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 sequence of program states. Each state induces the following state, up to the failure. Which variables and values of a program state are 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—by assessing the outcome of altered executions to determine wether a change in the program state makes a difference in the test outcome. Applying Delta Debugging to multiple states of the program automatically reveals the *cause-effect chain* of the failure—that is, the variables and values that caused the failure.

In a case study, our prototype implementation successfully isolated the cause-effect chain for a failure of the GNU C compiler: "Initially, the C program to be compiled contained an addition of 1.0; this caused an addition operator in the intermediate RTL representation; this caused a cycle in the RTL tree—and this caused the compiler to crash."

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, diagnostics, testing tools, tracing*

General Terms

Algorithms, Reliability, Experimentation, Verification

Keywords

Automated debugging, program comprehension, testing, tracing

1. INTRODUCTION

Program debugging is commonly understood as the process of identifying and correcting errors in the program code. Debugging is a difficult task, because normally, errors can only be detected indirectly by the failures they cause. Now, let us assume we have some program test that fails. How did this failure come to be?

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

Traditionally, approaches to facilitate debugging have relied on static or dynamic program analysis to detect anomalies or dependencies in the source code and thus narrow the set of potential failure causes. In this paper, we propose a novel and very different approach. Rather than focusing on the source code as failure cause, we concentrate on *program states* as they occur during program execution—especially, 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 *automated testing*, we systematically narrow these initial differences down to a small set of variables: "The failure occurs if and only if variable x has the value y (instead of y')". That is, $x = y$ is a *cause* for the failure: if x is altered to y', the failure no longer occurs. If we narrow down the relevant state differences at multiple locations in the program, we automatically obtain a *cause-effect-chain* that lists the consecutive relevant state differences—from the input to the failure.

State differences are not only causes of failures, but also *effects* of the program code. By increasing the granularity of the cause-effect chain, one can interactively isolate the moment where the program state changed from "intended" to "faulty". This moment in time is when the piece of code was executed that caused the faulty state (and thus the failure)—that is, "the error" in the program to be examined.

Our approach also differs from program analysis in that it is *purely experimental*: All that is needed is the ability to run an automated test and to access and alter program states. Knowledge or analysis of the program code is not required, although hints on dependencies and anomalies can effectively guide the experimental narrowing process and thus reduce the number of test runs.

This paper is organized as follows. Section 2 summarizes how to isolate failure-inducing circumstances automatically, using a GCC failure as example. Section 3 shows how to access program states and to isolate their difference, obtaining a cause-effect chain for the GCC failure. Section 4 shows how to narrow down failure-inducing program code—that is, "the error" in GCC. Section 5 answers why and when this approach actually works. Section 6 discusses related work and Section 7 closes with conclusion and consequences.

2. ISOLATING RELEVANT INPUT

In this Section, we recapitulate our earlier work on isolating failure-inducing input [21]. As an ongoing example, consider the *fail.c* program in Figure 1. 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
$ _
```

```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

Figure 1: The *fail.c* program that crashes GCC

#	GCC input	*test*
1	double **mult**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
2	double **mult**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
3	double **mult**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
4	double **mult**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
5	double **mult**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
6	double **mult**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
⋮	⋮	⋮
18	... *z*[*i*] = *z*[*i*] * (*z*[0] + 1.0); ...	✘
19	... *z*[*i*] = *z*[*i*] * (*z*[0] + 1.0); ...	✔
20	... *z*[*i*] = *z*[*i*] * (*z*[0] + 1.0); ...	?
21	... *z*[*i*] = *z*[*i*] * (*z*[0] + 1.0); ...	?

Table 1: Isolating failure-inducing GCC input

If we say "*fail.c* causes GCC to crash", what do we actually mean? Generally, the *cause* of any event is a preceding event without which the event in question (the *effect*) would not have occurred. Indeed, if we remove the contents of *fail.c* from the input—that is, we compile an empty file—, GCC works fine. These two experiments (the failing and the passing run) actually prove that *fail.c* is a cause of the failure.

2.1 A Divide-and-Conquer Process

In practice, we typically want a more *precise* cause than just the contents of *fail.c*—that is, a smaller difference between the failing and the passing experiment. For instance, we could try to isolate the *smallest possible difference* between two GCC inputs. This can be done using a simple divide-and-conquer method, as illustrated in Table 1.

In Step 1, we see the entire input *fail.c*, causing the GCC failure ("✘"). In Step 2, the contents of *fail.c* have been deleted (shown in grey); this empty input compiles fine ("✔"). In Step 3, we take away only the *mult* body. This input also compiles fine. Thus, we have narrowed the failure-inducing difference to the *mult* body: "Something within the *mult* body causes GCC to crash". In Steps 4 and 5, we have narrowed the cause to the *for* loop.

Continuing this divide-and-conquer method, we eventually narrow down the cause to the characters "+1.0" in Steps 18 and 19. This difference "+1.0" is minimal, as it can not be further reduced: Removing either "+" or "1.0" would result in a GCC syntax error (Steps 20 and 21)—a third outcome besides failure and success, denoted here as "?". So, we have isolated "+1.0" as a minimal difference or precise cause for the GCC failure—GCC fails if and only if "+1.0" is present in *fail.c*.

2.2 Delta Debugging

The interesting thing about the divide-and-conquer process to isolate failure-inducing input is that it can be *automated*—all one needs is a means to alter the input, and an automated test to assess the effects of the input. In fact, Table 1 does not show the narrowing process as conducted by a human, but the execution of the *Delta Debugging* algorithm, an automatic experimental method to isolate failure causes [21].

Delta Debugging requires two program runs $r_{✘}$ and $r_{✔}$—one run $r_{✘}$ where the failure occurs, and one run $r_{✔}$ where the failure does not occur. The *difference* between these two runs is denoted as δ; the difference can be *applied* to $r_{✔}$ to produce $r_{✘}$, or $\delta(r_{✔}) = r_{✘}$. Formally, δ is a failure cause—the failure occurs if and only if δ is applied. The aim of Delta Debugging, though, is to produce a cause that is as precise as possible. We thus decompose the original difference into a number of *atomic* differences $\delta = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$.

