CHAPTER

14 Isolating Cause–Effect Chains

This chapter presents a way of narrowing down failure causes even further. By extracting and comparing program states, delta debugging automatically isolates the *variables and values* that cause the failure, resulting in a cause–effect chain of the failure: "variable `x` was 42; therefore `p` became null; and thus the program failed."

14.1 USELESS CAUSES

In Chapter 13, we saw how to isolate inputs, code changes, or schedules that cause a given failure. In many cases, such causes directly lead to the defect in question. There are cases, though, where a difference in the input, for instance, gives few clues, if any, to the nature of the error. This is particularly true if program processes input at several places, such that it is difficult to relate a difference to some specific code.

One typical example of such programs is a *compiler*. A compiler processes the original source code through several stages until it produces an executable.

1. For C, C++, and other languages, the source code is first passed through a *preprocessor.*
2. The compiler proper parses the source code into a *syntax tree*.
3. By traversing the syntax tree, the compiler emits *assembler code*.
4. The *assembler* translates the code into *object code*.
5. The *linker* binds the objects into an *executable*.

In addition, each step can include a number of *optimizations*. The compiler, for instance, optimizes expressions found in the syntax tree, as well as the generated assembler code.

As an example, consider the `fail.c` program shown in Example 14.1. It is interesting only in one aspect: compiling `fail.c` with the GNU compiler (GCC) version 2.95.2 on Linux with optimization enabled causes the compiler to crash. In fact, depending on the version of Linux you are using it does not just crash but allocates all memory on the machine, causing other processes to die from starvation. When I tried this example first, every single process on my machine died, until only the Linux kernel and GCC remained—and only then did the kernel finally kill

EXAMPLE 14.1: The `fail.c` program that makes the GNU compiler crash

```
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];
}
```

the GCC process. (Actually, this happened while I was remotely logged in on the workstation of our system administrator, effectively terminating his session. I cannot recommend repeating the experience.)

The `fail.c` program in Example 14.1 is an input (to GCC), and thus we can isolate the actual cause—using delta debugging, for instance. Thus, we may find that if we change the line

```
z[i] = z[i] * (z[0] + 1.0);
```

to

```
z[i] = z[i] * (z[0]);
```

the program compiles just fine. Thus, we now know that the piece of code `+ 1.0` in `fail.c` causes the failure.

With this knowledge, we may now be able to *work around* the problem. If the `mult()` function in `fail.c` were part of our program, we could rearrange its code such that its semantics remained unchanged but still be capable of being compiled with GCC. To *correct* GCC, though, this knowledge is pretty much useless, even if we were compiler experts. As discussed previously, GCC processes the program code at a large number of places (in particular, optimizations), and thus there is no direct linkage from a piece of input (such as `+ 1.0`) to a piece of code that handles this input. (Contrast this to printing a `<SELECT>` tag in MOZILLA, for instance. There is one piece of code in MOZILLA that does exactly this.)

To understand what an input such as `+ 1.0` does, we must take a look into the actual computation and see what is going on. Consider a program execution as a series of states (Figure 14.1). A difference in the input (such as `+ 1.0`) causes a difference in the following states. As the computation progresses, these state differences again cause differences in later states. Thus, the differences propagate through the computation until they become observable by the user—as the difference that makes the program fail.

The difference between the two inputs is a failure cause, as are the later differences between states. Each state difference, however, is also an *effect* of earlier

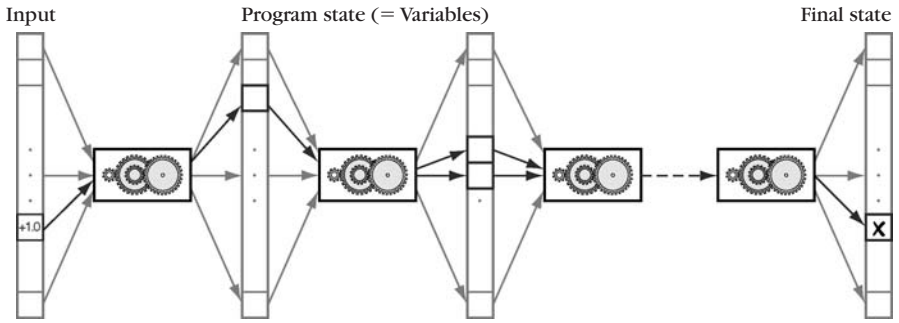

FIGURE 14.1

How differences propagate through a program run. An initial difference in the input, such as `+ 1.0`, causes further differences in the state—up to the final difference in the test outcome.

differences. Thus, the chain of differences forms a *cause–effect chain* along the computation—or, in our case, along the GCC run. If we could know what this cause–effect chain looks like, we would obtain a good understanding of how the failure came to be.

The question is: Can we actually leverage such differences by comparing program states? And how do we capture program states up front? We shall work our way through four steps and show:

1. How to capture program states as *memory graphs*.
2. How to compare program states to reveal *differences*.
3. How to narrow down these differences to reveal *causes*.
4. How to combine the causes into *cause–effect chains*.

For the sake of simplicity, we shall first study the individual steps on the well-known `sample` program (Example 1.1)—and having mastered that, we will face the mountain of complexity that is GCC. Our key question is:

HOW DO WE ISOLATE CAUSES IN PROGRAM STATE OR CODE?

14.2 CAPTURING PROGRAM STATES

To see how differences propagate along program states, we must find a way of *capturing* program states. At the lowest level, this is simple: As the program stores its state in computer memory, all one needs is a dump of that memory region. When it comes to *comparing* program states, though, we want to use the same abstraction level as when observing a program state—that is, we want to compare (and thus capture) the program state in terms of variables, values, and structures.

Chapter 8 discussed how to use a debugger to observe arbitrary parts of the program state during a program run. A debugger also allows us to list all variables of the program—that is, all global variables as well as all local variables of functions that are currently active. We call these variables *base variables*.

As an example, recall the GDB session from Section 8.3.1 in Chapter 8, where we ran the GNU debugger (GDB) on the `sample` program (Example 1.1). GDB provides three commands for listing variables:

- `info variables` lists the names of global variables.
- `info locals` shows all local variables in the current frame.
- `info args` shows all function arguments in the current frame.

