Isolating Cause-Effect Chains from Computer Programs

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken, Germany
zeller@acm.org

ABSTRACT

Consider the execution of a failing program as a series of program states consisting of variables and their values. Each state induces the following state, up to the failure. Which part of a program state is relevant for the failure? We show how the *Delta Debugging* algorithm isolates the relevant variables and values by systematically narrowing the state difference between a passing run and a failing run. Applying Delta Debugging to multiple states of the program automatically reveals the *cause-effect chain* of the failure: "Initially, variable v_1 was x_1, thus variable v_2 became x_2, thus variable v_3 became x_3 ... and thus the program failed."

In a case study, the HOWCOME prototype successfully extracted the cause-effect chain for a failure of the GNU C compiler—with a precision far higher than static analysis. Although relying on several test runs to prove causality, the isolation of cause-effect chains requires no manual interaction and thus automates the most time-consuming part of program debugging: finding out how the failure came to be.

1. CAUSES, EFFECTS, AND FAILURES

One other obvious way to conserve programmer time is to teach machines how to do more of the low-level work of programming.

—Eric S. Raymond, *The Art of Unix Programming*

1.1 The Problem

Finding out the cause for a program failure is still the most time-consuming part of debugging. Why is this so? Consider the execution of a failing program as a series of program states. Each state consists of the program's variables and their values. Each program step takes the current program state and creates a new state (for instance, by computing and assigning new values); this continues until, finally, some invalid state is reached—the program fails (Figure 1).

In this view, each state is induced by its predecessor. But normally, only a fraction of a program state is relevant for computing the following state. If a variable is never read, for instance, it does not influence any state. Likewise, only a fraction of a program state is actually relevant for the failure. These relevant fractions form a *cause-effect chain*. The goal of debugging is to *break* this chain; to break it, one must *understand* it.

Traditionally, research in program comprehension has focused on *program analysis* to separate possible relevance from certain irrelevance—in an assignment $x := y$, for instance, variable z cannot be relevant for the value of x (unless y is an alias of z). However, several possible relevances still remain, and the number of possible relevances multiplies with the number of states. Consequently, the success of debugging still depends on the programmer's abilities.

1.2 The Solution

In this paper, we present a radically different approach to determine what is relevant for a program failure and what is not. Our approach relies on five novel building blocks:

Causality. In general, the cause of any event is a preceding event without which the event in question would not have occurred. Applied to our model of program states, this reads as follows: A variable v at some state in the failing program run $r_{✘}$[1] is *cause* of a failure if *we can alter the value of v such that the failure no longer occurs.* As an example, consider a pointer v that has an invalid value $x_{✘}$ at some point within $r_{✘}$. If (and only if) we find an alternate value $x_{✔}$ such that setting v to $x_{✔}$ and resuming execution makes the failure no longer occur, then we have proven that v's value $x_{✘}$ is a cause for the failure.

Alternate Run. On its own, the idea of causality does not help much. How should we know how to find the relevant variable v, and how should we know the "correct" value $x_{✔}$? The solution is to consider an *alternate run* $r_{✔}$ where the failure does *not* occur. Any variable v that has a different value in $r_{✔}$ and $r_{✘}$ is a candidate for causing the failure.

Systematic Narrowing. Even with an alternate run, the number of variables with differing values may still be large. The task is to reduce the difference systematically, resulting in a smaller number of causal variables. In earlier work, we had already shown how to narrow down *external* failure-inducing differences—for

[1]Throughout this paper, ✘ (read "fail") denotes failure, ✔ ("pass") stands for success, and **?** ("unresolved") stands for unresolved outcomes.

ICSE submission—2002 Buenos Aires, Argentina

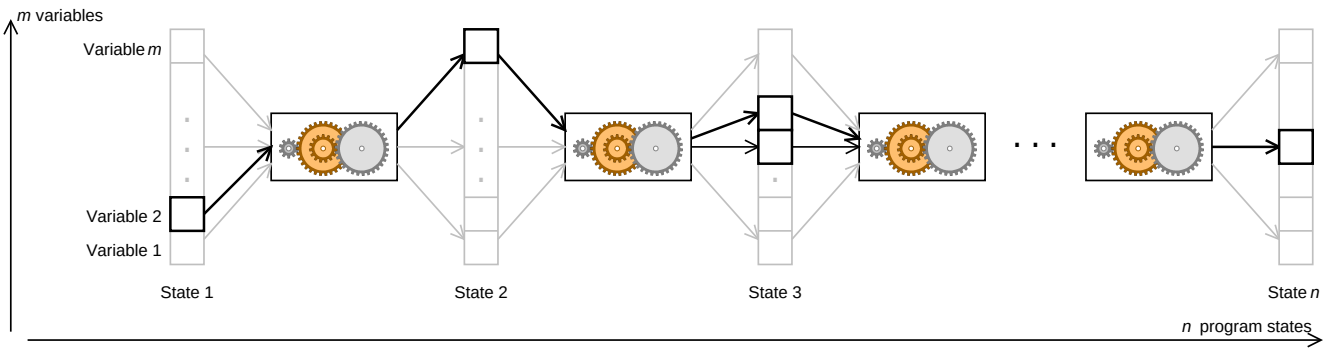

Figure 1: A cause-effect chain. In each state, out of m variables, only few are relevant for the failure.

instance, between two program versions [17] or two program inputs [6], using an algorithm called *Delta Debugging*. In this work, we apply Delta Debugging to narrow down the difference between *internal* program states—isolating automatically those differences that are relevant for the occurrence of the failure.

Memory Graphs. Two program states may not only differ by variable values, but also by *data structures*. We capture program states into *graphs* and use common subgraph algorithms to identify structural differences.

Multiple States. We can apply Delta Debugging on several states within a program run. For each state, this results in the set of variables and values that are relevant for the failure. The set of all relevant variables and values forms the *cause-effect chain* that leads to the failure: "Initially, variable v_1 was x_1, thus variable v_2 became x_2, thus variable v_3 became x_3 ... and thus the program failed."

To our knowledge, this is the first approach that automatically explains the causes of concrete program failures—requiring several test runs to establish causality, but with a precision far higher than static analysis.

1.3 This Paper

This paper is organized as follows. Section 2 discusses how our HOWCOME[2] prototype accesses, compares, and alters basic program states, using a sorting program as example. In Section 3, we show how to capture and compare memory graphs; Section 4 shows how HOWCOME successfully extracts a cause-effect chain from the GNU C compiler. Section 5 discusses related work; Section 6 closes with conclusions and consequences.

2. ISOLATING RELEVANT STATES

Something impossible occurred,
and the only solid information is that it really did occur.
So we must think backwards from the result
to discover the reasons.

—Brian W. Kernighan and Rob Pike,
The Practice of Programming

As a first example, consider the `sample` sorting program in Figure 3.[3] Normally, `sample` should sort its arguments numerically and print the sorted list, as in this run ($r_{✓}$):

```
$ gcc -g -o sample sample.c
$ ./sample 9 8 7
Output: 7 8 9
$ _
```

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

```
$ ./sample 11 14
Output: 0 11
$ _
```

Although the output is properly sorted, one of the arguments has been replaced by another number—in run $r_{✗}$, the argument `14` has been replaced by `0`.

We shall now show how HOWCOME extracts the cause-effect chain that leads to this failure.