Let us illustrate these sets in our GCC example. $r_{✔}$ is the GCC run on the empty input, and $r_{✘}$ is the run on the failure-inducing input *fail.c*. We model the difference δ between $r_{✘}$ and $r_{✔}$ as a set of atomic deltas δ_i, where each δ_i inserts the i-th C token of *fail.c* into the input. We further assume the existence of a *testing function* named *test* that takes a set of atomic differences, applies the differences to $r_{✔}$, and returns the test outcome—✘ if the test fails (i.e. the expected failure occurs), ✔ if the test passes (the failure does *not* occur), and ? in case the outcome is *unresolved*—such as a non-expected failure.

Let us define $c_{✔} = \emptyset$ and $c_{✘} = \{\delta_1, \delta_2, \ldots, \delta_n\}$ as sets of atomic differences. By definition, $test(c_{✔}) = ✔$ holds (because *no* changes are applied to $r_{✔}$); $test(c_{✘}) = ✘$ holds, too (because *all* changes are applied to $r_{✔}$, changing it to $r_{✘}$).

In our GCC example, *test* constructs the input from the given changes and checks whether the failure occurs. $test(c_{✔})$ applies no changes to the empty input and runs GCC; the failure does not occur. $test(c_{✘})$ inserts all characters of *fail.c* into the empty input, effectively changing the input to *fail.c*, and runs GCC; the failure would occur. To make sure that *test* returns ✘ if and only if the original failure occurs, we make *test* return ✘ if and only if the run crashes at the same location as $r_{✘}$—that is, the program counter and the backtrace of calling functions must be identical. Otherwise, *test* returns ✔ if the program exits normally, and ? in all other cases.

Given $c_{✔}$, $c_{✘}$, and *test*, Delta Debugging now isolates two sets $c'_{✔}$ and $c'_{✘}$ with $c_{✔} \subseteq c'_{✔} \subseteq c'_{✘} \subseteq c_{✘}$, $test(c'_{✔}) = ✔$, and $test(c'_{✘}) = ✘$. Furthermore, the set difference $\Delta = c'_{✘} - c'_{✔}$ is *1-minimal*—that is, no single $\delta_i \in \Delta$ can be removed from $c'_{✘}$ to make the test pass or added to $c'_{✔}$ to make the test fail. Hence, Δ is a precise cause for the failure.

Applied to the GCC input, Delta Debugging executes exactly the tests as illustrated in Table 1. Delta Debugging first splits the input in two parts[1]. Compiling the header alone works fine (Step 3), and adding the initialization of i and j to the input (Step 4) does not yet make a difference. However, adding the "for" loop (Step 5) makes GCC crash. At this stage, $c'_{✔}$ is set up as shown in Step 4, $c'_{✘}$ is set up as in Step 5, and their difference $\Delta = c'_{✘} - c'_{✔}$ is exactly the "for" loop—in other words, the "for" loop is a more precise cause of the failure.

Resuming the narrowing process eventually leads to $c'_{✘}$ as shown in Step 18 and $c'_{✔}$ as shown in Step 19: The remaining difference is exactly the addition of "+1.0". Steps 20 and 21 verify that each of these two remaining tokens is actually relevant for the failure. So, the remaining 1-minimal difference "+1.0" is what Delta Debugging returns—after only 21 tests, or roughly 2 seconds.[2] Let

[1] In this example, we assume a "smart" splitting function that splits input at C delimiters like parentheses, braces, or semicolons.

[2] All times were measured on a LINUX PC with a 500 MHz Pentium III processor.

Let $\mathcal{C}$ be the set of all differences (in input or state) between program runs. Let $test : 2^{\mathcal{C}} \to \{\text{✘}, \text{✔}, \text{?}\}$ be a testing function that determines for a configuration $c \subseteq \mathcal{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).

Now, let $c_{\text{✔}}$ and $c_{\text{✘}}$ be 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.

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

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

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 2: The Delta Debugging algorithm in a nutshell. The function *dd* isolates the failure-inducing difference between two sets $c_{\text{✔}}$ and $c_{\text{✘}}$. For a full description of the algorithm and its properties, see [21].

us assume that an automated test already exists (for instance, as part of the GCC test suite); also, let us assume we have a simple scanner to decompose the input (a 10-minute programming assignment). Then, finding the cause in any GCC input comes at virtually no cost, compared to the manual editing and testing of *fail.c*.

The actual algorithm is summarized in Figure 2. The number of required tests grows with the number of unresolved test outcomes. In the worst case, nearly all outcomes are unresolved; then, the number of tests is $t = |c_{\text{✘}}|^2 + 3|c_{\text{✘}}|$. However, this worst case never occurs in practice, because in case of unresolved outcomes, the Delta Debugging algorithm has been designed to try runs more similar to $c_{\text{✔}}$ and $c_{\text{✘}}$. The central assumption is that the closer we are to the original runs, the lesser are the chances of unresolved test outcomes—an assumption backed by a number of case studies [21]. In the best case, we have no unresolved test outcomes; all tests are either passing or failing. Then, the number of tests t is limited by $t \leq \log_2\big(|c_{\text{✘}}|\big)$—basically, we obtain a binary search.

3. ISOLATING RELEVANT STATES[3]

Let us reconsider the isolated cause "+1.0". Although removing "+1.0" from *fail.c* makes the GCC failure disappear, this is not the way to fix the error once and for all; we rather want to fix the GCC code instead. Unfortunately, processing such arithmetic operations is scattered all over the compiler code. Nonetheless, during the compilation process, "+1.0" eventually induces a faulty GCC state which manifests itself as a failure. How does "+1.0" eventually cause the failure? And how do we get to the involved program code?

The basic idea of this paper is illustrated in Figure 3, which depicts a program execution as a series of *program states*—that is, variables and their values. On the left hand side, the program processes some input. Only a part of the input is relevant for the failure—that is, a difference like "+1.0" between an input that is failure-inducing and an input that is not. This difference in the input causes a difference in later program states, up to the difference in the final state that determines whether there is a failure or not.

[3]The process described in Section 3 is patent pending.