If we stop at the `shell_sort()` function, for instance, we can examine all local variables.

```
(gdb) break shell_sort
Breakpoint 1 at 0x1b00: file sample.c, line 9.
(gdb) run 9 8 7
Breakpoint 1, shell_sort (a=0x8049880, size=4)
    at sample.c:9
9           int h = 1;
(gdb) info args
a = (int *) 0x8049880
size = 4
(gdb) info locals
i = 0
j = 10
h = 0
(gdb) _
```

By moving through the stack frames, we can obtain all variable values for the calling functions.

```
(gdb) frame 1
#1  0x00001d04 in main (argc=3, argv=0xbffff6fc)
    at sample.c:36
36          shell_sort(a, argc);
(gdb) info args
argc = 4
argv = (char **) 0xbffff7a4
(gdb) info locals
a = (int *) 0x8049880
i = 3
(gdb) _
```

`sample` has no global variables; otherwise, we could have obtained them via GDB's `info variables` command.

With these names and values, we can easily capture a program state as a mapping of base variables to values, as outlined in Table 14.1. For the sake of avoiding ambiguity, we suffix each (local) variable with its frame number. This way, a_0—the

Table 14.1 Base Variables of the `sample` Program

Variable	Value	Variable	Value
a_0	0x8049880	$argc_1$	4
$size_0$	4	$argv_1$	0xbffff7a4
i_0	0	a_1	0x8049880
j_0	10	i_1	3
h_0	0	—	—

Table 14.2 Derived Variables of the `sample` Program

Variable	Value	Variable	Value	Variable	Value
a_0[0]	9	a_1[0]	9	$argv_1$[0]	"sample"
a_0[1]	8	a_1[1]	8	$argv_1$[1]	"9"
a_0[2]	7	a_1[2]	7	$argv_1$[2]	"8"
—	—	—	—	$argv_1$[3]	"7"
—	—	—	—	$argv_1$[4]	0x0

argument `a` in frame 0 (`shell_sort()`)—cannot be confounded with a_1, the local variable `a` in frame 1 (`main()`).

Unfortunately, this naive approach is not enough. We must record the values of *references*, such as `a` or `argv`, and we must take into account the data structures being referenced. In other words, we must also take care of *derived variables* such as `argv[0]`, `a[1]`, and so on. One simple approach toward doing so is to *unfold* the program state—that is, follow all references until the state reaches a fix point.

1. Start with a program state consisting of all base variables and their values.
2. For each pointer in the state, include the variables it references.
3. For each array in the state, include its elements.
4. For each composite data in the state (objects, records, and so on), include its attributes.
5. Continue until the state can no longer be expanded.

Such a process can easily be automated (by instrumenting GDB, for instance) as long as we can accurately determine the types and sizes of all objects. (Section A.2.7 in the Appendix sketches how to handle such issues in C.) Doing so for the `sample` run reveals a number of *derived* variables, outlined in Table 14.2. These are obtained from following the pointers `a` and `argv` and including the elements of the arrays being pointed to.

Base and derived variables, as outlined in Tables 14.1 and 14.2, form the entire program state. Every memory location a program can (legally) access is covered.

Unfortunately, a simple name/value representation does still not suffice, because *aliasing* is not reflected. Whereas `a0` and `a1` are different variables, `a0[0]` and `a1[0]` are not. Because the pointers `a0` and `a1` have the same value, `a0[0]` and `a1[0]` refer to the same location in memory.

Visual debuggers such as DDD (see Section 8.5 in Chapter 8, on visualizing state) have addressed this problem by showing the program state not as pairs of name and values but as a *graph* in which variable values are represented by *vertices* and references by *edges*. We shall follow the same approach, but (in contrast to a debugger) capture the graph for the entire state, obtaining a so-called *memory graph*. The basic structure of a memory graph is as follows:

- *Vertices* stand for *variable values*. Each memory location that is accessible via some base or derived variable becomes a vertex.
- *Edges* stand for *references*. Each reference points to the location being referenced. Its *expression* shows how to access the location.

As an example, consider the memory graph for `sample` shown in Figure 14.2. Starting from the root vertex at the top, we can follow the individual edges to the base variables. The `size` edge, for instance, brings us to the location where the `size` value (4) is stored. The `a` variables (one for each frame) both reference the same array `[...]` referencing the three values 9, 8, and 7. Likewise, `argv` unfolds into an array of five pointers, referencing the strings `"sample"`, `"9"`, `"8"`, and `"7"`; the fifth pointer is NULL.

Some of the names attached to the references may appear rather cryptic. What does `(()[0] @ 3)` mean, for instance? The string `()` is a placeholder for the

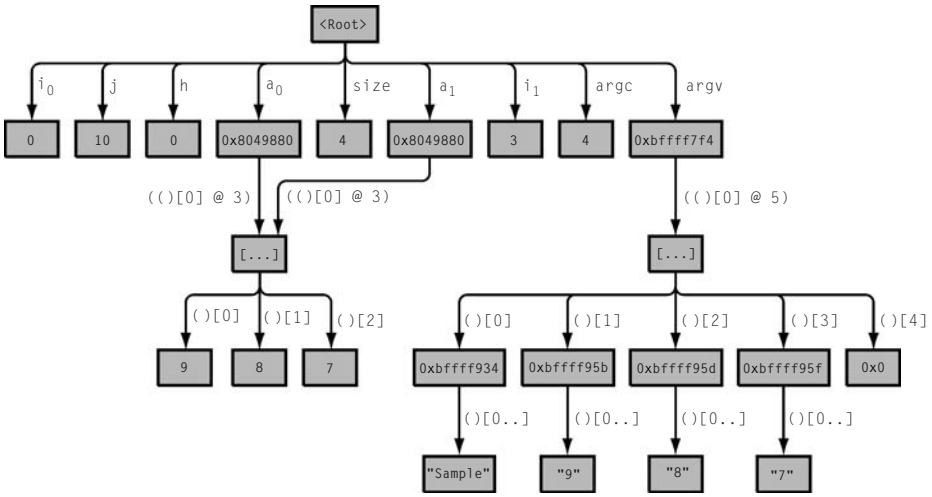

FIGURE 14.2

The state of a passing `sample` run.

expression of the *parent*—in our case, the unambiguous `a`. The `@` operator is special to GDB, where `x @ n` means "the array that consists of `x` and the `n − 1` elements following in memory." Thus, `(a[0] @ 3)` stands for the three-element array starting at `a[0]`, and this is exactly what `[...]` stands for. For a formal definition, see Section A.2.1 in the Appendix.