2.1 Capturing State

As sketched in Section 1, we need to capture states from both runs $r_{✓}$ and $r_{✗}$. To do so, HOWCOME uses an ordinary command-line debugger—the GNU debugger (GDB). After interrupting the program at a specific breakpoint, HOWCOME queries GDB for the debuggee's variables and their values (Figure 2).

We capture the state of `sample` when execution reaches line 9—that is, immediately at the beginning of *shell_sort*. For the two runs $r_{✓}$ and $r_{✗}$, we obtain the variables and values listed in Table 1. (a and i occur in *shell_sort* and in *main*; the *shell_sort* instances are denoted as a' and i'.)

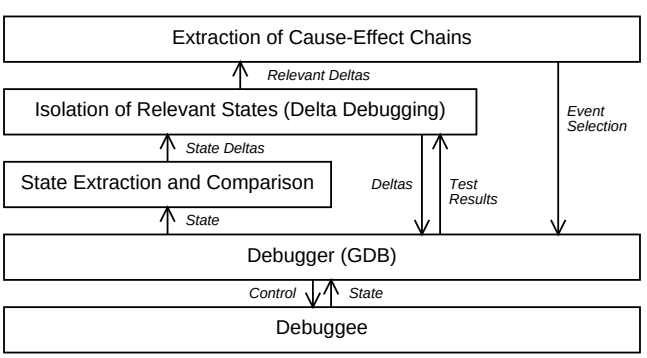

Figure 2: HOWCOME components

[2]Among others, HOWCOME stands for "HOWCOME outlines weird causes of mysterious errors".

[3]A HOWCOME demonstrator that isolates the cause-effect chain as shown here is available for LINUX PCs [20]

```
1  /* sample.c — Sample C program to be debugged */

   #include <stdio.h>
   #include <stdlib.h>
5
   static void shell_sort(int a[], int size)
   {
       int i, j;
       int h = 1;
10     do {
           h = h * 3 + 1;
       } while (h <= size);
       do {
           h /= 3;
15         for (i = h; i < size; i++)
           {
               int v = a[i];
               for (j = i; j >= h && a[j - h] > v; j -= h)
                   a[j] = a[j - h];
20             if (i != j)
                   a[j] = v;
           }
       } while (h != 1);
   }
25
   int main(int argc, char *argv[])
   {
       int *a;
       int i;
30
       a = (int *)malloc((argc - 1) * sizeof(int));
       for (i = 0; i < argc - 1; i++)
           a[i] = atoi(argv[i + 1]);

35     shell_sort(a, argc);

       printf("Output: ");
       for (i = 0; i < argc - 1; i++)
           printf("%d ", a[i]);
40     printf("\n");

       free(a);

       return 0;
45 }
```

Figure 3: This sorting program does not sort.

2.2 Narrowing Differences

We now compare the two states to find out what is different. The state differences, highlighted in Table 1, reflect exactly the differences in the program arguments. Denoting a different value of variable v in $r_{✔}$ and $r_{✘}$ by Δ_v, we obtain the set of differences

$$\mathcal{C} = \{\Delta_{argc}, \Delta_{argv[1]}, \Delta_{argv[2]}, \Delta_{argv[3]}, \Delta_{size}, \Delta_i, \Delta_{a[0]}, \Delta_{a[1]}, \Delta_{a[2]}, \Delta_{a'[0]}, \Delta_{a'[1]}, \Delta_{a'[2]}\}$$

Of all the differences Δ_v in $\mathcal{C}$, only some may be relevant for the failure. This is an application of Delta Debugging. The exact details of the Delta Debugging algorithm *dd* are summarized in Figure 4; here are the basics:

Delta Debugging takes two sets of differences—$c_{✔}$ and $c_{✘}$—and a testing function *test*. The testing function *test* applies a set of differences that lies between $c_{✔}$ and $c_{✘}$ and checks whether the failure occurs (✘) or not (✔) or whether the outcome is unresolved (**?**). Depending on the test outcome, Delta Debugging systematically narrows the difference between $c_{✔}$ and $c_{✘}$. Eventually, only a small set remains where every single difference is relevant.

In our setting, to *apply* a Δ_v means to set v in $r_{✔}$ to the value of v in $r_{✘}$. Applying $\Delta_{argv[1]}$, for instance, would change $argv[1]$ from `"9"` (its value in $r_{✔}$) to `"11"` (its value in $r_{✘}$). In practice, HOWCOME does this by issuing the GDB command `set variable argv[1] = "9"`.

Variable	Value in $r_{✔}$	Value in $r_{✘}$
$argc$	**4**	**5**
$argv[0]$	`"./sample"`	`"./sample"`
$argv[1]$	`"9"`	`"11"`
$argv[2]$	`"8"`	`"14"`
$argv[3]$	`"7"`	`0x0` (NIL)
i'	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
$size$	**4**	**3**

Variable	Value in $r_{✔}$	Value in $r_{✘}$
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 $r_{✔}$ and $r_{✘}$ in line 9

Our testing function *test* applies a set of differences and resumes execution to check the outcome. The "checking" part of *test* is typically part of an automated test suite—namely, the test that fails; in our case, *test* fails if the output begins with the number `0` and *test* passes if the output is `7 8 9`. In all other cases, the outcome is unresolved.

As an example, consider testing the set $\{\Delta_{argv[1]}\}$: *test* would set $argv[1]$ to `"11"` in line 9 and resume execution of $r_{✔}$. As $argv[1]$ is no longer accessed after the change, the output is still `7 8 9` and thus, $test(\{\Delta_{argv[1]}\}) = ✔$ holds. With this test run, we have proven that $argv[1]$ is irrelevant for the failure.

To apply Delta Debugging in our setting, we define the initial sets of differences $c_{✔} = \emptyset$ and $c_{✘} = \mathcal{C}$—that is, applying $c_{✔}$ to $r_{✔}$ results in $r_{✔}$ (hence $test(c_{✔}) = ✔$), and applying $c_{✘}$ to $r_{✔}$ results in $r_{✘}$ (hence $test(c_{✘}) = ✘$). *dd* will now narrow the difference between $c_{✔}$ and $c_{✘}$.

For reasons of efficiency, *dd* does not check differences one by one, but starts with large subsets instead. The actual run is shown in Table 2: A black square stands for a difference that has been applied. The first two steps simply verify that $test(c_{✔}) = ✔ \wedge test(c_{✘}) = ✘$ hold. In Step 3, *dd* applies half of the differences. The failure persists: We have narrowed down the difference from 12 to 6 variables.

dd repeats the tests until the minimal difference $\{\Delta_{a'[2]}\}$ is finally isolated: As demonstrated in Steps 1 and 8, changing the variable $a'[2]$ from 7 to 0 induces the failure. All other differences are irrelevant.

2.3 Isolating the Cause-Effect Chain

With our previous Delta Debugging run, we have only identified one element of the cause-effect chain. To obtain further elements, we must capture and compare more states during the program execution.