	δ_1	δ_2	δ_3	δ_4	
	reg_rtx_no	*cur_insn_uid*	*first_loop_store_insn*	*last_linenum*	*test*
$r_{\text{✘}}$	32	74	0x81fc4e4	15	✘
$r_{\text{✔}}$	31	70	0x81fc4a0	14	✔

Table 2: Differing variables in two GCC runs

#	*reg_rtx_no*	*cur_insn_uid*	*first_loop_store_insn*	*last_linenum*	*test*
1	32	74	0x81fc4a0	14	✔
2	32	74	0x81fc4e4	14	?
3	32	74	0x81fc4a0	15	✔

Table 3: Isolating failure-inducing variables

The problem, though, is, that even a minimized difference in the input may induce a large difference in the program state. Yet, only some of these differences are relevant for the failure. So, we *apply Delta Debugging on program states* in order to isolate the variables and values that are relevant for the failure; these isolated variables constitute the *cause-effect chain* that leads from the root cause to the failure. This is the contribution of this paper: a fully automatic means to narrow down program states and program runs to the very small fraction that is actually relevant for the given failure.

3.1 Accessing and Comparing States

Our requirements are easy to satisfy; all we need is an ordinary debugger tool that allows us to retrieve and alter variables and their values. Let us initiate two GCC runs: a run $r_{\text{✘}}$ on *fail.c* and a run $r_{\text{✔}}$ on *pass.c*, where *fail.c* and *pass.c* differ only by "+1.0". Using the debugger, we interrupt both runs at the same location L; then, we retrieve each GCC state as a set of (*variable*, *value*) pairs. As a mental experiment, let us assume that all variables have identical values in $r_{\text{✔}}$ and $r_{\text{✘}}$, except for the four variables in Table 2.

Obviously, this difference in the program state is the difference which eventually causes the failure: If we set the differing variables in $r_{\text{✔}}$ to the values found in $r_{\text{✘}}$ and resume execution, then GCC should behave as in $r_{\text{✘}}$ and fail.

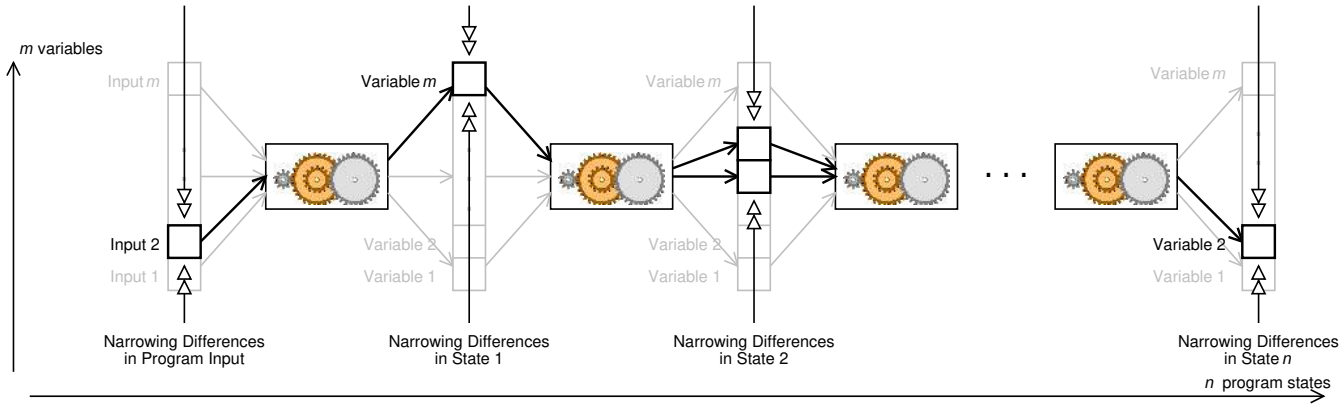

Figure 3: Narrowing a cause-effect chain. In each state, out of m variables, only few are relevant for the failure. These can be isolated by narrowing the state difference between a working run and a failing run.

We can use Delta Debugging to narrow down the cause. Now, the deltas δ_i become *differences between variable values*: applying a δ_i in $r_{✔}$ means setting the i-th differing variable in $r_{✔}$ to the value found in $r_{✘}$. The *test* function executes $r_{✔}$, interrupts execution at L, applies the given deltas, resumes execution and determines the test outcome.

As a difference example, consider δ_1 and δ_2 from Table 2. To test δ_1 and δ_2 means to execute GCC on *pass.c* (run $r_{✔}$), to interrupt it at L, to set *reg_rtx_no* to 32 and *cur_insn_uid* to 74, and to resume execution. If we actually do that, it turns out that GCC runs just fine—*test* returns ✔ and we have narrowed the failure cause by these two differences.

Unfortunately, things are not so simple. If we continue the narrowing process using Delta Debugging, we end up in trouble, as shown in Table 3. Step 1 is the application of δ_1 and δ_2, as discussed before—everything fine so far. In Step 2, though, we would apply the change δ_3, setting the pointer *first_loop_store_insn* to the address found in $r_{✘}$. This would cause an immediate core dump of the compiler—not really surprising, considering that an address from $r_{✘}$ probably has little meaning in $r_{✔}$.

In Step 3, we can exclude *last_linenum* as a cause and thus effectively isolate *first_loop_store_insn* as remaining failure-inducing difference, so we can easily see that our process is feasible. However, our state model is insufficient: We must also take *derived* variables into account—that is, all memory locations being pointed to or otherwise accessible from the base variables.

3.2 Memory Graphs

To fetch the *entire* state, we capture the state of a program as a *memory graph* [22]. 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 a memory graph example, consider Figure 4, depicting a subgraph of the GCC memory graph. The immediate descendants of the ⟨*Root*⟩ vertex are the base variables of the program. For instance, the base variable *first_loop_store_insn* can be found by following the leftmost edge from ⟨*Root*⟩. The dereferenced value is found following the edge labeled **()*[4]—a record with three members *value*, *fld[0].rtx*, and *fld[1].rtx*. The latter points to another record which is also referenced by the *link* variable.