14.3 COMPARING PROGRAM STATES

Once we can extract program states as separate entities, there are many things we can do with them. We can observe them (as long as we can focus on a manageable subset), and we can check whether they satisfy specific properties—although assertions (see Chapter 10) would probably be better tools for this. The most important thing, though, to do with program states is *compare* them against program states observed in different runs or from different program versions.

As an example of comparing program states, consider Figure 14.3. This memory graph was obtained from a *failing* run of `sample`; namely, the run with the arguments `11` and `14`. In this visualization, we have highlighted the *differences* with respect to the passing state shown in Figure 14.2. Out of the 19 values, only 8 have a differing value. This difference in the state is caused by the difference in the input. However, this difference in the state also causes the difference in the final outcome. Thus, if we search for failure causes in the state, we can *focus on the differences* (as highlighted in Figure 14.3).

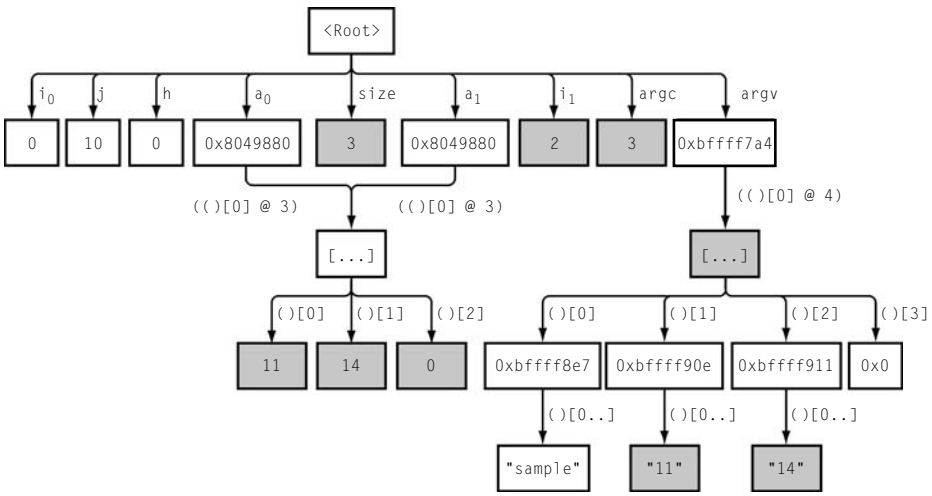

FIGURE 14.3

Differences in the state of the failing `sample` run.

How does one compute such differences? The basic idea is to compute a *matching* between both memory graphs $G_{\checkmark}$ and $G_{\times}$.

- A *vertex* in $G_{\checkmark}$ matches a vertex in $G_{\times}$ if
  - both vertices are not pointers, and have the same type, value, and size; or
  - both vertices are pointers of the same type and are NULL; or
  - both vertices are pointers of the same type and are non-NULL.

  This rule abstracts from memory locations: Regardless of where a value is stored, it can be matched. In Figure 14.3, for instance, `argv[0]`'s value differs from the one in Figure 14.2. As both are non-NULL pointers, though, they match each other.
- An *edge* in $G_{\checkmark}$ matches an edge in $G_{\times}$ if
  - the edge expressions are equal, and
  - the source and target vertices match each other.

Any vertex or edge that is not matched thus becomes a difference.

The question is now: How do we establish the best possible matching? This is an instance of the *largest common subgraph* problem, which is known to be NP-complete. The best-known algorithms have a running time that is exponential in the number of vertices. A pragmatic alternative is to compute a large common subgraph (and thus a large matching) by *parallel traversal*. Starting from the ⟨Root⟩ vertex, we determine all matching edges originating from the current vertex 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 are sufficiently large for practical purposes. The complexity is that of a simple graph traversal. (Details of the algorithm are found in Section A.2.4 in the Appendix.)

14.4 ISOLATING RELEVANT PROGRAM STATES

Focusing on the differences between two states can already be helpful for debugging—simply because the differences in the state cause the failure. As pointed out in Chapter 12, though, we normally do not search for some cause but for actual causes—that is, minimal differences between the world in which the failure occurs and the alternate world in which it does not occur. In Chapter 13 we saw how delta debugging narrows down actual causes in the program input and other circumstances. Can we apply similar techniques to automatically isolate actual causes in the program state?

In principle, we can see each program state as *input* to the remainder of the program run. Thus, we may be able to isolate failure-inducing differences in the state just as we did within the original input. What we need, though, is a difference we can

- *apply* to change the passing state into the failing state, and
- *decompose* into smaller differences to narrow down the actual cause.

EXAMPLE 14.2: GDB commands that change the `sample` state from passing to failing

```
frame 0                     # shell_sort()
set variable size = 3
frame 1                     # main()
set variable a[0] = 11
set variable a[1] = 14
set variable a[2] = 0
set variable i = 2
set variable argc = 3
set variable argv[1] = \
    (char *)strncpy((char *)malloc(3), "11", 3)
set variable argv[2] = \
    (char *)strncpy((char *)malloc(3), "14", 3)
set variable argv[3] = 0x0
```

Applying differences is not too difficult. All we need to do is translate the state differences between $G_{\checkmark}$ and $G_{\times}$ into debugger commands that alter the state. In the two `sample` graphs shown in Figures 14.2 and 14.3 there are 22 vertices and edges that are not matched. Thus, we obtain 22 differences, each adding or removing a vertex or adjusting an edge. These 22 differences translate into 10 GDB commands, shown in Example 14.2.

