^* n \leftarrow^* d$).

This knowledge about cause transitions and dependencies can be exploited by the programmer, using alternate search strategies along the PDG:

Exploiting relevance. A node $m \notin \{n \mid b \rightarrow^i \leftarrow^j n\}$ neither can have caused an infection in b nor meets with the effect of b to cause the failure—at least not in the distance $i + j$ considered so far. In Figure 7, such *irrelevant* locations are shown in grey.

Exploiting infections. Let us assume that a programmer is able to tell infected from non-infected variables in a report, and that one of the blamed locations b is a cause transition from some non-infected state to an infected state. Then, $b \leftarrow^* d$ must hold for the defect d, and we can use this single cause transition as starting point.

Both concepts exploited here are unique to cause transitions. Code coverage does not convey data information, and like violations of dynamic invariants or other anomalies, it can not be shown to cause the failure.

7.5 Experiment Setup

We ran our techniques on the 132 variations of the Siemens programs. Three out of these 132 had to be ruled out (two because their test suites would not observe any failure, and one due to issues with input processing). In order to treat program input and program state uniformly, we altered the programs so that they would read input from internal variables rather than from external files.

For each variation $p_\times$, we then randomly picked one failing run $r_\times$ as well as the passing run $r_\checkmark$ that would produce a maximum of cause transitions. (Just as Renieris and Reiss, we thus exploited the fact that multiple test runs were available.) The runs $r_\times$ and $r_\checkmark$ were then fed into the defect locator, resulting in a report R. From $p_\checkmark$, $p_\times$, and R, we then computed the score S of the resulting report, using CODESURFER to compute the PDG of $p_\times$.

Table 4 shows statistics about the experiments, summarized by program. "#calls" is the lengths of the traces (in function calls), "*avg(time)*" is the average time in seconds for computing the diagnosis, "#tests" is the number of executed test runs per diagnosis, and "#cts" the number of isolated direct cause transitions (i.e., the number of "blamed" nodes).

Name	PDG size	#calls	*avg(time)*	#tests	#cts
print_tokens	1448	30–1845	2590.1	1–42	1–5
print_tokens2	1420	40–1587	6556.5	5–44	1–6
replace	1252	3–1139	3588.9	1–38	1–3
schedule	1350	2–575	1909.3	4–69	1–6
schedule2	1164	2–1336	7741.2	2–63	1–11
tcas	454	1–14	184.8	4–31	1–4
tot_info	728	14–350	521.4	2–41	1–5

Table 4: Properties of sampled programs

7.6 Results and Analysis

The evaluation results are summarized in Table 5, as relative distribution of scores per method and test runs. The data for the nearest neighbor method ("NN/Perm") is taken from [12]; one can see that no test run has a score of 100%, meaning the defect is never pinpointed. However, 16.51% of all test runs achieve a score of 90% or better, meaning that the programmer can stop her search at 10% of the code in 16.51% of all test runs.

The cause transitions method ("CT"), in comparison, pinpoints the defect in 4.65% of all test runs, and 26.36% of all test runs achieve a score of 90% or better, meaning an increase of 60%.

Figure 8 shows a cumulative plot of the score data in Table 5. It is easy to see that CT outperforms NN/Perm for all scores of 70% and more. (NN/Perm is slightly better for lower scores. This advantage is mostly irrelevant, though, given that the programmer has to examine a third of the program or more.)

Using cause transitions, the programmer need not explore irrelevant nodes, as discussed in Section 7.4. We obtain the results shown in the "CT/relevant" column in Table 5. As shown in Figure 8, this increases the score significantly. The defect can be pinpointed (score: 100%) in 5.43% of all runs—the best result for any method. 35.66% of all test runs achieve a score of 90% or better, meaning an increase of 116% over the nearest neighbor method.

If we assume the programmer can tell infected from non-infected values, as discussed in Section 7.4, the score also increases. This is shown as "CT/infected" in Table 5 and Figure 8. Exploiting infections, more than 55% of all runs achieve a score of 75% or better.

In 45% of all test runs, all methods achieved scores of 60% or lower. We have not yet found a common property of these runs, but a possible hypothesis is that there are defects whose location simply cannot be predicted. As an example, consider an uninitialized global variable: Although the initialization (i.e., the fix) can take place at almost any place in the program, a good score requires that the method predict the exact place where the initialization has been removed—which is impossible for any method. Note, though, that an uninitialized variable can be isolated as a failure cause and reported as part of a cause transition—even if the cause transition itself occurs far away from the defect.