[4]Each variable name is constructed from the incoming edge, where the placeholder () stands for the name of the parent.

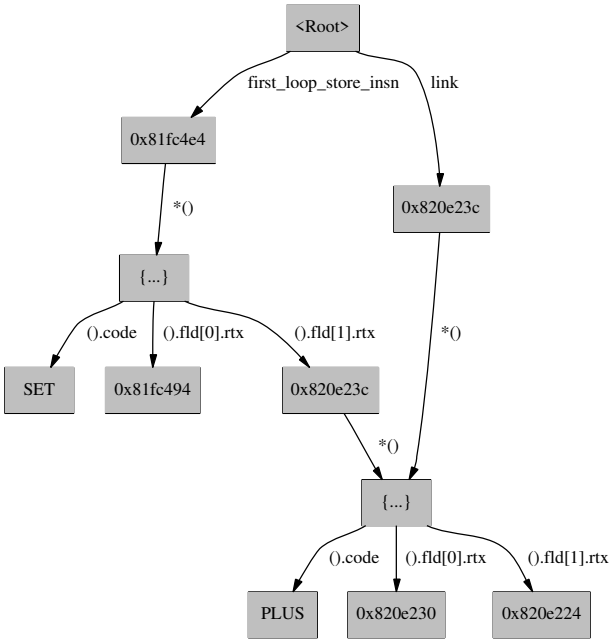

Figure 4: A simple memory graph. Pointers reference records, each referencing its members.

Memory graphs are obtained by querying the base variables of a program and by systematically unfolding all data structures encountered; if two values share the same type and address, they are merged to a single vertex. (More details on memory graphs, including formal definitions and extraction methods, are available [22].) Memory graphs give us access to the entire state of a program and thus avoid problems due to incomplete comparison of program states. They also abstract from concrete memory addresses and thus allow for comparing and altering pointer values appropriately.

However, memory graphs also indicate another problem: The set of variables itself may *differ* in the two states to be compared. As an example, consider Figure 5. In the upper left corner, you can see two memory graphs $G_{✔}$ and $G_{✘}$, obtained from the two runs $r_{✔}$ and $r_{✘}$. As a human, you can quickly see that, to change $G_{✔}$ into $G_{✘}$, one must insert element 15 into the list and delete element 20. To detect this automatically for arbitrary data structures,

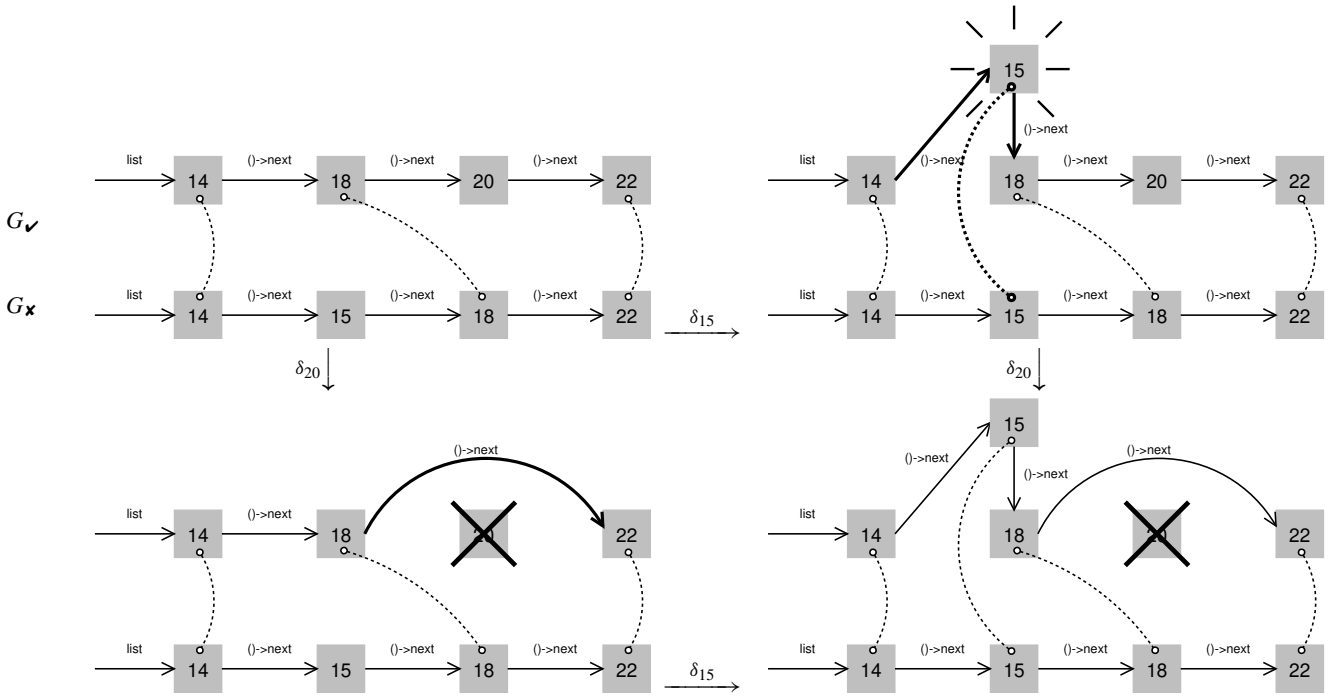

Figure 5: Determining structural differences between memory graphs. Any node not contained in the common subgraph (dotted lines) is either inserted or deleted (top left). Applying δ_{15} on $r_✔$ creates the list element 15, applying δ_{20} deletes list element 20. Applying both deltas (bottom right) transforms $r_✔$ to $(\delta_{15} \circ \delta_{20})(r_✔) = r_✘$.

one must compute a *common subgraph* of $G_✔$ and $G_✘$: Any vertex that is not in the common subgraph of $G_✔$ and $G_✘$ has either been inserted or deleted.

How does one compute a large common subgraph? We actually use two different algorithms. For small graphs, we compute the *largest* common subgraph, for larger graphs, we quickly compute a *large* subgraph.