(Details on how to obtain these commands are listed in Section A.2.5 in the Appendix.) We can apply *all* of these GDB commands on the passing run, thus changing the state such that it is identical to the state of the failing run.

```
(gdb) break shell_sort
Breakpoint 1 at 0x1b00: file sample.c, line 9.
(gdb) run 9 8 7
Breakpoint 1, shell_sort (a=0x8049880, size=4)
    at sample.c:9
9           int h = 1;
(gdb) set variable size = 3
(gdb) frame 1
#1  0x00001d04 in main (argc=3, argv=0xbffff6fc)
    at sample.c:36
36          shell_sort(a, argc);
(gdb) set variable a[0] = 11
(gdb) set variable a[1] = 14
⋮

(gdb) set variable argv[3] = 0x0
(gdb) _
```

Because the program state determines the remainder of the execution, the remaining behavior is exactly the behavior of the failing run.

```
(gdb) continue
Continuing.
Output: 0 11

Program exited normally.
(gdb) _
```

Let's summarize. If we apply *no* differences, we get the unchanged passing run. If we apply *all* differences, we get the failing run. Consequently, one or more of the differences in the program state must form the actual cause of the failure.

To decompose differences, we could simply take the individual debugger commands and find out which of these are relevant for producing the failure. However, it is wiser to operate at a higher level of abstraction—that is, at the memory graph level. The following is the plan:

1. Take a subset of the memory graph differences.
2. Compute the appropriate debugger commands.
3. Apply them to the passing run.
4. Resume execution.
5. See whether the failure still occurs or not.

This can be easily implemented in a *test* function and then invoked from a delta debugging algorithm such as *dd*. By applying a subset of the differences, we effectively create a *mixed program state* containing parts of the passing state and parts of the failing state. After resuming execution, we assess whether a mixed state results in a passing (✓), failing (✗), or unresolved (**?**) outcome (Figure 14.4).

Eventually, delta debugging should isolate a relevant difference—at least, this is our hope, as it may well be that such mixed states always result in unresolved outcomes.

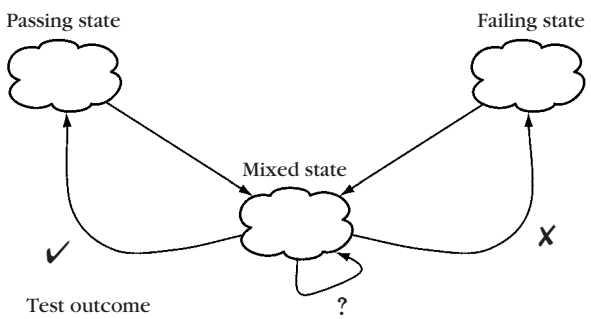

FIGURE 14.4

Narrowing down state differences. Depending on the test outcome, delta debugging uses the mixed state either as a passing or as a failing state. If the test outcome is unresolved (**?**), delta debugging creates a new mixed state.

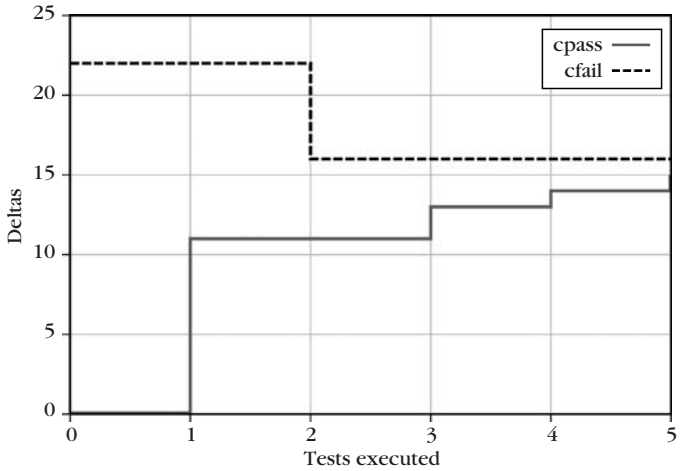

FIGURE 14.5

Isolating the failure-inducing state in `sample`. After five tests, delta debugging has narrowed down the failure cause to one single variable.

Applied to `sample`, it turns out that delta debugging performs quite well. Figure 14.5 shows what happens if we actually run *dd* on the `sample` differences.

- *Test 1:* In the first test, *dd* applies half the differences, resulting in all of `a[]`, $\texttt{i}_1$, `size`, `argc`, and `argv[]` being set to the failing state. It turns out that the failure (`0` being output) still persists, and thus the variables `j`, `h`, and $\texttt{i}_0$ are ruled out as causes.
- *Test 2: dd* only sets `a[]` and `argv[1]`. The failure occurs. Now, $\texttt{i}_1$, `size`, and `argc` are ruled out.
- *Test 3: dd* only sets `argv[1]`. The test passes, ruling out `argv[1]` as a failure cause. Only `a[]` remains.
- *Test 4: dd* sets `a[0] = 11`. The test passes, ruling out `a[0]`.
- *Test 5: dd* sets `a[0] = 11` and `a[1] = 14`. The test also passes, ruling out `a[1]`.

The only difference that remains after five tests is `a[2]`. Setting `a[2] = 0` in the passing run causes the failure in `sample`. Thus, *dd* reports `a[2]` as an actual failure cause. The failure occurs if and only if `a[2]` is zero.

Is this a good diagnosis? Yes, because it immediately helps in understanding the failure. If I sort `11` and `14`, the value of `a[2]` should not influence the outcome at all—yet it does. Therefore, this diagnosis points immediately to the defect.

On the other hand, this example raises some of the delta debugging issues discussed in Section 13.8 in Chapter 13. In particular, it shows that although delta debugging returns causes (such as `a[2]`) it need not return *infections* (such as `size`). One might wish to have *dd* isolate an infection such as the value of `size`.

However, although `size` has a different value in the two runs, and could thus be isolated as the cause, changing `size` from 3 (the value in the failing run) to 4 (the value found in the passing run) only changes the outcome if `a[2]` is also set to zero.

However, even if delta debugging "only" returns causes, these causes can again be very helpful in understanding how the failure came to be. We have built a prototype called IGOR ("Igor, go fetch bugs!") that runs the previously cited steps automatically. It determines the places to compare states, determines the differences, and runs delta debugging on the differences. IGOR can be downloaded (it is open source) and installed on your system. Originally, there also was an automated debugging server ASKIGOR that provided a public interface. Figure 1.9 shows ASKIGOR with a diagnosis for `sample`, computed as described in this section.

14.5 ISOLATING CAUSE–EFFECT CHAINS

Let's now go back to the original problem and address the GCC failure. `A + 1.0` in the input is the beginning of a long cause–effect chain that eventually leads to the failure.