As initially stated, we had multiple passing runs available and picked the run $r_\checkmark$ that would result in a maximum of cause transitions. If we do not take advantage of the presence of multiple passing runs and pick a random passing run instead, cause transitions still outperform NN/Perm.

7.7 Threats to Validity

Like any empirical study, this study has limitations that must be considered when interpreting its results. As stated initially, results obtained from the Siemens suite cannot be generalized to arbitrary programs. In particular, larger programs, or more precisely, programs with a greater separation of concerns are likely to produce better localization results, regardless of the method applied; this view is also supported by the GCC example in Section 5.

To a certain extent, this threat to external validity also applies to the relative performance of the discussed methods: it is well possi-

Score	NN/Perm	CT	CT/relevant	CT/infected
100%	0.00	4.65	5.43	4.55
90–99%	16.51	21.71	30.23	26.36
80–90%	9.17	11.63	6.20	10.91
70–80%	11.93	13.18	6.20	13.64
60–70%	13.76	1.55	9.30	4.55
50–60%	19.27	6.98	10.08	6.36
40–50%	3.67	3.10	3.88	1.82
30–40%	6.42	7.75	10.08	3.64
20–30%	1.83	4.65	3.10	7.27
10–20%	0.00	6.98	10.85	0.00
0–10%	17.43	17.83	4.65	20.91

Table 5: Evaluating defect locators: Cause transitions (CT) versus nearest neighbor (NN)
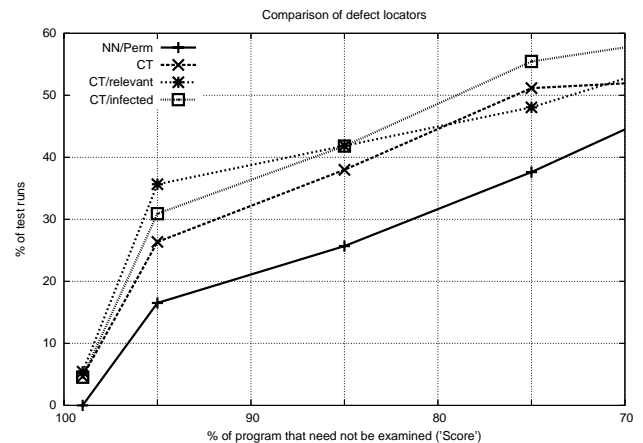

Figure 8: Evaluation details. For 30% of all test runs, cause transitions achieve a score of 90% or higher, narrowing down the defect to 10% or less of the code.

ble that for specific kinds of programs or defects, alternate methods perform better than cause transitions. This can be addressed by further case studies.

Threats to construct validity concern the appropriateness of our measures for capturing our dependent variables. The evaluation setup of Renieris and Reiss assumes an ideal programmer who is able to distinguish defects from non-defects at each location, and can do so at the same cost for each location considered. However, some defects are easier to spot than others, and this difference is not taken into account. The same applies to the ability of a programmer to tell infections from non-infections. Ultimately, the influence of such factors can only be determined by running studies with humans.

Threats to internal validity concern our ability to draw conclusions about the connections between our independent and dependent variables. In particular, our implementation could contain errors that affect the outcome. To control for these threats, we ensured that the diagnosis tools had no access to the corrected versions or any derivative thereof. Also, we repeated the CT experiments using Renieris and Reiss' framework instead of ours and found similar scores, which validates our evaluation setup,

8. CONCLUSION AND CONSEQUENCES

Cause transitions locate the software defect that causes a given failure, performing twice as well as any other technique previously known. The technique requires an automated test, a means to observe and manipulate the program state, as well as at least one alternate passing test run.

Typically, the technique would be used as an add-on to running

an automated test suite; if a test fails, the tool reports the cause-effect chain from input to failure, listing the isolated cause transitions as likely locations to fix. Thus, we not only know *that* a test has failed, but also *why* and *where* it failed—in terms of state and code, with each causality proven by a set of experiments. We expect such automatic diagnosis tools to reduce debugging efforts significantly.

Besides general issues of performance or portability, our future work will concentrate on the following topics:

Hierarchical search. Instead of searching for the exact location of state transfer, we may start with function calls as comparable locations. This reduces the number of transfer points and the length of the execution trace.

Ranking transitions. We expect that the greater the difference between the involved states and locations is, the more likely a transition is to be a defect. We want to identify the features of cause transitions that are most likely correlated with a defect, and use these features to focus on the most likely defects.

