CHAPTER

9 Tracking Origins

Once we have observed an infection during debugging, we need to discover its origin. In this chapter, we discuss *omniscient debugging*, a technique that records an entire execution history such that the user can explore arbitrary moments in time without ever restarting the program. Furthermore, we explore *dynamic slicing,* a technique that tracks the origins of specific values.

9.1 REASONING BACKWARD

A common issue with observation tools, as discussed in Chapter 8, Observing Facts, is that they execute the program *forward* in time, whereas the programmer must reason *backward* in time. Applied to interactive debugging tools, this means that the programmer must carefully approach the moment in time where the infection is observable. As soon as the infection is found, he or she must restart the program and stop at some earlier moment in time to explore the previous state.

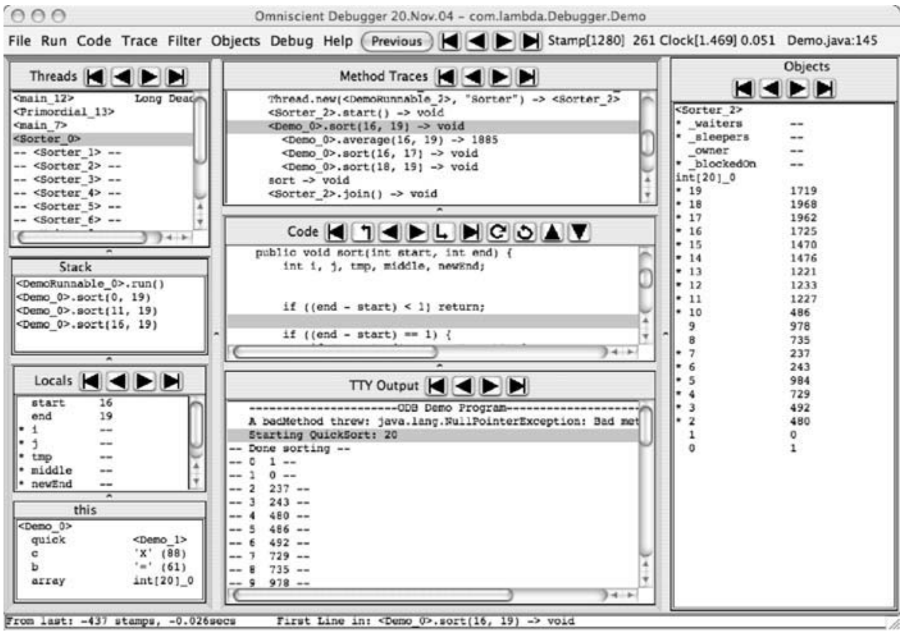
Restoring the steps to get to the earlier state can be pretty time consuming—and if we go just one step too far, the program must be restarted anew. One key issue for (human) debuggers is thus how to ease the task of discovering the *origin* of a value, and how to keep a *memory* of what was going on during the run. Here, we ask:

WHERE DOES THIS VALUE COME FROM?

9.2 EXPLORING EXECUTION HISTORY

So you want to support the programmer in examining the history? One first idea would be to have a means of undoing the last execution steps. An even better idea is to *explore the entire execution history backward.* In other words, we *record* the execution—every single change to every single aspect of the state.

This radical idea has been realized under the name of *omniscient debugging* in a number of recent debuggers. Rather than accessing the program while it is running,