In this case study, we have set up HOWCOME to capture

Step	$\Delta a'[0]$	$\Delta a[0]$	$\Delta a'[1]$	$\Delta a[1]$	$\Delta a'[2]$	$\Delta a[2]$	$\Delta argc$	$\Delta argv[1]$	$\Delta argv[2]$	$\Delta argv[3]$	Δi	$\Delta size$	Output	Test
1	□	□	□	□	□	□	□	□	□	□	□	□	7 8 9	✔
2	■	■	■	■	■	■	■	■	■	■	■	■	0 11	✘
3	■	■	■	■	■	■	□	□	□	□	□	□	0 11 14	✘
4	■	■	■	□	□	□	□	□	□	□	□	□	7 11 14	**?**
5	□	□	□	■	■	■	□	□	□	□	□	□	0 9 14	✘
6	□	□	□	■	□	□	□	□	□	□	□	□	7 9 14	**?**
7	□	□	□	□	■	■	□	□	□	□	□	□	0 8 9	✘
8	□	□	□	□	■	□	□	□	□	□	□	□	0 8 9	✘
Result					■									

Table 2: Isolating a failure-inducing state difference

Let $\mathcal{C}$ be the set of all possible circumstances (i.e. state deltas). Let $test : 2^{\mathcal{C}} \to \{\text{✘}, \text{✔}, \text{?}\}$ be a testing function that determines for a test case $c \subseteq \mathcal{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).

Now, let $c_{\text{✔}}$ and $c_{\text{✘}}$ be test cases 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" test case (typically, $c_{\text{✔}} = \emptyset$ holds) and $c_{\text{✘}}$ is the "failing" test case.

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 $c'_{\text{✘}} - c'_{\text{✔}}$ is *1-minimal*—that is, no single circumstance 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 dd algorithm is defined as $dd(c_{\text{✔}}, c_{\text{✘}}) = dd_2(c_{\text{✔}}, c_{\text{✘}}, 2)$ with

$$dd_2(c'_{\text{✔}}, c'_{\text{✘}}, n) = \begin{cases} dd_2(c'_{\text{✔}}, c'_{\text{✔}} \cup \Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✘} \\ dd_2(c'_{\text{✘}} - \Delta_i, c'_{\text{✘}}, 2) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✔} \\ dd_2\big(c'_{\text{✔}} \cup \Delta_i, c'_{\text{✘}}, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✔} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}} - \Delta_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✘} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}}, \min(2n, |\Delta|)\big) & \text{else if } n < |\Delta| \\ (c'_{\text{✔}}, c'_{\text{✘}}) & \text{otherwise} \end{cases}$$

where $\Delta = c'_{\text{✘}} - c'_{\text{✔}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ with all Δ_i pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx (|\Delta| / n)$ holds.
The recursion invariant for dd_2 is $test(c'_{\text{✔}}) = \text{✔} \wedge test(c'_{\text{✘}}) = \text{✘} \wedge n \leq |\Delta|$.

Figure 4: The Delta Debugging algorithm in a nutshell. The function dd isolates the failure-inducing difference between two test cases $c_{\text{✔}}$ and $c_{\text{✘}}$. For full description of the algorithm and its properties, see [18].

the state at each function invocation and at each function return—that is, immediately after *main* in line 31 and after completion of *shell_sort* in line 37. Repeating the Delta Debugging runs for these states isolates the following differences:

- In line 31, the change of *argc* from 4 to 3 is relevant for the failure. The actual arguments are irrelevant.
- In line 37, the change of $a[0]$ from 7 to 0 is relevant for the failure.

We end up in the following cause-effect chain, as reported by HOWCOME. The values reported are the failure-inducing values of $r_{\text{✘}}$; the "instead of" values in parentheses are the alternative values of $r_{\text{✔}}$.

```
Cause-effect chain for './sample'
Arguments are 11 14 (instead of 9 8 7)
therefore at main, argc = 3 (instead of 4)
therefore at shell_sort, a[2] = 0 (instead of 7)
therefore at sample.c:37, a[0] = 0 (instead of 7)
therefore the run fails.
```

The whole run required 32 tests, or 3.9 seconds.[4] We could easily plug this into an automated testing environment, and whenever a test fails, our method could automatically explain the causes for the failure.

2.4 Fixing the Error

Didn't we forget something? We still need to actually debug the program. To fix the error, the programmer must *break the cause-effect chain*. To do so, she must

1. *compare* the failure-inducing with the *intended* values
2. *fix* the program such that the failure-inducing values no longer occur—but in a *most general way*, such that other similar failures are excluded as well.
3. *verify* that the fix was successful, i.e. the failure no longer occurs.

In our case,

1. a change in $a[2]$ triggers the failure, which means that $a[2]$ is actually accessed by the program. However, only the two elements $a[0]$ and $a[1]$ have been allocated and set.
2. To fix the error in the most general way, *shell_sort* in line 35 must be invoked with the correct number of elements in a—that is, $argc - 1$ instead of *argc*.
3. After the fix, repeating the test shows that the failure no longer occurs. Success!

Steps 1 and 2 indicate why debugging will always contain manual activities—fixing an error means writing programs.[5] However, fixing the error is mostly trivial once one has understood how and why the failure occurs. And this is where automatic isolation of cause-effect chains can be helpful.

3. COMPARING DATA STRUCTURES

When a fatal error occurs, the immediate cause may be that a pointer has been trashed due to a previous fandango on core. However, this fandango may have been due to an earlier *fandango, so no amount of analysis will reveal (directly) how the damage occurred.*

— The Jargon File 4.1.0 "Secondary Damage"

Here comes bad news. Extracting cause-effect chains as demonstrated so far works only for a small fraction of real-life programs. The reason has a name: pointers. A simple

[4] All times were measured on a LINUX PC with a 500 MHz Pentium III processor. The time given is real running time of our HOWCOME prototype.

[5] This also indicates why there can be no automated process that says "Fix the error at line n": No automated process can determine which element of the cause-effect chain is different from the intended value—or determine the most general way to fix an error. At best, such processes can be based on *heuristics* such that "v has a bad value; the statement where v was last set is most likely to be erroneous".

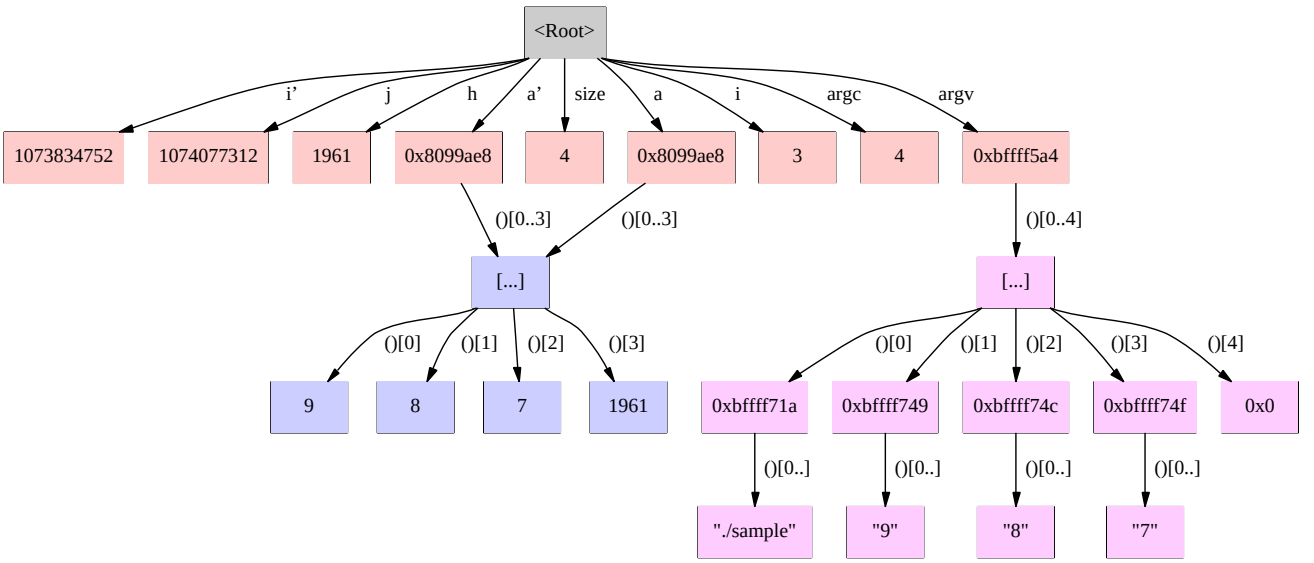

Figure 5: A memory graph showing the state of `sample` in line 9. Compare the view to $r_{✓}$ in Table 1.