- To compute the *largest* common subgraph, we use the approach of Barrow and Burstall [4], starting from a *correspondence graph* as computed by the algorithm of Bron and Kerbosch [5]. The correspondence graph matches corresponding vertex contents and edge labels. This is very suitable in our case, since we normally have several differing contents and labels. However, in the worst case (all contents and labels are equal), computing the largest common subgraph is an NP-complete problem.

- To compute a *large* common subgraph, we use simple *parallel traversal:* Starting from the ⟨*Root*⟩ vertex, we determine all matching edges originating from the current node and ending in a vertex with matching content. These edges and vertices become part of the common subgraph; the process is then repeated recursively. The resulting common subgraphs are not necessarily the largest, but sufficiently large for our purposes. Also, the complexity is that of a simple graph traversal.

In Figure 5, we have determined the largest common subgraph, drawn using dotted lines as a *matching* between $G_✔$ and $G_✘$.[5] It is plain to see that element 15 in $G_✘$ has no match in $G_✔$; likewise, element 20 in $G_✔$ has no match in $G_✘$.

For our purposes, we translate these differences into atomic deltas that create or delete new variables—one delta for each non-matched variable. In this example, we obtain a delta δ_{15} that creates the list element 15 and a delta δ_{20} that deletes list element 20. Both deltas can be applied independently (upper right and lower left). Altogether, we thus obtain deltas that change variable values, as sketched in Section 3.1, as well as deltas that alter data structures.

3.3 Isolating the GCC Cause-Effect Chain

Let us now put all these building blocks together. We have built a prototype called HOWCOME that relies on the GNU debugger (GDB) to extract the program state (Figure 6). Each δ_i is associated with appropriate GDB commands that alter the state.

From Section 2, we already know the failure-inducing difference in the input, namely the token sequence "+1.0", which is present

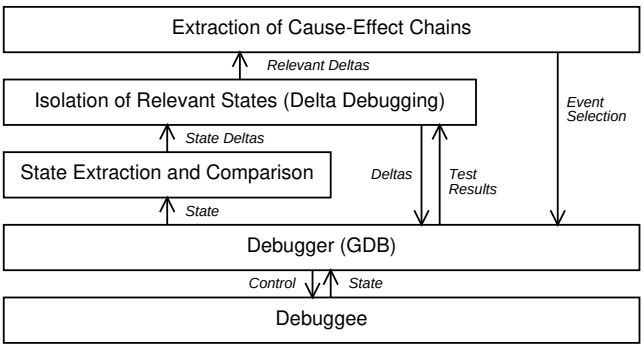

Figure 6: HOWCOME components

[5] An edge is part of the matching (= the common subgraph) if its vertices match; there is no such edge in this example.

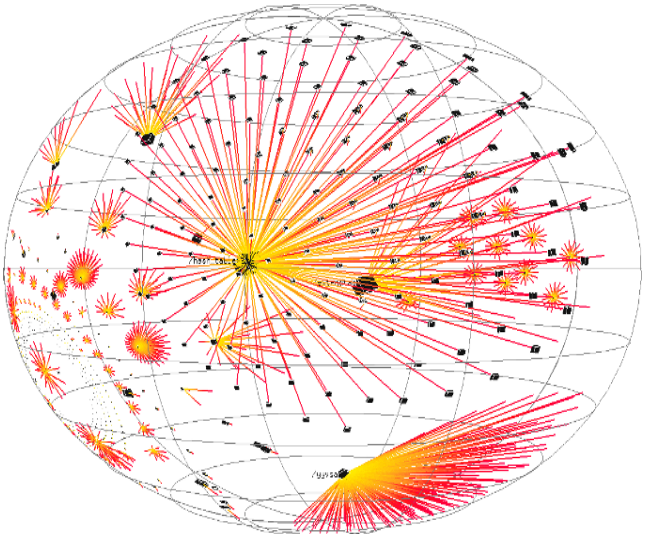

Figure 7: The GCC $G_{\checkmark}$ memory graph

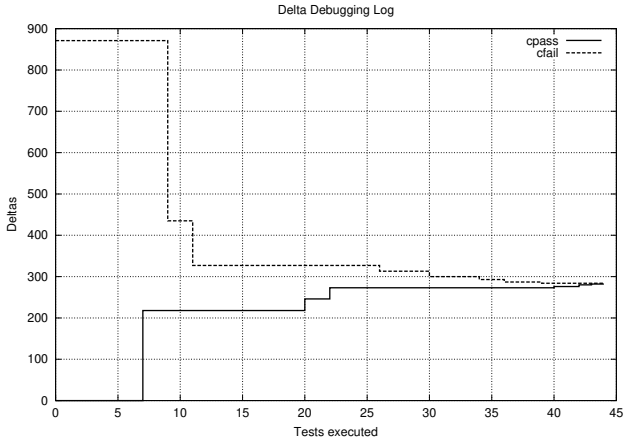

Figure 8: Narrowing at *combine_instructions*

in *fail.c*, but not in *pass.c*. HOWCOME's *test* function is also set up as discussed in Section 2.

At which locations do we compare executions? For technical reasons, we require *comparable* states—since we cannot alter the set of local variables, the current program counter and the backtrace of the two locations to be compared must be identical. From the standpoint of causality, though, any location during execution is as causal as any other.

HOWCOME thus starts with a sample of *three events,* occurring in both the passing run $r_{\checkmark}$ and the failing run $r_{\times}$:

1. After the program start (in our case, when GCC's subprocess *cc1* reaches the function *main*)
2. In the middle of the program run (when *cc1* reaches the function *combine_instructions*)
3. Shortly before the failure (when *cc1* reaches the function *if_then_else_cond* for the 95th time—a call that never returns)

3.3.1 At main

HOWCOME starts by capturing the two program states of $r_{\checkmark}$ and $r_{\times}$ in *main*. Both graphs $G_{\checkmark}$ and $G_{\times}$ have 27139 vertices and 27159 edges (Figure 7); to squeeze them through the GDB command-line bottleneck requires 15 minutes each.