Omniscient Debugger 20.Nov.04 – com.lambda.Debugger.Demo
File Run Code Trace Filter Objects Debug Help Previous
Threads
Method Traces
Code
Stack
Locals
this
TTY Output
Objects

```
Stamp[1280] 261 Clock[1.469] 0.051 Demo.java:145
```

```
<main_12>              Long Dead
<Primordial_13>
<main_7>
<Sorter_0>
-- <Sorter_1> --
-- <Sorter_2> --
-- <Sorter_3> --
-- <Sorter_4> --
-- <Sorter_5> --
-- <Sorter_6> --
```

```
    Thread.new(<DemoRunnable_2>, "Sorter") -> <Sorter_2>
    <Sorter_2>.start() -> void
    <Demo_0>.sort(16, 19) -> void
      <Demo_0>.average(16, 19) -> 1885
      <Demo_0>.sort(16, 17) -> void
      <Demo_0>.sort(18, 19) -> void
    sort -> void
    <Sorter_2>.join() -> void
```

```
  public void sort(int start, int end) {
      int i, j, tmp, middle, newEnd;


      if ((end - start) < 1) return;

      if ((end - start) == 1) {
```

```
<DemoRunnable_0>.run()
<Demo_0>.sort(0, 19)
<Demo_0>.sort(11, 19)
<Demo_0>.sort(16, 19)
```

```
  start     16
  end       19
• i         --
• j         --
• tmp       --
• middle    --
• newEnd    --
```

```
<Demo_0>
  quick          <Demo_1>
  c              'X' (88)
  b              '=' (61)
  array          int[20]_0
```

```
   ---------------------ODB Demo Program-------------------
   A badMethod threw: java.lang.NullPointerException: Bad met
   Starting QuickSort: 20
-- Done sorting --
-- 0  1 --
-- 1  0 --
-- 2  237 --
-- 3  243 --
-- 4  480 --
-- 5  486 --
-- 6  492 --
-- 7  729 --
-- 8  735 --
-- 9  978 --
```

```
<Sorter_2>
• _waiters      --
• _sleepers     --
  _owner        --
• _blockedOn    --
int[20]_0
• 19            1719
• 18            1968
• 17            1962
• 16            1725
• 15            1470
• 14            1476
• 13            1221
• 12            1233
• 11            1227
• 10            486
  9             978
  8             735
• 7             237
• 6             243
• 5             984
• 4             729
• 3             492
• 2             480
  1             0
  0             1
```

```
From last: -437 stamps, -0.026secs     First Line in: <Demo_0>.sort(16, 19) -> void
```

FIGURE 9.1

Exploring execution history in ODB. Users can navigate backward and forward through the execution, and along events related to variables, statements, and outputs.

an omniscient debugger first executes the program and records it. Once the run is complete, the omniscient debugger loads the recording and makes it available for observation.

Figure 9.1 shows the ODB debugger for JAVA, a prototype that pioneered and proved the concept. At the center of ODB, one can see the traditional source code window, highlighting the current execution position. The other windows show more of the program state, such as current threads, the stack content, variable values, the console output, and so on.

Using the arrow buttons above the source code window, one can step through the program's execution. In contrast to ordinary debuggers, though, ODB also features buttons to step *backward* through the execution. As one walks back in time, all windows are synchronized properly. If you step before an assignment, the variable value shown will go back to the earlier value, and if you step before an output statement, the appropriate output will disappear from the console.

In addition to stepping backward and forward along program lines, one can step back and forth between *time stamps* specific to the individual windows. In the code window, each moment of line execution has a time stamp (nothing special here). In the data window, though, time stamps correspond to *value changes* of the displayed variables. To find out when `array[]` was last changed, you can select `array[]` in the window listing the variable values, and then use the buttons above that window to walk back and forth in the value history. Variables with a value that has changed from the previously shown time stamp are highlighted with a leading *.

Each time an earlier point in the execution history is reached, the other windows synchronize as well. Therefore, one sees the previous value and the line that assigned the value, as well as every other aspect of the current state. In addition to stepping through events, the programmer can issue *event queries* (as discussed in Section 8.4.2 in Chapter 8) and thus directly jump to specific events. (In contrast to COCA, though, ODB does not need to renew the execution, which makes querying much faster.) A typical debugging session using ODB proceeds as follows:

1. Start with the failure, as reported in the console window. Clicking on the output will lead to the moment in time the output was produced—that is, the output statement will be shown in the code window.
2. Step backward, searching the infection at each point.
3. Follow the infection back to the defect.

The main drawback, of course, is that recording all state changes is expensive. First, recording adds a certain overhead, such that a typical program will be slowed down by a factor of 10 or more. Second, recording needs memory—a lot of it. ODB generates data at the rate of 100 MB per second, meaning that a 2-GB address space will fill up in 20 seconds. To deal with the memory issue, one can

- record specific events only (such as the last second before the failure, or whatever fits into memory),
- record specific parts of the system only (we do not care what is going on in the runtime library), or
- use a compressed storage of the individual events.

All in all, though, the advantages far outweigh the disadvantages. Using omniscient debugging, the programmer has random access to every moment in time and every aspect of the execution, without ever needing to restart the program—a tremendous advantage over ordinary interactive debuggers. It is not unlikely that omniscient debugging will become a standard feature of future interactive debuggers.