(*name*, *value*) table as in Section 2 is not accurate for capturing large data structures with pointers, aliases, and references. For instance, let a pointer p have a value of `0x814abba` in $r_{✓}$ and `0x814beef` in $r_{✗}$. We may easily identify this difference, but we cannot set p in $r_{✓}$ to the value found in $r_{✗}$ and expect anything reasonable to happen. Rather than comparing pointer values, we must take into account the objects pointed to and the resulting data structures.

3.1 Capturing State as Memory Graphs

To solve the problem, we capture the state of a program as a so-called *memory graph*. A memory graph contains all values and all variables of a program, but represents operations like variable access, pointer dereferencing, struct member access, or array element access by edges.

As an example, consider the memory graph obtained from the `sample` program in run $r_{✓}$, shown in Figure 5. The immediate descendants of the $\langle Root \rangle$ vertex are the base variables of the program. By dereferencing a (following the edge labeled `()[0..3]`)[6], we find the array pointed to by a, and its elements $a[0], \ldots, a[3]$. (Note that the memory graph properly reflects that a and a' point to the same array, a property that is not visible from a (*name*, *value*) table like Table 1.) Likewise, *argv* points to an array of 5 pointers, each pointing to a character string holding a program argument.

How does one obtain such a memory graph? The basic idea is to query the base variables of a program and to systematically unfold all data structures encountered; if two values share the same type and address, they are merged to a single vertex. The C language brings caveats, though:

Uninitialized pointers may point to garbage. The state extractor of HOWCOME handles this by checking whether a pointer points to a valid memory area such as the stack, the heap, or static memory.

Array sizes may be unknown. This typically happens if the array was allocated dynamically on the heap (such as a or a' in Figure 5). One technique to handle these problems is to query the internal *malloc* structures to check the number of elements in the area pointed to.

Types may have different interpretation. In the programming language C, pointers can be freely casted to integers, other pointer types, generic pointer types, and vice versa. The state extractor optimistically assumes that the declared types, as reported by the debugger, hold. However, in the case of *unions* (i.e. variant records without explicit tag field), the state extractor must rely on *heuristics* or on *external specifications* to follow the most likely interpretation.

More details of capturing and visualizing memory graphs, as well as the formal construction are available [19].

3.2 Comparing Memory Graphs

Assume we have two memory graphs $G_{✓}$ and $G_{✗}$, representing a state from a passing run $r_{✓}$ and a failing run $r_{✗}$, respectively. In Section 2.2, we simply compared variable values to detect differences. This no longer works for memory graphs:

- $G_{✓}$ may contain variables that do not exist in $G_{✗}$, or vice versa (i.e. the set of vertices differs)
- Pointers in $G_{✓}$ may point to other variables as in $G_{✗}$ (i.e. the set of edges differs)

As an example, consider the two memory graphs in Figure 6. What has changed?

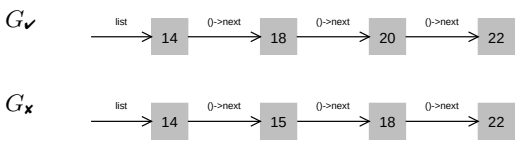

Figure 6: Graph differences

As a human, you can quickly see that the element 15 has been inserted into the list and the element 20 has been deleted. To detect this automatically for arbitrary data

[6] In a memory graph, each variable name is constructed from the incoming edge, where the placeholder () stands for the name of the parent.

structures, though, requires some graph operations. The solution is to compute a *maximum common subgraph* (MCS) of $G_{\checkmark}$ and $G_{\times}$: Every vertex that is not in both $G_{\checkmark}$ and $G_{\times}$ has either been inserted or deleted.

Figure 7 shows the MCS for $G_{\checkmark}$ and $G_{\times}$ as introduced above—drawn as a *matching* between $G_{\checkmark}$ and $G_{\times}$.[7] It is plain to see that the element 15 in $G_{\times}$ has no match in $G_{\checkmark}$; likewise, the element 20 in $G_{\checkmark}$ has no match in $G_{\times}$.

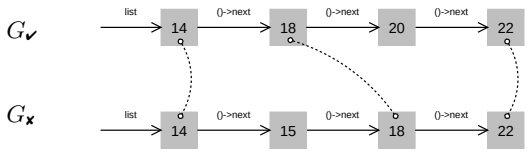

Figure 7: Subgraph matching

To compute the MCS efficiently, we use the algorithms of Barrow and Burstall [2], creating a *correspondence graph* between $G_{\checkmark}$ and $G_{\times}$; the maximum clique of the correspondence graph then corresponds to the MCS [3]. As a domain-specific optimization, we first determine an initial matching by traversing both graphs in parallel, starting with the $\langle Root \rangle$ vertex. (Details can be found in [19]).

3.3 Applying Graph Differences

We not only need a means to detect differences in data structures, we also need a means to *apply* these differences. We shall first concentrate on applying *all* differences between $r_{\checkmark}$ and $r_{\times}$ to $r_{\checkmark}$—that is, we compute debugger commands that change the state of $r_{\checkmark}$ such that, eventually, its memory graph is identical to $G_{\times}$.

For this purpose, we require three graph traversals. During these steps, $G_{\checkmark}$ is transformed to become equivalent to $G_{\times}$; each graph operation is translated into debugger commands which perform the equivalent operation on $r_{\checkmark}$. Again, we use the example graphs from Figure 6.

1. **(Set and create variables)** *For each vertex $v_{\times}$ in $G_{\times}$ without a matching vertex in $G_{\checkmark}$, create a new vertex $v_{\checkmark}$ as a copy of $v_{\times}$; $v_{\times}$ is matched to $v_{\checkmark}$.* After this step, each vertex $v_{\times}$ has a matching vertex $v_{\checkmark}$.

   Figure 8 shows our example graphs after this step.

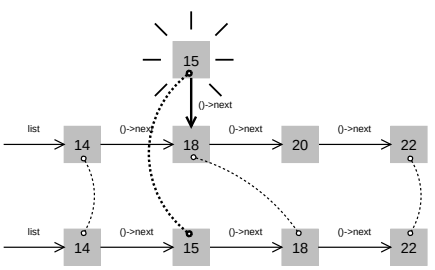

Figure 8: Creating new variables

   To generate debugger commands, for each addition of a vertex $v_{\checkmark}$, we identify the appropriate variable v in $r_{\times}$ and generate a command that

   - creates v in $r_{\checkmark}$, if it does not exist yet;
   - sets v to the value found in $r_{\times}$.