Because GCC executes for a long time, the first question is: At which locations should IGOR compare executions? For technical reasons, we require *comparable* states. Because we cannot alter the set of local variables, the current program counters and the backtraces of the two locations to be compared must be identical. In addition to this constraint, though, we can choose arbitrary locations during execution. Because the causes propagate through the run, the cause–effect chain can be observed at any location.

However, for crashing programs such as GCC the backtrace of functions that were active at the moment of the crash have turned out to be a good source for locations. Example 14.3 shows the backtrace of the crash. Given a backtrace, IGOR starts with a sample of *three events* from the backtrace.

- After the program start—that is, the location at the bottom of the backtrace, when GCC's subprocess `cc1` reaches the function `main()`.
- In the middle of the program run—that is, in the middle of the backtrace, when `cc1` reaches the function `combine_instructions()`.
- Shortly before the failure—that is, the top of the backtrace, when `cc1` reaches the function `if_then_else_cond()` for the 95th time—a call that never returns.

All these events occur in both the passing run $r_{\checkmark}$ and the failing run $r_{\times}$. Let's examine these events (and associated locations) in detail.

At `main()`. We start by capturing the two program states of $r_{\checkmark}$ and $r_{\times}$ in `main()`. The graph $G_{\checkmark}$ and $G_{\times}$ has 27,139 vertices and 27,159 edges. To squeeze them through the GDB command-line bottleneck requires 15 minutes each.

EXAMPLE 14.3: The GCC backtrace

```
Frame   Address      Location
#0      0x810b19f    in if_then_else_cond( ) at combine.c:6788
  ⋮
#3189   0x810b19f    in if_then_else_cond( ) at combine.c:6788
#3190   0x8105449    in simplify_rtx( ) at combine.c:3329
#3191   0x8105029    in subst( ) at combine.c:3183
  ⋮
#3198   0x8105029    in subst( ) at combine.c:3183
#3199   0x8102457    in try_combine( ) at combine.c:1758
#3200   0x810110b    in combine_instructions( ) at combine.c:659
#3201   0x804c7fb    in rest_of_compilation( ) at toplev.c:4092
#3202   0x8183aa4    in finish_function( ) at c-decl.c:7268
#3203   0x81751ea    in yyparse( ) at c-parse.y:349
#3204   0x804b2f1    in compile_file( ) at toplev.c:3265
#3205   0x804e59a    in main( ) at toplev.c:5440
```

It takes a simple graph traversal to determine 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, and thus IGOR does not need a delta debugging run to narrow it further.

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. IGOR captures the graphs $G_{\checkmark}$ with 42,991 vertices and 44,290 edges, as well as $G_{\times}$ with 43,147 vertices and 44,460 edges. The common subgraph of $G_{\checkmark}$ and $G_{\times}$ has 42,637 vertices. Thus, we have 871 vertices that have been added in $G_{\times}$ or deleted in $G_{\checkmark}$. (The graph $G_{\times}$ is shown in Figure 1.2.)

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 ($G_{\times}$). Is there anything good that can come out of this mixed state? No. GCC immediately aborts with an error message complaining about an 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 14.6). Eventually, after only 44 tests, delta debugging 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
```

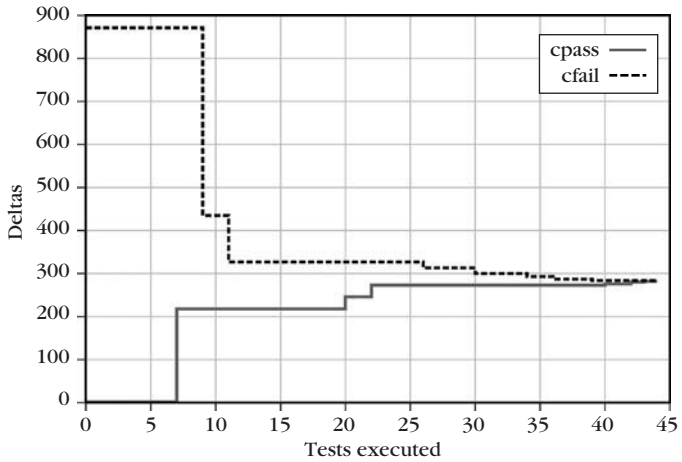

FIGURE 14.6

Narrowing at `combine_instructions()`. After 44 tests, delta debugging has narrowed down the failure cause to one single state difference—a `PLUS` operator.

```
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`.

At `if_then_else_cond()`. Shortly before the failure, in `if_then_else_cond()` IGOR captures the graphs $G_{\checkmark}$ with 47,071 vertices and 48,473 edges, as well as $G_{\times}$ with 47,313 vertices and 48,744 edges. The common subgraph of $G_{\checkmark}$ and $G_{\times}$ has 46,605 vertices; 1,224 vertices have been either added in $G_{\times}$ or deleted in $G_{\checkmark}$.

Again, delta debugging runs on the 1,224 differing vertices (Figure 14.7). As every second test fails, the difference narrows quickly. After 15 tests, delta debugging 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`.

The complete cause–effect chain for `cc1`, as reported by ASKIGOR, is shown in Figure 14.8.

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

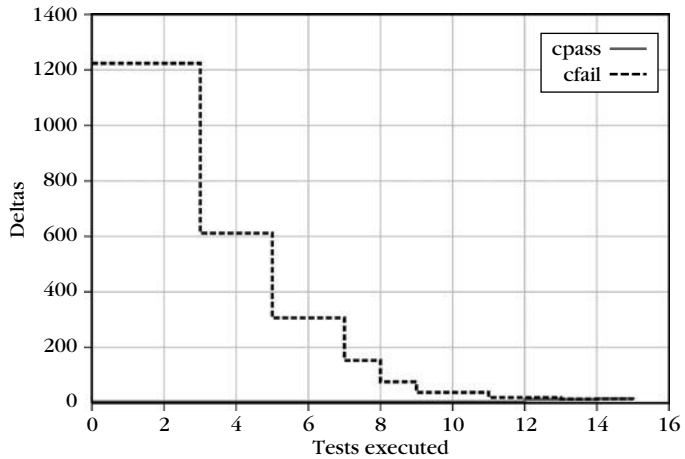

FIGURE 14.7

Narrowing at `if_then_else_cond()`. After 15 tests, delta debugging has isolated a tree cycle as the cause for the GCC crash.