9.3 DYNAMIC SLICING

Although omniscient debugging is handy for accessing every aspect of a program execution, the programmer still has to figure out how the individual values came to be. This is where *dependences* come in handy, as discussed in Chapter 7. If we know that the bad value of variable *A* can only come from variable *B* at some earlier moment in time, we can immediately focus on *B*. A good programmer would thus use both observation (of the program run) and deduction (from the program code) to quickly progress toward the defect.

Chapter 7 treated dependences in an abstract way—that is, we explored dependences as they hold for arbitrary runs of the program. When debugging, we have a concrete failing run at hand, and we would like to know the *dependences for this concrete run* in order to trace back origins.

This is where the technique of *dynamic slicing* comes in handy. Like a static slice, a dynamic slice encompasses a part of the program—that is, the part of the program that could have influenced (or could be influenced by) a specific variable at some point. However, a dynamic slice does not hold for *all* possible runs but for one single *concrete* run.

As an example of a static versus a dynamic slice, consider the program shown in Example 9.1. The static backward slice (a) of `s`, being output in line 15, encompasses the entire program (try it!). The dynamic slice in (b) applies to the run in which `n` and `a` are read in as `2` and `0`, respectively. Note that a large number of statements has no effect on the final value of `s`. The dynamic slice for the run is more precise than the static slice for the entire program.

How can a slice as shown in Example 9.1 be computed? To compute a dynamic slice, one requires a *trace*—a list of statements in the order they were executed during the concrete run. Such a trace is either created by *instrumenting* the program—that is, having the compiler or another tool embed special tracing commands—or by running the program in an interpreter. The leftmost column of Example 9.2 shows the trace of statements from the run `n = 2, a = 0` in Example 9.1(b).

In this trace, one records the variables that were read and written—just as in Table 7.1, except that now the effects are recorded for each statement as it is executed. In addition, one introduces a *predicate pseudovariable* for each predicate that controls execution (such as `p8` for the predicate `i <= n` in line 8). Each of

EXAMPLE 9.1: Static and dynamic slices

```
 1 n = read();
 2 a = read();
 3 x = 1;
 4 b = a + x;
 5 a = a + 1;
 6 i = 1;
 7 s = 0;
 8 while (i <= n) {
 9     if (b > 0)
10         if (a > 1)
11             x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
```

(a) Static slice for `s`

```
 1 n = read(); // 2
 2 a = read(); // 0
 3 x = 1;
 4 b = a + x;
 5 a = a + 1;
 6 i = 1;
 7 s = 0;
 8 while (i <= n) {
 9     if (b > 0)
10         if (a > 1)
11             x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
```

(b) Dynamic slice for `s`

EXAMPLE 9.2: Computing a dynamic slice from a trace

Trace	Write	Read	Dynamic Slice
1 n = read();	n		
2 a = read();	a		
3 x = 1;	x		
4 b = a + x;	b	a, x	**2, 3**
5 a = a + 1;	a	a	**2**
6 i = 1;	i		
7 s = 0;	s		
8 while (i <= n) {	p8	i, n	**6, 1**
9 if (b > 0)	p9	b, p8	**2, 3, 6, 1, 4, 8**
10 if (a > 1)	p10	a, p9	**2, 3, 6, 1, 4, 8, 5, 9**
12 s = s + x;	s	s, x, p8	**6, 1, 7, 3, 8**
13 i = i + 1;	i	i, p8	**6, 1, 8**
8 while (i <= n) {	p8	i, n	**6, 1, 8, 13**
9 if (b > 0)	p9	b, p8	2, 3, 6, 1, 4, 8, **13**
10 if (a > 1)	p10	a, p9	2, 3, 6, 1, 4, 8, 5, 9, **13**
12 s = s + x;	s	s, x, p8	6, 1, 7, 3, 8, **13, 12**
13 i = i + 1;	i	i, p8	6, 1, 8, **13**
8 while (i <= n) {	p8	i, n	6, 1, 8, 13
15 write(s);	o15	s	**6, 1, 7, 3, 8, 13, 12**

these pseudovariables is "written" by the statement that controls execution and "read" by the statements that are controlled. Example 9.2 shows the effects of the individual statements.

From these effects, one can now compute dynamic slices by following the read/write dependences. The following is a method that computes all dynamic slices for all written values at once.