After 12 seconds, HOWCOME determines that exactly one vertex is different in $G_{\checkmark}$ and $G_{\times}$—namely *argv*[2], which is `"fail.i"` in $r_{\times}$ and `"pass.i"` in $r_{\checkmark}$. These are the names of the preprocessed source files as passed to *cc1* by the GCC compiler driver. This difference is minimal, so we do not need a Delta Debugging run to narrow it further.

3.3.2 At combine_instructions

As *combine_instructions* is reached, GCC has already generated the intermediate code (called RTL for "register transfer list") which is now optimized. HOWCOME quickly captures the graphs $G_{\checkmark}$ with 42991 vertices and 44290 edges as well as $G_{\times}$ with 43147 vertices and 44460 edges. The common subgraph of $G_{\checkmark}$ and $G_{\times}$ has 42637 vertices; thus, we have 871 vertices that have been added in $G_{\times}$ or deleted in $G_{\checkmark}$.

The deltas for these 871 vertices are now subject to Delta Debugging, which begins by setting 436 GCC variables in the passing run to the values from the failing run ($G_{\times}$). This obviously is a rather insane thing to do, and GCC immediately aborts with an error message complaining about inconsistent state. Changing the other half of variables does not help either. After these two unresolved outcomes, Delta Debugging increases granularity and alters only 218 variables. After a few unsuccessful attempts (with various uncommon GCC messages), this number of altered variables is small enough to make GCC pass (Figure 8). Eventually, after only 44 tests, HOWCOME has 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 proven effect of the initial change "+1.0" from *pass.c* to *fail.c*. Each of the tests required about 20 to 27 seconds of HOWCOME time, and 1 second of GCC time.

3.3.3 At if_then_else_cond

Shortly before the failure, in *if_then_else_cond*, HOWCOME captures the graphs $G_{\checkmark}$ with 47071 vertices and 48473 edges as well as $G_{\times}$ with 47313 vertices and 48744 edges. The common subgraph of $G_{\checkmark}$ and $G_{\times}$ has 46605 vertices; 1224 vertices have been either added in $G_{\times}$ or deleted in $G_{\checkmark}$.

Again, HOWCOME runs Delta Debugging on the deltas of the 1224 differing vertices (Figure 9). As every second test fails, the difference narrows quickly. After 15 tests, HOWCOME has isolated a minimal failure-inducing difference—a single pointer adjustment, created with the GDB command

set variable link→fld[0].rtx→fld[0].rtx = link

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

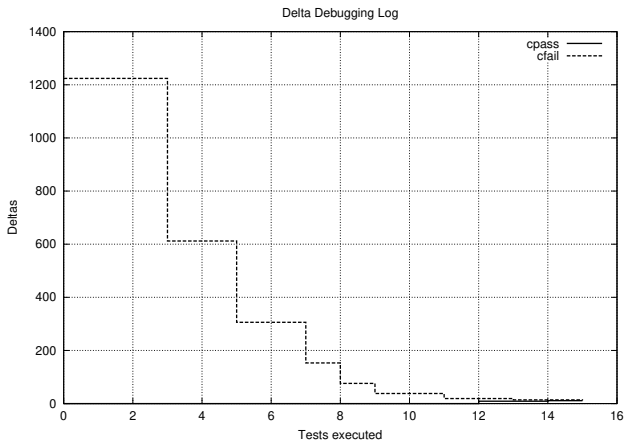

Figure 9: Narrowing at *if_then_else_cond*

3.4 The GCC Cause-Effect Chain

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

This is what happens when you invoke cc1 as "cc1 -O fail.i":

1. *Execution reaches* ***main.***
   Since the program was invoked as "cc1 -O fail.i",
   variable ***argv[2]*** *is now* ***"fail.i"****.*

2. *Execution reaches* ***combine_instructions.***
   Since argv[2] was "fail.i",
   variable ****first_loop_store_insn→fld[1].rtx→fld[1].rtx→fld[3].rtx→fld[1].rtx*** *is now* ⟨***new rtx_def***⟩*.*

3. *Execution reaches* ***if_then_else_cond (95th hit).***
   *Since *first_loop_store_insn→fld[1].rtx→fld[1].rtx→fld[3].rtx→fld[1].rtx was* ⟨*new rtx_def*⟩*,*
   variable ***link→fld[0].rtx→fld[0].rtx*** *is now* ***link****.*

4. *Execution ends.*
   Since variable link→fld[0].rtx→fld[0].rtx was link,
   the program now ***terminates with a SIGSEGV signal****.*
   The program fails.

With this summary, 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 GCC state (each taking 15–20 minutes) and 3 Delta Debugging runs (each taking 8–10 minutes) to isolate the failure-inducing differences.[6]

It should be noted again that the output above is produced in a fully automatic fashion. All the programmer has to specify is the program to be examined as well as the passing and failing invocations of the automated test. Given this information, HOWCOME then automatically produces the cause-effect chain as shown above.

4. ISOLATING THE ERROR

The ultimate aim of debugging is to break the cause-effect chain such that the failure no longer occurs. Our cause-effect chain for GCC lists some possibilities: One could prevent an input of "+1.0", avoid PLUS operators in RTL or break cycles in the RTL tree. Again, from the standpoint of causality, each of these fixes is equivalent in preventing the failure.

[6] A non-prototypical implementation could speed up state access by 1–3 magnitudes by bypassing the GDB command line.

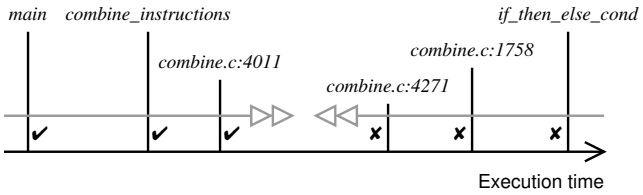

Figure 10: Narrowing down relevant events

For the programmer, though, these fixes are *not* equivalent—obviously, we need a fix that not only prevents the failure in question, but also prevents similar failures, while preserving the existing functionality. The programmer must thus choose the best place to break the cause-effect chain—a piece of code commonly referred to as "the error". Typically, this piece of code is found by determining the *transition* between an *intended* program state and a *faulty* program state. In the absence of an oracle, we must rely on the programmer to make this distinction: A cause can be determined automatically; the fault is in the eye of the beholder.