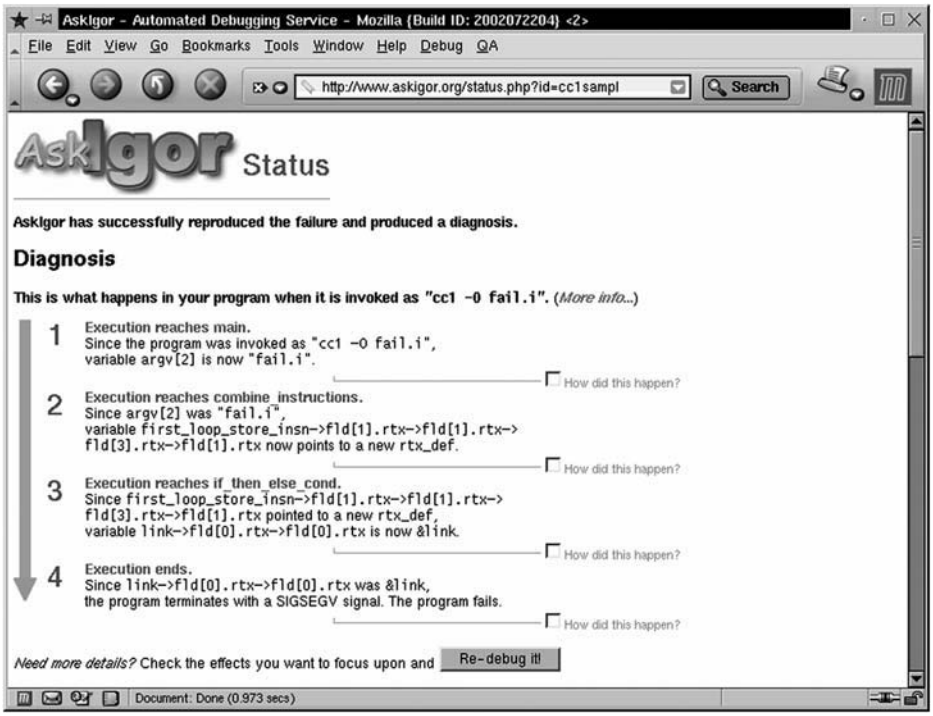

FIGURE 14.8

The GCC cause–effect chain as reported by ASKIGOR.

in the RTL tree) to the final effect (a cycle in the RTL tree). The entire diagnosis was generated automatically from the passing and the failing run. No manual interaction was required.

IGOR required six runs to extract the GCC state (each taking 15–20 minutes) and three delta debugging runs (each taking 8–10 minutes) to isolate the failure-inducing differences. (Most of this overhead is due to accessing and manipulating memory via the GDB command line. A nonprototypical implementation, built into GDB or linked with the debuggee, could speed up state access by a factor of 10 to 1,000.)

Again, it should be noted that IGOR produces this diagnosis 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, IGOR then automatically produces the cause–effect chain as shown in Figure 14.8.

14.6 ISOLATING FAILURE-INDUCING CODE

So far, we have been able to isolate causes in the program state. Ultimately, though, we are looking for causes in the program code—that is, the *defect that causes the failure*. This implies *searching in time* for the moment the defect executed and originated the infection chain.

In the GCC example, we assume that the states at `main()` and the states at `combine_instructions()` are sane. The RTL cycle at `if_then_else_cond()` obviously is not. Thus, somewhere between the `combine_instructions()` and `if_then_else_cond()` invocation the state must have changed from sane to infected. An experienced programmer would thus try to identify the moment in time where the transition from sane to infected takes place—for instance, by setting up appropriate invariant assertions, such as `assert(isAcyclicTree(root))`, in all executed functions that modify the RTL tree.

However, there is another way of coming closer to the defect—and this can also be fully automated. The idea is to search for *statements that cause the failure-inducing state*. In other words, when we find a cause in the program state, we search the code that created this very cause—in the hope that among these pieces of code we find the actual defect.

To find such causes in the code, one idea is to look at the *variables* associated with the cause in the program state. Assume there is a point where a variable *A* ceases to be a failure cause, and a variable *B* begins. (These variables are isolated using delta debugging, as described earlier.) Such a *cause transition* from *A* to *B* is an origin of *B* as a failure cause. A cause transition is thus a good place to *break* the cause–effect chain and to fix the program. Because a cause transition may be a good fix, it may also indicate the actual defect.

How do we locate such transitions? The actual algorithm *cts* is formally defined in Section A.3 in the Appendix, but it is easy to see how it works. Figure 14.9

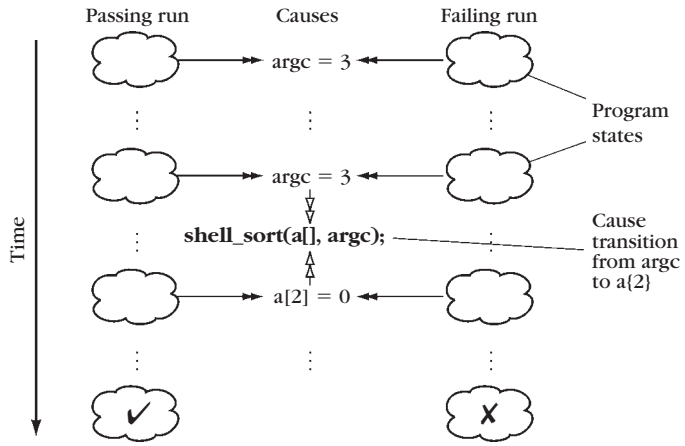

FIGURE 14.9

Locating a cause transition. Delta debugging can detect failure causes in the program state at arbitrary moments in time. When the cause shifts from one variable to another, we can use binary search in time to narrow down the transition—a statement that is likely to cause the failure.

sketches its application to the `sample` program. Before the call to `shell_sort()`, delta debugging isolates *argc* as a failure cause. Afterward, *a*[2] is the failure cause. To find the moment of that cause transition, we apply delta debugging in the middle of the interval. Then we repeat the process for the two subintervals, effectively narrowing down the transitions until we find only *direct* transitions from one moment to the next—that is, at a single statement. Overall, *cts* returns three direct cause transitions:

- From *argc* to *a*[2] in line 36
- From *a*[2] to *v* in line 18
- From *v* to *a*[0] in line 22

Each of these cause transitions is where a cause originates—that is, it points to *program code* that causes the transition and thus the failure. *a*[2] gets its value in lines 32–35, *v* gets its value in line 18, and so on. Each of these cause transitions is thus a candidate for a code correction. Moreover, each is a likely defect. Indeed, the first transition in line 36 of `sample` is exactly the location of the defect.