   In our example, we would obtain the GDB commands

```
set variable $m1 = (List *)malloc(sizeof(List))
set variable $m1->value = 15
set variable $m1->next = list->next
```

2. **(Adjust pointers)** *For each pointer vertex $p_{\times}$ in $G_{\times}$, determine the matching vertex $p_{\checkmark}$ in $G_{\checkmark}$. Let $*p_{\times}$ and $*p_{\checkmark}$ be the vertices that $p_{\times}$ and $p_{\checkmark}$ point to, respectively (reached via the outgoing edge). If $*p_{\checkmark}$ does not exist, or if $*p_{\checkmark}$ and $*p_{\times}$ do not match, adjust $p_{\checkmark}$ such that it points to the matching vertex of $*p_{\times}$.*

   In our example, the *next* pointers from 14 to 18 and from 18 to 20 must be adjusted; the resulting graphs are shown in Figure 9.

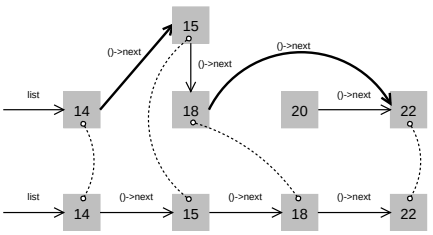

Figure 9: Adjusting pointers

   Again, any adjustment translates into appropriate debugger commands.

3. **(Delete variables)** *Each remaining vertex $v_{\checkmark}$ in $G_{\checkmark}$ that is not matched in $G_{\times}$ must be deleted, including all incoming and outgoing edges.* After this last step, $G_{\checkmark}$ is equal to $G_{\times}$.

   In our example, the vertex 20 must be deleted; the resulting graphs are shown in Figure 10.

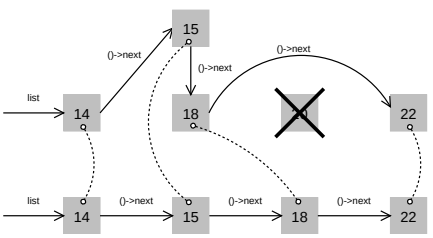

Figure 10: Deleting variables

   Such a deletion of a vertex v translates into debugger commands that set all pointers that point to v to NIL, such that v becomes unreachable. Additionally, one might want to free the associated dynamic memory.

After these three steps, we have successfully transferred the changes in a data structure from a run $r_{\times}$ to a run $r_{\checkmark}$.

3.4 Applying Partial State Changes

For the purpose of Delta Debugging, transferring *all changes* is not sufficient: We need to apply *partial* state changes as well. For this purpose, we associate a delta Δ_v with each vertex v in $G_{\checkmark}$ or $G_{\times}$ that is not contained in the matching. If v is in $G_{\checkmark}$ only, applying Δ_v is supposed to delete it from $G_{\checkmark}$; if v is in $G_{\times}$ only, applying Δ_v must add it to $G_{\checkmark}$.

[7] An edge is part of the matching (= the common subgraph) if its vertices match; this is not the case in this example.

Let $c_{\text{✘}}$ be the set of all deltas so obtained; as always, $c_{\text{✔}} = \emptyset$ holds. In Figure 7, for instance, we would obtain two deltas $c_{\text{✘}} = \{\Delta_{15}, \Delta_{20}\}$. The idea is that Δ_{15} is supposed to add vertex 15 to $G_{\text{✔}}$; Δ_{20} should delete vertex 20 from $G_{\text{✔}}$. Applying both Δ_{15} and Δ_{20} should change $G_{\text{✔}}$ to $G_{\text{✘}}$.

To apply a *subset* $c \subseteq c_{\text{✘}}$ only, we run the state transfer method of Section 3.3, but with the following differences:

- In Step 1 and Step 3, we generate or delete a vertex v only if Δ_v is in c.
- In Step 2, we adjust a pointer $p_{\text{✔}}$ with a matching $p_{\text{✘}}$ only if $\Delta_{*p_{\text{✘}}}$ is in c or $\Delta_{*p_{\text{✔}}}$ is in c.

As an example, let us apply $c = \{\Delta_{15}\}$ only. Step 1 generates the new vertex; Step 2 adjusts the pointer from 14 such that it points to 15. However, the pointer from 18 to 20 is not changed, because Δ_{20} is not in c. We obtain a graph (and appropriate GDB commands) where only element 15 has been inserted (Figure 11).

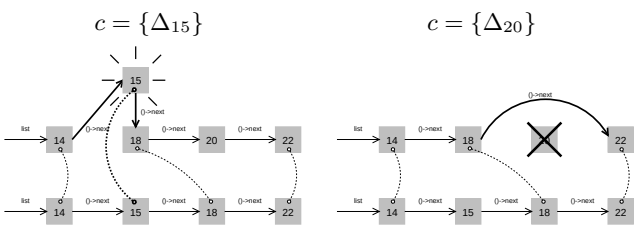

Figure 11: Applying partial state changes

Likewise, if we apply $c = \{\Delta_{20}\}$ only, Step 1 does not generate a new vertex; however, Step 2 adjusts the pointer from 18 such that it points to 22, and Step 3 properly deletes element 20 from the graph.

4. CASE STUDY: GCC EATS THE STACK

If you never know failure, how can you know success?
— The Matrix

We have now all building blocks in place to apply our method to a real-life program with a real-life bug. The GNU compiler collection is anything else but trivial; the C compiler alone (GNU CC) has more than 100,000 lines of code.[8]

The C program `bug.c` in Figure 12 causes GNU CC to crash—at least, when using the LINUX version 2.95.2 with optimization enabled.[9]

```
t(double z[],int n){int i,j;for(;;){i = i + j + 1;z[i] =
z[i] * (z[0] + 0);}return z[n];}
```

Figure 12: Compiling `bug.c` crashes GNU CC

Before crashing, GNU CC grabs all available memory for the function call stack, such that other processes may run out of resources and die. The latter can be prevented by limiting the stack memory available to GNU CC, but the effect remains—the `cc1` component crashes:

```
$ (ulimit -H -s 256; gcc -O bug.c)
gcc: Internal compiler error:
     program cc1 got fatal signal 11
$ _
```

[8]The GNU CC C sources contain 104,176 semicolons.
[9]The error is fixed in GNU CC 2.95.4 and later.

In previous work [6], we had already shown how to simplify failure-inducing input like `bug.c`. Actually, `bug.c` was obtained by simplifying a larger program automatically; every single character in `bug.c` is relevant for the failure. While this tells us about the external root cause of the problem, we cannot see its internal effects within `cc1`. Therefore, we shall now attempt to isolate the cause-effect chain within `cc1` that leads to the failure.

4.1 Setting up GNU CC

To isolate the cause-effect chain within `cc1`, we need:

An alternate run. Our run $r_{\text{✘}}$ has already been set. We now need a run $r_{\text{✔}}$ that passes the test.