Nonetheless, cause-effect chains can be an effective help for the programmer to isolate the transition: All the programmer has to do is to decide whether the isolated state in the failing run is intended or not. In the GCC example, we assume that the states at *main* and at *combine_instructions* are intended; the RTL cycle at *if_then_else_cond* obviously is not. So, somewhere between the invocation of *combine_instructions* and *if_then_else_cond*, the state must have changed from intended ("✔") to faulty ("✘"). We focus on this interval to isolate further differences.

Figure 10 shows the narrowing process. We isolate the failure-inducing state at some point in time between *combine_instructions* and *if_then_else_cond*, namely at *combine.c* in line 1758: Here, the *newpat* variable points back to *link*—the cause for the cycle and thus a faulty state. The transition between intended and faulty state must have occurred between *combine_instructions* and line 1758.

Only two more narrowing steps are required: At line 4011, HOWCOME again isolates an additional PLUS node in the RTL tree—an intended effect of the "+1.0" input (not faulty);[7] at line 4271, HOWCOME again finds a failure-inducing RTL cycle (faulty). This isolates the transition down to lines 4013–4019. In this piece of code, executed only in the failing run, the RTL expression

$$(\texttt{MULT}\ (\texttt{PLUS}\ a\ b)\ c)$$

is transformed to

$$(\texttt{PLUS}\ (\texttt{MULT}\ a\ c_1)(\texttt{MULT}\ b\ c_2))$$

where $c = c_1 = c_2$ holds.[8] Unfortunately, c_1 and c_2 are created as *aliases* of c, which causes the cycle in the RTL tree! To fix the error, one should make c_2 a true copy of c_1—and this is how the error was fixed in GCC 2.95.3.

Do we really need the programmer to narrow down the point in time where the state becomes faulty? Not necessarily:

- First, one could simply increase the *granularity* of the cause-effect chain, and thus present more detailed information.

[7] Actually, HOWCOME reports this PLUS node as being located at *undobuf.undos→next→next→next→next→next→next→next→next→next*, which indicates that finding the most appropriate denomination for a memory location is an open research issue.

[8] This application of the distributive law allows for potential optimizations, especially for addresses.

- Second, one could attempt to isolate *cause transitions* automatically. For instance, the narrowing process as shown above could also have been guided by the fact whether the RTL tree difference PLUS is relevant or not—and would have isolated the very same location.

- Third, one could apply *heuristics* to automatically focus on events that are likely to be relevant—such as code being executed in only one of the two runs. We are currently experimenting with different *anomaly detection* methods listed in Section 6.2.

5. WHY DOES THIS WORK? AND WHEN DOES THIS WORK?

Besides GCC, we have applied HOWCOME to some more well-known programs to isolate cause-effect chains (Table 4):

- In the *sample* example from the DDD manual, Delta Debugging quickly isolated a bad *shell_sort* call.[9]

- 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 file 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.

What we found most surprising about these experiments was that one can alter program variables to more or less meaningless values and get away with it. We made the following observations, all used by Delta Debugging:

1. The altered values are not meaningless; they stem from a consistent state, and it is only a matter of statistics (e.g. which and how many variables are transferred) whether they induce an inconsistent state. The chances for consistency can be increased by grouping variables according to the program structure (which HOWCOME does not do yet).

2. The remainder of the program (and the final *test* function) acts as a *filter*: If anything happens that did not happen in the two original runs, the test outcome becomes unresolved, and the next alternative is sought. If variables have been altered and the outcome is still similar to the original two runs, then these variables are obviously irrelevant for the outcome. Precision can be arbitrarily increased by making the *test* function pickier about similarity [21].

3. In a program with a good separation of concerns, only a few variables should be responsible for a specific behavior, including failures—and this small number makes Delta Debugging efficient.

4. Program state has a *structure* and can thus easily be decomposed. In contrast, decomposing *input* as sketched in Section 2 requires the input syntax to be specified manually for each new program. And, of course, isolating relevant states is much more valuable than isolating input alone, since we can actually look at what's going on inside the program.

[9] A HOWCOME demonstration program for this example, is available online, including sample Delta Debugging source code [2].

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 4: Summary of case studies

Nonetheless, isolating cause-effect chains as presented here has its weaknesses, all to be considered:

- Delta Debugging always requires an *alternate run* in order to compare states.[10] This alternate run also determines the causes Delta Debugging can infer: A variable can be isolated as a cause only if it exists in both runs and if its value differs.

- An isolated cause may be helpful only *indirectly*. If the value reported for the failing run is not "faulty", we found an accomplice, but not yet the scoundrel: One must infer how the isolated cause interacts with the common state. In most cases, though, we expect this to be indicated by the remainder of the cause-effect chain.

- Delta Debugging as presented here isolates only one cause from *several potential causes*—for instance, *fail.c* can be changed in several ways besides removing "+1.0", and so can the induced states. Although Delta Debugging could easily be extended to search for alternative causes—which is the "best" cause, then, to present to the programmer?

- Delta Debugging may require a large number of tests to find that a *large difference* can no longer be narrowed. Such large differences will typically occur in programs where a large part of the state decides whether a test passes or fails; typical examples are numerical or cryptographic programs.

In general, weaknesses in searching algorithms can be overcome by increasing the knowledge about the search domain, and Delta Debugging is no exception. Hence, we expect weaknesses in Delta Debugging to be overcome by combining Delta Debugging with analysis methods as discussed in the next Section.

6. RELATED WORK

6.1 Program Slicing

Program slicing [17, 18] facilitates debugging by focusing on relevant program fragments. Roughly spoken, a *slice* for a statement *s* in a program consists of all other statements that could possibly influence some variable at *s* ("all statements that *s* depends upon").

[10] For an "almost correct" program, this should not be too difficult; if a program fails under all conditions, anomaly detection techniques (Section 6.2) are probably a better choice.

As a very simple slicing example, consider the code

$$\textbf{if}\ \ p\ \ \textbf{then}\ \ x' := x * y\ \ \textbf{fi}$$

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 [11]. 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 [9].