User-side diagnosis. Diagnosis methods may not only be part of testing environments, but also included in applications and operating systems, thus providing *diagnosis at the user's site.* Such diagnoses can then be collected and summarized by the maintainer.

Statistical causality. In the presence of multiple runs, statistical correlation between features of program runs and test failures [10] can be seen as a strong indicator of causality. During the search for causes and transitions, we can focus on such indicators, joining the predictive power of both statistical and experimental causality.

A discipline of debugging. Notions like causes and cause transitions can easily be generalized to serve in arbitrary debugging contexts. We are currently compiling a *textbook* [17] that shows how debugging can be conducted as systematically and as all other software engineering disciplines—be it manually or automated.

For the ASKIGOR source code and related work, see

<div align="center">http://www.askigor.org/</div>

9. REFERENCES

[1] AskIgor web site. http://www.askigor.org/.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, volume 25(6) of *ACM SIGPLAN Notices*, pages 246–256, White Plains, New York, June 1990.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

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

[5] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[6] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proc. 14th International Conference on Software Engineering*, pages 392–411, 1992.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[8] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, Florida, May 2002.

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

[10] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of the SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, June 2003.

[11] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated fault localization using potential invariants. In M. Ronsse, editor, *Proc. Fifth Int. Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Ghent, Belgium, Sept. 2003.

[12] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. 18th Int. Conference on Automated Software Engineering*, Montreal, Canada, 2003.

[13] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1998.

[14] J. M. Voas. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

[15] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[16] A. Zeller. Isolating cause-effect chains from computer programs. In W. G. Griswold, editor, *Proc. Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-10)*, pages 1–10, Charleston, South Carolina, Nov. 2002. ACM Press.

[17] A. Zeller. *Why does my program fail? A guide to automated debugging*. Morgan Kaufmann Publishers, May 2005. To appear (ISBN 1-55860-866-4).

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

[19] T. Zimmermann and A. Zeller. Visualizing memory graphs. In S. Diehl, editor, *Proc. of the International Dagstuhl Seminar on Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 191–204, Dagstuhl, Germany, May 2002. Springer-Verlag.

APPENDIX

A. FORMAL DEFINITIONS

In this appendix, we give a formal definition of cause transitions, using the `sample` program from Figure 1 to illustrate these definitions. We first recall the formal definitions of Delta Debugging [18], applied to program states:

Tests. A program run r is a sequence of states $r = [s_1, s_2, \ldots, s_n]$. Let $\mathcal{C}$ be the set of all state differences between program runs. The testing function $test : 2^{\mathcal{C}} \to \{\text{✘}, \text{✔}, \text{?}\}$ determines for a configuration $c \subseteq \mathcal{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).

Applied to the `sample` program, $test(c)$ would run $r_{\text{✔}}$ up to the location in question, apply the differences $c \subseteq \mathcal{C}$, resume execution, and return ✘, if the output contains a zero, and ✔, if not. (It would return ? if the output is never generated.)

Configurations. Let $c_{\text{✔}}$ and $c_{\text{✘}}$ be configurations with $c_{\text{✔}} \subseteq c_{\text{✘}} \subseteq \mathcal{C}$ such that $test(c_{\text{✔}}) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$. $c_{\text{✔}}$ is the "passing" configuration (typically, $c_{\text{✔}} = \emptyset$ holds) and $c_{\text{✘}}$ is the "failing" configuration.

In the case of `sample`, $c_{\text{✔}}$ and $c_{\text{✘}}$ are state differences with respect to $r_{\text{✔}}$, and obtained from $r_{\text{✔}}$ and $r_{\text{✘}}$, respectively. $c_{\text{✔}} = \emptyset$ always holds; $c_{\text{✘}}$ would contain differences as listed in Table 1.

Isolation. The *Delta Debugging algorithm* $dd(c_{\text{✔}}, c_{\text{✘}})$ isolates the failure-inducing difference between $c_{\text{✔}}$ and $c_{\text{✘}}$. It returns a pair $(c'_{\text{✔}}, c'_{\text{✘}}) = dd(c_{\text{✔}}, c_{\text{✘}})$ such that $c_{\text{✔}} \subseteq c'_{\text{✔}} \subseteq c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✔}}) = \text{✔}$, and $test(c'_{\text{✘}}) = \text{✘}$ hold and $\Delta = c'_{\text{✘}} \setminus c'_{\text{✔}}$ is *1-minimal*—that is, no single difference of $c'_{\text{✘}}$ can be removed from $c'_{\text{✘}}$ to make the failure disappear or added to $c'_{\text{✔}}$ to make the failure occur. The full definition of dd is found in [18].