Finding a program which GNU CC compiles without crashing is not difficult. However, as we are interested in isolating small state differences, we should better begin with a *small input difference*. Any valid subset of `bug.c` does the job; Figure 13 shows the alternate input `ok.c` which is identical to `bug.c`, with one exception—the "+ 0" has been removed.

```
t(double z[],int n){int i,j;for(;;){i = i + j + 1;z[i] =
z[i] * (z[0]);}return z[n];}
```

Figure 13: Compiling `ok.c` works fine

In contrast to `bug.c`, compiling `ok.c` works fine. Our task is now to identify how the extra "+ 0" in `bug.c` causes the failure.

An automated test. The next item on our list is an automated test that decides whether the failure occurs or not. For crashing programs, HOWCOME brings a predefined *test* function that queries GDB for the *backtrace* at the moment of the crash—that is, the current program counter and the stack of calling functions at the moment of the crash. The predefined *test* then returns ✘ if the given run crashes at the same location as $r_{\text{✘}}$, ✔ if the program exits normally, and **?** otherwise.

In our case, the backtrace of `cc1` shows that it crashes in *if_then_else_cond*() at the location `combine.c:6788`; *test* will return ✘ if and only if the given run crashes at this very location.

A choice of events. We must decide at which events we should capture and compare program states—that is, moments in time during program execution.

HOWCOME provides various strategies for choosing events automatically (Figure 14); for instance, the event granularity can be finer as the failure comes closer. User-specified events are possible as well.

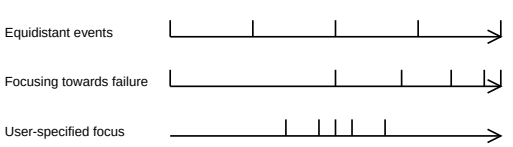

Figure 14: Strategies for choosing events

Here, we restrict ourselves to three equidistant events; HOWCOME will interrupt `cc1` and capture and compare states as soon as execution reaches the *main*, *combine_instructions*, and *if_then_else_cond* functions.

4.2 Extracting the Cause-Effect Chain

We now describe what happens while HOWCOME captures and narrows states at the various events within `cc1`.

4.2.1 At main

HOWCOME starts by capturing the two program states of $r_{✓}$ and $r_{✗}$ in *main*. Both graphs $G_{✓}$ and $G_{✗}$ have 27139 vertices and 27159 edges (Figure 15); to squeeze them through the bottleneck of the GDB command-line requires 15 minutes each. (This is why we look at three events only.)

After 12 seconds, HOWCOME determines that exactly one vertex is different in $G_{✓}$ and $G_{✗}$—namely *argv*[2], which is `"bug.i"` in $r_{✗}$ and `"ok.i"` in $r_{✓}$. These are the names of the preprocessed source files as passed to `cc1` by the GNU CC compiler driver. This difference is minimal, so HOWCOME does not need a Delta Debugging run to narrow it further.

4.2.2 At combine_instructions

As *combine_instructions* is reached, GNU CC has already generated the intermediate code (called RTL for "register transfer list") which is now optimized. Unfortunately, this RTL code brings a problem for HOWCOME.

The tree containing the RTL code contains *unions* (i.e. variant records without explicit tag field), where the proper interpretation must be chosen. To disambiguate unions, HOWCOME implements some *heuristics*—for instance, to unfold each single interpretation and choose the one resulting in the largest graph. These heuristics have worked well for all GNU CC data structures—except for the RTL tree.

Each node in the RTL tree contains a dynamic array of up to 20 unions, each with 10 members—in other words, each RTL tree node has up to 10^{20} possible interpretations. To avoid HOWCOME work its way through this combinatorial explosion, we had to provide a *union disambiguation table* that tells HOWCOME which union members to unfold, based on the (specified) union tag.[10]

Given the disambiguation table, HOWCOME quickly captured the graphs $G_{✓}$ with 42991 vertices and 44290 edges as well as $G_{✗}$ with 43147 vertices and 44460 edges. The common subgraph of $G_{✓}$ and $G_{✗}$ has 42637 vertices; thus, we have 871 vertices that have been either added in $G_{✗}$ or deleted in $G_{✓}$.

[10]Such a table must be supplied once for each union declaration. However, unions could also be disambiguated automatically. For each union instance v, one could determine the moment its location is next accessed for read or write (using appropriate debugger features), and then examine v's interpretation by the debuggee. Unfortunately, this method requires one test run for each union instance of each state—and we did not want to wait for so long.

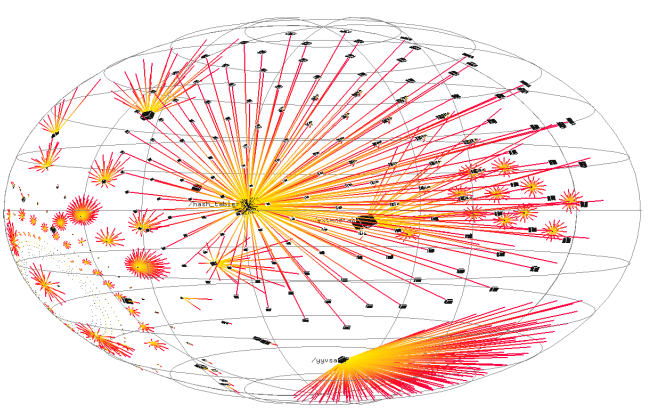

Figure 15: The GNU CC $G_{✓}$ memory graph (excerpt)

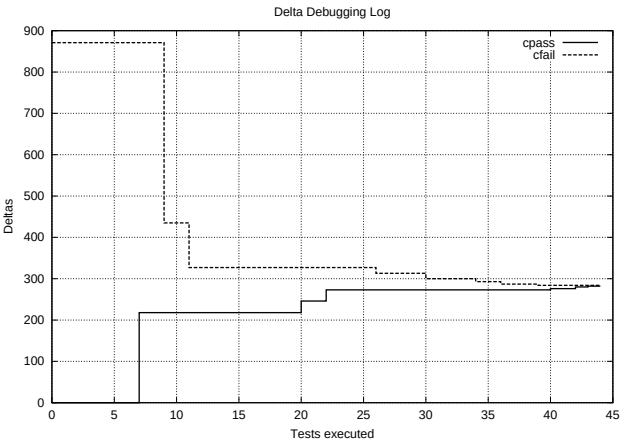

Figure 16: Narrowing at *combine_instructions*

The deltas for these 871 vertices are now subject to Delta Debugging; the resulting run is shown in Figure 16. After only 44 tests, we have narrowed the failure-inducing difference to one single vertex, created with the GDB commands

```
set variable $m9 = (struct rtx_def *)malloc(12)
set variable $m9->code = PLUS
set variable $m9->mode = DFmode
set variable $m9->jump = 0
set variable $m9->fld[0].rtx = loop_mems[0].mem
set variable $m9->fld[1].rtx = $m10
set variable first_loop_store_insn->fld[1].rtx->
  fld[1].rtx->fld[3].rtx->fld[1].rtx = $m9
```

That is, the failure-inducing difference is now the insertion of a node in the RTL tree containing a `PLUS` operator—the effect of the initial change "+ 0" from `ok.c` to `bug.c`. Each of the tests required about 20 to 27 seconds of HOWCOME time, and 1 second of GNU CC time.

4.2.3 At if_then_else_cond

At this last event, HOWCOME captured the graphs $G_{✓}$ with 47071 vertices and 48473 edges as well as $G_{✗}$ with 47313 vertices and 48744 edges. The common subgraph of $G_{✓}$ and $G_{✗}$ has 46605 vertices; 1224 vertices have been either added in $G_{✗}$ or deleted in $G_{✓}$.