Let's apply this concept of cause transitions to the GCC example. Table 14.3 outlines all cause transitions occurring in GCC between the invocation and the failure. We find that the failure cause propagates through the GCC execution in four major blocks:

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

Table 14.3 Cause Transitions in GCC

No.	Location	Cause Transition to Variable
0	⟨Start⟩	`argv[3]`
1	toplev.c:4755	`name`
2	toplev.c:2909	`dump_base_name`
3	`c-lex.c:187`	`finput→_IO_buf_base`
4	c-lex.c:1213	`nextchar`
5	c-lex.c:1213	`yyssa[41]`
6	c-typeck.c:3615	`yyssa[42]`
7	c-lex.c:1213	`last_insn→fld[1].rtx →fld[1].rtx→fld[3].rtx →fld[1].rtx.code`
8	c-decl.c:1213	`sequence_result[2] →fld[0].rtvec →elem[0].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[3].rtx→fld[1].rtx.code`
9	combine.c:4271	`x→fld[0].rtx→fld[0].rtx`

- In the lexical and syntactical analysis (transitions 4–6), it is the actual difference in file content that becomes a failure cause—that is, the characters `+ 1.0`.
- The difference in file content becomes a difference in the abstract syntax tree, where `+ 1.0` induces `fld[1].rtx` to hold an additional node (`fld[1].rtx.code` is `PLUS`) in the failing run (transitions 7–8). Thus, the `+` in the input has caused a `PLUS` node, created at transition 8.
- In transition 9, the failure cause moves from the additional `PLUS` node to the cycle in the abstract syntax tree. We have

  `x→fld[0].rtx→fld[0].rtx = x,`

  meaning that the node at `*x` is its own grandchild. That is, we have again found the cycle in the RTL tree (albeit involving a different base pointer). As discussed in Section 14.5, this cycle ultimately causes an endless recursion and thus the GCC crash. However, transition 9 is where this cycle originates!

At `combine.c:4271`, the location of the last transition, we find a single statement

```
return x;
```

This line is not likely to be a defect. Let's take a look at the direct origin of `x`, in `combine.c:4013-4019`, listed in Example 14.4.

EXAMPLE 14.4: The GCC defect

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

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

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

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

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

(This application of the distributive law allows for potential optimizations, especially for addresses.) Unfortunately, in the `apply_distributive_law()` call (Example 14.4), c_1 and c_2 share a common grandchild (the macro `XEXP(x, 1)` translates into `x→fld[1].rtx`), which leads to the cycle in the abstract syntax tree. To fix the problem, one should call the function with a *copy* of the grandchild—and this is how the error was fixed in GCC 2.95.3.

At this point, one may wonder why cause transitions did not single out the call to `apply_distributive_law()` as a cause transition. The answer is simple: This piece of code is executed only during the failing run. Therefore, we have no state to compare against, and therefore cannot narrow down the cause transition any further. Line 4,271, however, has been executed in both runs, and thus we are able to isolate the failure-inducing state at this location.

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

Of course, cause transitions cannot always pinpoint a defect—simply because neither delta debugging nor the isolation of cause transitions has any notion of what is correct, right, or true. However, cause transitions are frequently also defects. In fact, cause transitions predict defect locations significantly better than any of the anomaly-based methods discussed in Chapter 11. This was found by Cleve and Zeller (2005).

Applied on the *Siemens* test suite (see Section 11.2 in Chapter 11), cause transitions narrowed down the defect location to 10 percent or less of the code in 36 percent of the test runs. In 5 percent of all runs, they even exactly pinpointed the defect. Again, these figures do not generalize to larger programs but show the potential of the concept.

14.7 ISSUES AND RISKS

Section 13.8 in Chapter 13 discussed some issues to be aware of when using delta debugging. These issues are also valid for applying delta debugging to program states. In particular,

- The alternate (passing) run should be as close as possible to the actual (failing) run.
- One may be unable to decompose large differences.
- One should take extra care to avoid *artifacts* (for instance, by comparing the backtrace).
- The actual cause reported may be one of multiple actual causes.
- The actual cause need not be an error.

In addition, applying delta debugging on program states raises its own issues, which one should be aware of.

How do we capture an accurate state? In C and C++, most of memory management is done by convention, and left to the abilities of the programmer. This can lead to ambiguous interpretations of memory content, and thus to inaccurate memory graphs. Section A.2.7 in the Appendix lists some potential pitfalls. This issue is nonexistent for languages with managed memory such as JAVA or C#, because the garbage collector must always know which objects are referenced by which other objects.

How do we ensure the cause is valid in the original runs? Each cause, as reported by delta debugging, consists of two configurations (states) $c'_{\text{✗}}$ and $c'_{\text{✓}}$ such that the difference $\Delta = c'_{\text{✗}} - c'_{\text{✓}}$ is minimal. This difference Δ between states determines whether the outcome is ✓ or ✗ and thus is an actual failure cause.

However, Δ is a failure cause only in a specific *context*—the configuration $c'_{\text{✓}}$—and this context may or may not be related to the original passing or failing runs. It is conceivable that $c'_{\text{✓}}$ may not be *feasible*—that is, there is no possible

input such that $c'_{\checkmark}$ is ever reached. It is yet unknown whether this causes problems in practice. A stronger checking for artifacts may avoid the problem.

Where does a state end? As described here, we assume that the program state is accessible via an interactive debugger. However, differences may also reside *outside* the program memory. For instance, a file descriptor may have the same value in $r_{\times}$ and $r_{\checkmark}$ but be tied to a different file. To some extent, such "greater" states can be seen as external input, such that the techniques discussed in Chapter 13 may be more appropriate.