In the case of the `sample` states as listed in Table 1, the set $\Delta = c'_{\text{✘}} \setminus c'_{\text{✔}}$ as returned by $dd(c_{\text{✔}}, c_{\text{✘}})$ would contain the relevant difference "$a[2] = 0$"—that is, an actual failure cause.

Not all states are comparable, though. We assume a matching function *match* that finds matching states:

Matching states. The function $match : (r_{\text{✘}} \to r_{\text{✔}} \cup \{\bot\})$ assigns each state $s_{\text{✘}t} \in r_{\text{✘}}$ a *matching state* $s_{\text{✔}t} \in r_{\text{✔}}$, or $\bot$, if no such match can be found.

Our implementation requires matching states to share a common calling context, implying an equal set of local variables.

Individual failure causes (= state differences) can be composed into a cause-effect chain.

Relevant deltas. For each $s_{\text{✘}t} \in r_{\text{✘}}$, let a *relevant delta* Δ_t be a failure-inducing difference, as determined by Delta Debugging: Let $s_{\text{✔}t} = match(s_{\text{✘}t})$; if $match(s_{\text{✘}t}) = \bot$ holds, then $\Delta_t = \bot$, too. Otherwise, let $c_{\text{✘}t}$ be the difference between $s_{\text{✔}t}$ and $s_{\text{✘}t}$, and let $c_{\text{✔}t} = \emptyset$. Let $(c'_{\text{✔}t}, c'_{\text{✘}t}) = dd(c_{\text{✔}t}, c_{\text{✘}t})$; then $\Delta_t = c'_{\text{✔}t} \setminus c'_{\text{✘}t}$ is a relevant delta.

Cause-effect chains. A sequence of relevant deltas $C = [\Delta_{t_1}, \Delta_{t_2}, \ldots]$ with $t_i < t_{i+1}$ is called a *cause-effect chain* if each Δ_{t_i} causes the subsequent $\Delta_{t_{i+1}}, \Delta_{t_{i+2}}, \ldots$ as well as the failure.

The ASKIGOR diagnosis in Figure 4 is a cause-effect chain at the moments in time $t_1 = \langle \text{Line 35 reached} \rangle$, $t_2 = \langle \text{Line 18 reached for the 2nd time} \rangle$, $t_3 = \langle \text{Line 15 reached for the 2nd time} \rangle$.

Within a cause-effect chain, *cause transitions* occur:

Cause transitions. Let $var(\Delta_t)$ be the set of variables affected by a state difference Δ_t; $var(\bot) = \emptyset$ holds. Then, two moments in time (t_1, t_2) are called a *cause transition* if $t_1 < t_2$, a cause-effect chain C with $[\Delta_{t_1}, \Delta_{t_2}] \subseteq C$ exists, and $var(\Delta_{t_1}) \neq var(\Delta_{t_2})$. A cause transition is called *direct* if $\neg \exists t : t_1 < t < t_2$.

In the ASKIGOR diagnosis in Figure 4, moment #1, the invocation of `shell_sort()`, is a cause transition from $var(\Delta_{t_1}) = \{\text{argc}\}$ to $var(\Delta_{t_2}) = \{\text{a[2]}\}$.

To isolate direct cause transitions, we use a *divide and conquer* algorithm. The basic idea is to start with the interval $(1, |r_{\text{✘}}|)$, reflecting the first and last state of $r_{\text{✘}}$. If a cause transition has occurred, we examine the state at the middle of the interval and check whether the cause transition has occurred in the first half and/or in the second half. This is continued until all cause transitions are narrowed down.

Isolating cause transitions. For a given cause-effect chain C, the algorithm $cts(t_1, t_2)$ narrows down the cause transitions between the moments in time t_1 and t_2:

$$cts(t_1, t_2) = \begin{cases} \emptyset & \text{if } var(\Delta_{t_1}) = var(\Delta_{t_2}) \\ cts(t_1, t) \cup cts(t, t_2) & \text{if } \exists t : t_1 < t < t_2 \\ \{(t_1, t_2)\} & \text{otherwise} \end{cases}$$

where $[\Delta_{t_1}, \Delta_{t_2}] \subseteq C$ holds.

A run of *cts* on `sample` is discussed in Section 4.

Our actual implementation computes C (and in particular, Δ_t) on demand. If we isolate a Δ_t between Δ_{t_1} and Δ_{t_2}, but find that Δ_t was not caused by Δ_{t_1}, we recompute Δ_{t_1} such that the cause-effect chain property is preserved.