Again, HOWCOME runs Delta Debugging on the deltas of the 1224 differing vertices (Figure 17). As every second test fails, the difference narrows quickly. After 15 tests, HOWCOME has isolated a minimal failure-inducing difference. It consists of one single pointer adjustment, created with the GDB command

```
set variable link->fld[0].rtx = &link
```

This final difference is the difference that causes GNU CC to fail: It creates a cycle in the RTL tree—the pointer *link*→*fld*[0].*rtx* points back to *link*! The RTL tree is no longer a tree, and this causes endless recursion in *if_then_else_cond*, eventually crashing `cc1`.

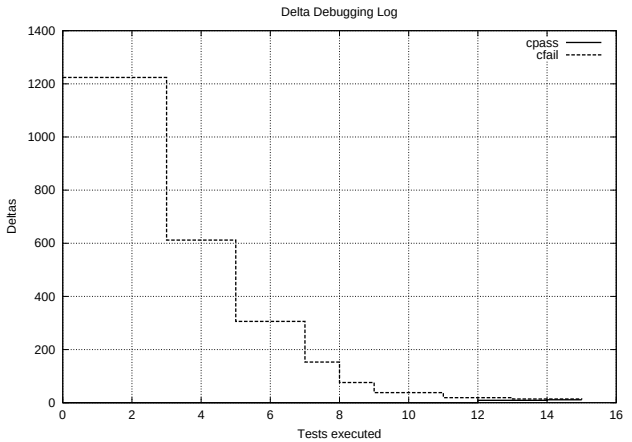

Figure 17: Narrowing at *if_then_else_cond*

4.3 The GNU CC Cause-Effect Chain

The total cause-effect chain for `cc1`, as reported by HOWCOME, looks like this:

```
Cause-effect chain for './gcc/cc1'
Arguments are -O bug.i (instead of -O ok.i)
therefore at main, argv[2] = "bug.i"
  (instead of "ok.i")
therefore at combine_instructions,
  *first_loop_store_insn->fld[1].rtx->
  fld[1].rtx->fld[3].rtx->fld[1].rtx =
  <new variable>
therefore at if_then_else_cond,
  link->fld[0].rtx = &link (instead of i1dest)
therefore the run fails.
```

With this output, the programmer can easily follow the cause-effect chain from the root cause (the passed arguments) via an intermediate effect (a new node in the RTL tree) to the final effect (a cycle in the RTL tree). The whole run was generated automatically; no manual interaction was required. HOWCOME required 6 runs to extract GNU CC state (each taking 15–20 minutes) and 3 Delta Debugging runs (each taking 8–10 minutes) to isolate the failure-inducing differences.[11]

Again, to fix the error, the programmer must check which of the reported values is at fault. Assuming that the state at *combine_instructions* is fine and that the RTL cycle is not intended, the programmer could re-run HOWCOME with a finer granularity to isolate the moment where the RTL cycle came to be. In future, we will extend HOWCOME to isolate such *transitions* of causes and effects automatically as well.

4.4 More Case Studies

Besides debugging GNU CC and the `sample` program from Section 2, we have applied HOWCOME to some more well-known programs. These case studies are part of the HOWCOME regression test suite. They are not related to a program failure; nonetheless, they reflect how a small change in the input propagates in the program state, and which state fractions are relevant for generating changed output.

[11]The running times are shortcomings of our prototype: Switching from GDB to direct memory access should speed up state access by at least 1–2 magnitudes; reimplementing HOWCOME in a compiled language instead of Python could speed up Delta Debugging by up to 1 magnitude.

Event	Edges	Vertices	Deltas	Tests
`sample` at *main*	26	26	12	4
`sample` at *shell_sort*	26	26	12	7
`sample` at *sample.c*:37	26	26	12	4
`cc1` at *main*	27139	27159	1	0
`cc1` at *combine_instructions*	42991	44290	871	44
`cc1` at *if_then_else_cond*	47071	48473	1224	15
`bison` at *open_files*	431	432	2	2
`bison` at *initialize_conflicts*	1395	1445	431	42
`diff` at *analyze.c*:966	413	446	109	9
`diff` at *analyze.c*:1006	413	446	99	10
`gdb` at *main.c*:615	32455	33458	1	0
`gdb` at *exec.c*:320	34138	35340	18	7

Table 3: Summary of case studies

Table 3 gives a short summary of the isolation details:

- In the `bison` parser generator, a shift/reduce conflict in the grammar input causes the variable *shift_table* to be altered, which in turn generates a warning.
- In the `diff` file comparison program, printing of differences is controlled by *changes*, whose value is again caused by *files→changed_flag*.
- Invoking the `gdb` debugger with a different debuggee changes 18 variables, but only the change in the variable *arg* is relevant for the actual debuggee selection.

In all cases, the resulting failure-inducing difference contained only one element; the number of tests was at most 42. No union disambiguation tables were needed.[12]

5. RELATED WORK

For the Snark's a peculiar creature, that won't
Be caught in a commonplace way.
Do all that you know, and try all that you don't:
Not a chance must be wasted to-day!

—Lewis Carroll, *The Hunting of the Snark*

5.1 Algorithmic Debugging

Algorithmic debugging [12] is a means to automate the debugging *process*. The idea is to isolate a failure-inducing clause in a PROLOG program by querying systematically whether subclauses hold or not. The query is resolved either manually by the programmer or by an oracle relying on an external specification.

Our approach is a divide-and-conquer scheme, too; however, it uses *one single test* to assess divisions of the problem space, which makes automation far easier. Also, our approach focuses on isolating failure-inducing program *state*, while Shapiro's approach focuses on isolating failure-inducing program *code*. One could easily combine both approaches (for instance, by querying "`Is a[2] = 0 correct?`") to interactively isolate the code fragment whose execution causes the state to become failure-inducing.

[12]The ultimate test would be, of course, to apply HOWCOME onto itself. Unfortunately, GDB does not support Python, the language HOWCOME is written in. However, whenever a HOWCOME regression test failed, we routinely ran Delta Debugging on the generated GDB commands and thus quickly isolated the failure causes.

5.2 Program Slicing

Program slicing [13, 14] is a means to facilitate debugging by focusing on relevant program fragments. A *slice* for a location p in a program consists of all other locations that could possibly influence p ("all locations that p depends upon"). As an example, consider the code fragment

```
if (p) { x = x * y; }
```

Here, the variable `x` is control dependent on `p` and data dependent on `x` and `y` (but *not* on, say, `z`); the slice of `x` would also include earlier dependencies of `p`, `x`, and `y`. The slice allows the programmer to focus on relevant statements; a slice also has the advantage that it is valid for *all possible program runs* and thus needs to be computed only once.

In practice, slicing is not yet as useful as would be expected, since each statement is quickly dependent on many other statements. The end result is often a program slice which is not dramatically smaller than the program itself—the program dependencies are too coarse [8]. Also, data and control-flow analysis of real-life programs is non-trivial. For programs with pointers, the necessary points-to analysis makes dependencies even more coarse [7].

Dynamic slicing [1, 4, 9] is a variant of slicing that takes a *concrete program run* into account. The basic idea is that within a concrete run, one can determine more accurate data dependencies between variables, rather than summarizing them as in static slicing. In the dynamic slice of `x`, as above, `x` is dependent on `x`, `y`, and `p` only if `p` was found to be true.