What is my cost? Determining cause transitions is very expensive—not because the algorithms are complex but because the states are very huge and because a large number of test runs is required. Furthermore, one needs a significant infrastructure. In contrast, comparing coverage (discussed in Section 11.2 in Chapter 11) is far more lightweight, can be implemented without much risk, and requires just two test runs (which may even be conducted manually). On the other hand, it is not as precise. Obviously, you get what you pay for.

The most interesting question for the future is how to combine the individual automated debugging techniques. For instance, one could combine coverage and cause transitions and focus on cause transitions occurring in code that executes only in failing runs (see Section 11.2 in Chapter 11). One could have delta debugging focus on state that correlates with failure (see Chapter 11)—and thus effectively combine correlation, as detected from a large number of runs, with causes, as determined by additional experiments. If one has a specification of what's correct, right, or true (see Chapter 10), this could effectively guide all searches toward defects. Obviously, we have come quite far, and there is every reason to believe that computer scientists will come up with even better tools and techniques in the future.

How far can we actually go? Unfortunately, there is no chance we will ever be able to automate the entire debugging process—in particular, because there *can be no automated way of determining the defect that causes a failure*. The argument is as follows:

- By definition, the defect is where the program code deviates from what is correct, right, or true. If we know the *correct code*, though, there is no point in debugging. We can simply use the correct code instead.
- Assume that the defect is where the program state becomes infected. To determine whether the state is infected or not requires a *complete specification* of the state—at all moments during execution. Such a specification is called a correct code, and we reenter the argument as previously explained.
- Furthermore, in the absence of a correct code (or, more precisely, in the absence of a fix) we cannot tell whether a defect *causes* the failure—because we need a fix to verify causality. In fact, determining the defect that causes a failure requires generating a fix (i.e., writing the correct program).

Thus, there is no chance of an automatic device that determines the *defects*—at least not until we find a way of writing programs automatically. As long as we can isolate *causes* automatically, though, we can come very close to the defects—and close to a good explanation of how the failure came to be.

14.8 CONCEPTS

How To

To understand how a failure cause propagates through the program run, one can apply delta debugging on *program states,* isolating failure-inducing variables and values.

To capture program states, use a representation that abstracts from concrete memory locations, such as *memory graphs*.

To compare program states, compute a *large common subgraph*. Any value that is not in the subgraph becomes a difference.

To isolate failure-inducing program states, have a *test* function that

- takes a subset of the memory graph differences,
- computes the appropriate debugger commands,
- applies them to the passing run, and
- sees whether the failure still occurs or not.

Using this *test* function in a delta debugging framework will return a 1-minimal failure-inducing program state.

A failure-inducing variable, as returned by delta debugging, can be altered to make the failure no longer occur. It is thus an actual cause. This does not mean, though, that the variable is infected. It also does not mean that there may be only one failure-inducing variable.

To find the code that causes the failure, one can automatically search for *cause transitions* where a variable *A* ceases to be a failure cause and a variable *B* begins. Such cause transitions are places where the failure can be fixed, and they are likely defects.

To narrow down the defect along a cause–effect chain, search for a cause transition from a sane variable to an infected variable.

Delta debugging on states is a fairly recent technique and not yet fully evaluated.

Whereas finding *failure causes* can be fully automated, finding the *defect* that causes a failure will always remain a manual activity.

14.9 TOOLS

IGOR and ASKIGOR. You can download an open-source command-line version of IGOR at *http://www.askigor.org/*. The ASKIGOR debugging server is no longer active.

14.10 FURTHER READING

The concept of memory graphs, as described in this book, was first formulated by Zimmermann and Zeller (2002). This paper also contains more details and examples on how to capture and compare memory graphs.

The idea of isolating cause–effect chains by applying delta debugging on program states was developed by Zeller (2002). This paper is also the basis for this chapter. In this paper, the central tool was called HOWCOME, which is now a part of IGOR.

The concept of cause transitions was developed by Cleve and Zeller (2005). The paper describes the details of cause transitions in the `sample` program, in GCC, and in the *Siemens* test suite. All of these papers, as well as recent work, are available at the delta debugging home page found at *http://www.st.cs.uni-saarland.de/dd/*.

Locating a defect becomes much easier if one has a specification handy. Such a specification can be combined with systematic experiments, as discussed in this chapter. A common issue with *model checkers,* for instance, is that they can detect that a program (or, more precisely, its model as a finite automaton) does not satisfy a given specification but fail to give a precise diagnosis why that would be. To this end, Groce and Visser (2003) used multiple passing and failing runs to provide a precise diagnosis, including likely defect locations. In that these runs are generated on demand, the approach is close to delta debugging on program states. In contrast to delta debugging, though, the approach can determine defects from nondefects due to the supplied specification. In Chaki et al. (2004), the technique showed excellent localization capabilities for nontrivial programs.

To actually compute the *largest* common subgraph instead of simply some large subgraph, one can use the approach of Barrow and Burstall (1976), starting from a *correspondence graph* as computed by the algorithm of Bron and Kerbosch (1973). The correspondence graph matches corresponding vertex content and edge labels. This is very suitable in our case, in that we normally have several differing content and labels. However, in the worst case (all content and labels are equal) computing the largest common subgraph has exponential complexity.

Compilers such as GCC have frequently been the subject of automated debugging techniques. Whalley (1994) describes how to isolate failure-inducing RTL optimizations in a compiler, using a simple binary search over the optimizations applied.

EXERCISES

14.1 Once again, consider the `bigbang` program (Example 8.3). If you change the `mode` variable in line 7, the failure no longer occurs.

(a) Sketch how the difference in `mode` propagates through the execution and how it prohibits the failure.

(b) Sketch the cause transitions in `bigbang`.

(c) Would these cause transitions help in locating the defect? If so, why? If not, why not?

14.2 Download the IGOR command-line tool from *http://www.askigor.org/*. Use IGOR to obtain a diagnosis for the `sample` program. If you alter the arguments, how does the diagnosis change? Why?

14.3 Give three examples of cause transitions that are defects, and three examples of cause transitions that are not defects.

> So assess them to find out their plans,
> both the successful ones and the failures.
> Incite them to action in order to find out
> the patterns of movement and rest.
>
> – SUN TZU
> *The Art of War* (c. 400 BC)