1. For each write w to a variable, assign an empty dynamic slice.

$$\text{DynSlice}(w) = \varnothing$$

2. Proceed forward through the trace (or execute the program, generating the trace). Whenever a value w is written, consider all variables r_i read in that statement. For each r_i, consider the line `line`(r_i) where r_i was last written, as well as its dynamic slice `DynSlice`(r_i). Compute the union of these lines and slices and assign it to the write of w.

$$\text{DynSlice}(w) = \bigcup_i \big(\text{DynSlice}(r_i) \cup \{\text{line}(r_i)\}\big)$$

As an example, consider the dynamic slice of line 4, `DynSlice`(4). In line 4, `b = a + x`, variable b is written and variables a and x are read, last written in lines 2 and 3, respectively. Therefore, the dynamic slice of b in line 4 is the union of

- the dynamic slice of a in line 2 (empty),
- the dynamic slice of x in line 3 (empty), and
- lines 2 and 3.

Formally, this reads:

$$\begin{aligned}\mathrm{DynSlice}(4) &= \mathrm{DynSlice}(2) \cup \{2\} \cup \mathrm{DynSlice}(3) \cup \{3\}\\ &= \varnothing \cup \{2\} \cup \varnothing \cup \{3\}\\ &= \{2, 3\}\end{aligned}$$

3. At the end of the execution, all definitions will be assigned a slice that holds all origins of all values.

As an example, consider the right column in Example 9.2, showing the dynamic slices as they are computed along the trace. (Values in bold indicate new additions to the slices.) The last line shows the dynamic backward slice for `s` in the statement `write(s)`. These are exactly the lines highlighted in Example 9.2.

On average, dynamic slices are far more precise than static slices. In a concrete run, all locations of all variables—including those in *computed* expressions such as `a[i]` or `*p`—are known, eliminating conservative approximation. Likewise, in a concrete run paths that were not taken need not be taken into account.

All in all, this makes dynamic slices smaller than static slices. Whereas a static backward slice typically encompasses 30 percent of a program's statements, a dynamic slice only encompasses 5 percent of the executed statements (note that the *executed* statements also form a subset of all statements). The increased precision comes at a price, though:

- *Overhead*. Dynamic slices depend on a trace of the program, which is difficult to obtain efficiently. Although we need not record the entire value history (as in omniscient debugging), we still need to record which statements were taken in which order.
- *Lack of generality*. Dynamic slices only apply to a single run of the program, and thus cannot be reused for other runs (in contrast to program-dependence graphs and static slices, which are valid for all runs).

9.4 LEVERAGING ORIGINS

How can dynamic slices be used in a debugger? This was explored by Ko and Myers (2004) in the WHYLINE system. WHYLINE stands for *W*orkspace that *H*elps *Y*ou *L*ink *I*nstructions to *N*umbers and *E*vents." It is a debugger whereby programmers can ask questions about why things happened, and why other things did not happen. In short, it is a debugger whereby you can ask: "Why did my program fail?"

The WHYLINE originally was designed for the ALICE language—a simple language in which three-dimensional objects can be defined and manipulated. ALICE is designed for novices learning programming. In the programming environment, users select and compose ALICE statements interactively rather than entering them as text. Nonetheless, ALICE is just as powerful as any other programming language.

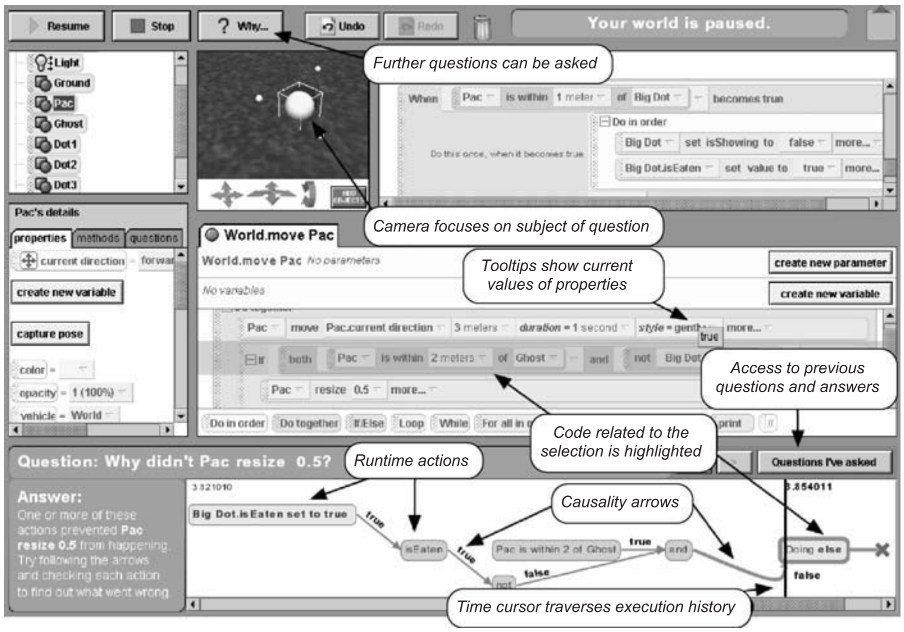
In Figure 9.2, we see a screenshot from a student's debugging session. In a PACMAN program, the protagonist Pac has collided with a ghost, but does not shrink as it should. The student uses the WHYLINE for debugging the program. In the center window, we see the code that should resize Pac by 0.5.

```
if both Pac is within 2 meters of Ghost and
    not Big Dot.isEaten:
    Pac resize 0.5
```