In *cause-effect chains*, `p`, `x`, and `y` are the cause for the value of `x` if and only if altering them also changes the value of `x`, as proven by test runs. If `x` is zero, for instance, `p` can never be a cause for the value of `x`, because `x` will never alter its value; `y` cannot be a cause, either. Consequently, cause-effect chains have a far higher precision than static or dynamic slices; furthermore, neither analysis nor complete knowledge of the program is required.

On the other hand, cause-effect chains require several test runs (which may or may not be faster than analysis), apply to a single program run only, and give no hints on the involved statements. The intertwining of program analysis and experimentation promises several mutual advantages.

5.3 Dicing

Dicing [10] determines the *difference* of two program slices. For instance, a dynamic dice could contain all the statements that may have influenced a variable v at some location in a failing run $r_{\text{✘}}$, but not in a passing run $r_{\text{✔}}$. The dice is likely to include the statement relevant for the value of v.

As discussed in Footnote 5, the idea that an automated process could isolate "the" erroneous statement can only be based on heuristics. Also, dicing does not explain *why* the erroneous statement was executed in $r_{\text{✘}}$ and not in $r_{\text{✔}}$—that is, how the altered control flow came to be (which the cause-effect chain explains). However, a dice can give good suggestions which events to include in the cause-effect chain.

5.4 Testing for Debugging

Surprisingly, there are very few applications of testing for purposes of debugging or program understanding. Our own contributions [6, 17] have already been mentioned. Specifically related to our GNU CC case study is the isolation of failure-inducing RTL optimizations in a compiler, using simple binary search over the optimizations applied [15].

6. CONCLUSION AND CONSEQUENCES

If brute force doesn't solve your problem,
you're just not using enough.
—Anonymous

Cause-effect chains explain the causes of program failures automatically and effectively. All that is required is an automated test, two comparable program runs and a means to access the state of an executable program. Although relying on several test runs to prove causality, the isolation of cause-effect chains requires no manual interaction and thus saves valuable developer time.

As the requirements are simple to satisfy, we expect that future automated test environments will come with an automatic isolation of cause-effect chains. Whenever a test fails, the cause-effect chain could be automatically isolated, thus showing the programmer not only *what* has failed, but also *why* it has failed. Although breaking the cause-effect chain by fixing the program is still manual work, we expect that the time spent for debugging will be reduced significantly.

All this optimism should be taken *cum grano salis*, as there is still much work to do. Our future work will concentrate on the following topics:

Optimization. As stated in Section 4.3, HOWCOME could be running faster by several orders of magnitude by bypassing the GDB bottleneck and re-implementing HOWCOME in a compiled language. Regarding Delta Debugging, we are working on *grouping variables* such that variables related by occurring in the same function or module are changed together, rather than having a random assignment of variables to subsets.

Program analysis. As hinted at in Section 5, the integration of program analysis could make extracting cause-effect chains much more effective. For instance, variables that cannot influence the failure in any way could be excluded right from the start. Shape analysis [16] could help to identify invalid data structures. Dicing can suggest relevant events during a program run.

Event selection and isolation. The cause-effect chains as shown so far contain only three elements; however, there is no reason not to include, say, 100 elements or more, especially if states are small and if testing is cheap. Our future work will concentrate on how to isolate the *relevant* events out of the cause-effect chain—that is, those events where a variable is first set to a failure-inducing value—and how to initially select good candidates for relevant events.

Finding alternate runs. Delta Debugging requires that a passing run is available, such that a there is a state difference that can be narrowed. In cases where no such run is available, we could search for *alternate paths* in a failing run that make the program pass the test. Such paths could be determined by isolating failure-inducing branches in the control flow.

Greater state. Right now, our method only works on the state that is accessible via the debugger. However, differences may also reside *outside* of the program state—for instance, a file descriptor may have the same value in $r_{\text{✘}}$ and $r_{\text{✔}}$, but be tied to a different file. We are working on how to capture such external differences.

Presentation. Presenting variables and values textually is appropriate for atomic values. However, if *structural* changes turn out to be relevant, a graphical presentation (like the figures in this paper) is much more accurate. Such presentations could also be animated, reflecting the dynamics of the program.

More case studies. Last but not least, we need many more large case studies to gain further experience with our method. We are currently building a so-called *debugging server* around HOWCOME where anyone can submit failing programs via the Web to have their cause-effect chains isolated automatically. As a side effect, this will generate a database of case studies.

Overall, we expect that programmers will be able to pass much of the boredom and monotony of debugging onto their machines. Eventually, debugging may become as automated as testing—not only detecting *that* a failure occurred, but also *why* it occurred.

Acknowledgments. The concept of isolating cause-effect chains was born after a thorough discussion with Jens Krinke on the respective strengths and weaknesses of program slicing and Delta Debugging. Tom Zimmermann implemented the initial memory graph extractor and the common subgraph algorithms. Holger Cleve, Stephan Diehl, Petra Funk, Kerstin Reese, and Tom Zimmermann provided substantial comments on earlier versions of this paper.

Figure 5 was generated by AT&T's DOT graph layouter; Figure 15 is a screenshot of Tamara Munzner's H3VIEWER.

The work on Delta Debugging was funded by the Deutsche Forschungsgemeinschaft, grant Sn 11/8-1.

More information on isolation of cause-effect chains as well as a HOWCOME demonstration program are available at the delta debugging web site,

http://www.st.cs.uni-sb.de/dd/

7. REFERENCES

[1] 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.

[2] H. G. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4(4):83–84, 1976.

[3] C. Bron and J. Kerbosch. Algorithm 457—Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[4] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In Nierstrasz and Lemoine [11], pages 303–321.

[5] M. J. Harrold, editor. *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Portland, Oregon, Aug. 2000.

[6] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In Harrold [5], pages 135–145.

[7] M. Hind and A. Pioli. Which pointer analysis should I use? In Harrold [5], pages 113–123.

[8] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *Proc. 2nd ACM SIGSOFT symposium on the Foundations of Software Engineering (FSE-2)*, pages 2–10, New Orleans, Lousiana, Dec. 1994.

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

[10] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *2nd International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California.

[11] O. Nierstrasz and M. Lemoine, editors. *Proc. ESEC/FSE'99 – 7th European Software Engineering Conference / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, Toulouse, France, Sept. 1999. Springer-Verlag.

[12] E. Y. Shapiro. *Algorithmic Program Debugging*. PhD thesis, MIT Press, 1982. ACM Distinguished Dissertation.

[13] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

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

[15] D. B. Whalley. Automatic isolation of compiler errors. *ACM Trans. Prog. Lang. Syst.*, 16(5):1648–1659, 1994.

[16] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proc. of CC 2000: 9th International Conference on Compiler Construction*, Berlin, Germany, Mar. 2000.

[17] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In Nierstrasz and Lemoine [11], pages 253–267.

[18] A. Zeller. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2001. To appear; available online [20].

[19] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Proc. of the Dagstuhl Seminar 01211 "Software Visualization"*, Lecture Notes in Computer Science, Dagstuhl, Germany, May 2001. Springer-Verlag. To appear; available online [20].

[20] Delta debugging web site. http://www.st.cs.uni-sb.de/dd/.