Dynamic slicing [3, 7, 13] 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 $x = 0$ holds, 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. On the other hand, cause-effect chains require several test runs (which is possibly slower than analysis), apply to a single program run only, and give no hints on the involved statements. The intertwining of program analysis and testing promises several mutual advantages.

6.2 Detecting Anomalies

Dicing [14] 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.

Running several tests at once allows one to establish *relationships* between the executed code and the test outcome. For instance, one could isolate code that was only executed in failing tests [12]. This differential approach would also have isolated the erroneous code in our GCC example.

Dynamic invariants [6] can be used to detect anomalous program behavior [8]. During execution, a tool checks the program against a model that is continuously updated; invariant violations can be immediately reported. This approach has several exciting uses; one related to our work is to check a failing run against invariants obtained from a passing run.

As discussed in Section 4, the idea that an automated process could isolate "the" erroneous code automatically in the absence of an oracle can only be based on *heuristics,* and this is what these approaches provide—including the risk of being misleading. Nonetheless, a heuristic can be very good at isolating possible causes; and it can be even more helpful when guiding a divide-and-conquer approach like Delta Debugging.

6.3 The Debugging Process

Algorithmic debugging [16] automates 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. This could easily be combined with our approach to narrow down the failure-inducing code as discussed in Section 4: "Is PLUS in the RTL tree correct (y/n)?"

6.4 Testing for Debugging

Surprisingly, there are very few applications of testing for purposes of debugging or program understanding. Our own contributions [21] as well as inferring relationships between code and tests [12] have already been mentioned.

Specifically related to our GCC case study is the isolation of failure-inducing RTL optimizations in a compiler, using simple binary search over the optimizations applied [19]. An experimental approach comparable to Delta Debugging is *change impact analysis* [15], identifying code changes that are relevant for a failure.

7. CONCLUSION AND CONSEQUENCES

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 access to 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 fixing the program is still manual (and creative) work, we expect that the time spent for debugging will be reduced significantly.

All this optimism should be taken with a grain of salt, as there is still much work to do. Our future work will concentrate on the following topics:

Optimization. As stated in Section 3.4, 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 randomly assigning variables to subsets.

Program analysis. As hinted at in Section 6, 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. Anomaly detection could help to guide the search towards specific variables or specific events.

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.

More case studies. We are currently building a *debugging server* named *AskIgor* [1] where anyone can submit failing programs via the Web (Figure 11) to have HOWCOME determine and report their cause-effect chains. As *AskIgor* requests feedback from its users, we will be able to evaluate the effectiveness and usability of our diagnoses for a large number of real-life case studies. We plan to extend *AskIgor* to accomodate and combine a variety of services for program comprehension, including program slicing and anomaly detection.

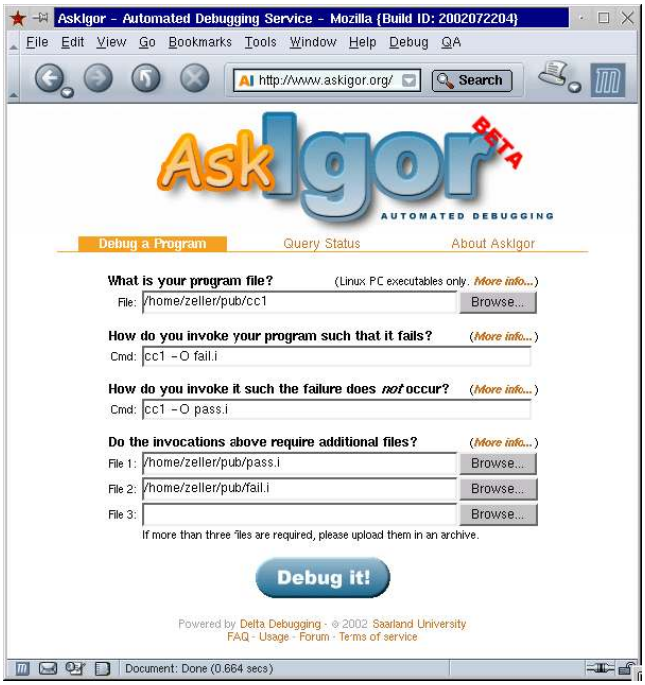

Figure 11: The *AskIgor* public debugging server

A discipline of debugging. Notions like causes and effects and approaches like running experiments under changed circumstances can easily be generalized to serve in arbitrary debugging contexts. We are currently compiling a *textbook* [20] that shows how debugging can be conducted as systematically and as all other software engineering disciplines—be it manually or automated.

Overall, we expect that debugging may become as automated as testing—not only detecting *that* a failure occurred, but also *why* it occurred. And since computers were built to relieve humans from boring, monotonous tasks—let's have them do the debugging!

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, Barbara Ryder, and Tom Zimmermann provided substantial comments on earlier versions of this paper. Many thanks go to the anonymous reviewers for their detailed and constructive comments.

More information on isolation of cause-effect chains, including a HOWCOME demonstration program, is available at the Delta Debugging web site [2].

Delta Debugging research is funded by Deutsche Forschungsgemeinschaft, grant Sn 11/8-1.

8. REFERENCES

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

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

[3] 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.

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

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

[6] 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.

[7] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proc. ESEC/FSE'99 – 7th European Software Engineering Conference / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 303–321, Toulouse, France, Sept. 1999. Springer-Verlag.

[8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In ICSE 2002 [10].

[9] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 113–123, Portland, Oregon, Aug. 2000.

[10] *Proc. International Conference on Software Engineering (ICSE)*, Orlando, Florida, May 2002.

[11] 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.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In ICSE 2002 [10].

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

[14] 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.

[15] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 46–53, Snowbird, Utah, June 2001.

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

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

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

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

[20] A. Zeller. *Automated Debugging*. Morgan Kaufmann Publishers, Aug. 2003. To appear (ISBN 1-55860-866-4).

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

[22] T. Zimmermann and A. Zeller. Visualizing memory graphs. In S. Diehl, editor, *Proc. of the International Dagstuhl Seminar on Software Visualization*, volume 2269 of *LNCS*, pages 191–204. Springer-Verlag, 2002.