However, this resizing code is not being executed. To find out why the branch is not taken, the student has clicked on the Why button. From a menu (Figure 9.3), she has opted to ask why `Pac resize 0.5` was not executed.

The WHYLINE 's answer is shown at the bottom of Figure 9.2. The diagnosis comes as a chain of dependences ending at the `else` branch (), consisting of the following events:

1. `Big Dot.isEaten` is set to true.
2. Therefore, the `isEaten` variable is true.
3. The negation `not` (from the previous code) is false.
4. Although `Pac is within 2 meters of Ghost` is true.
5. The `and` conjunction (from the previous code) evaluates to false.
6. Therefore, the `else` clause is taken.



When Pac is within 1 meter of Big Dot becomes true
Do this once, when it becomes true
Do in order
Big Dot set isShowing to false more...
Big Dot.isEaten set value to true more...

World.move Pac No parameters
No variables
Pac move Pac.current direction 3 meters duration = 1 second style = gently more...
If both Pac is within 2 meters of Ghost and not Big Dot
Pac resize 0.5 more...

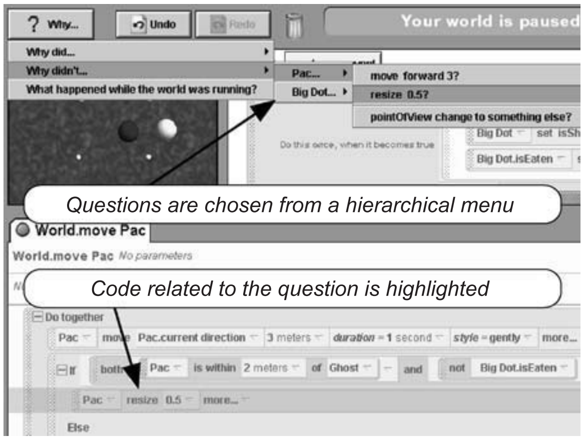
Question: Why didn't Pac resize 0.5?

Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.

3.321010
Big Dot.isEaten set to true
isEaten
not
Pac is within 2 of Ghost
and
Doing else
3.854011

FIGURE 9.2

Asking "Why didn't…?" questions in the WHYLINE. *Source*: Ko and Myers 2004.


? Why...
Undo
Redo
Your world is paused
Why did...
Why didn't...
What happened while the world was running?
Pac...
Big Dot...
move forward 3?
resize 0.5?
pointOfView change to something else?
Questions are chosen from a hierarchical menu
World.move Pac
World.move Pac No parameters
Code related to the question is highlighted

```
Do together
  Pac  move  Pac.current direction  3 meters  duration = 1 second  style = gently  more...
  If  both  Pac  is within  2 meters  of  Ghost  and  not  Big Dot.isEaten
    Pac  resize  0.5  more...
  Else
```

FIGURE 9.3

Selecting a question in the WHYLINE. At the bottom, the diagnosis shows why a specific piece of code was not executed. *Source*: Ko and Myers, 2004.

The student can further explore this diagnosis by scrubbing a time cursor over the dependency chain and thus access arbitrary moments in time, just as in omniscient debugging. When she moves the cursor over the `Big Dot.isEaten set to true` bubble, the code window shows the code in which the `Big Dot.isEaten` variable is set.

All of this is just correct behavior. It turns out that Pac did not resize simply because he had eaten a big dot before, making him immune against ghost attacks. (Rather than Pac being resized, one should see the ghost being eaten!)

How does the WHYLINE compute its diagnosis? The answer is simple: All the WHYLINE does is compute a *dynamic backward slice* of the queried property. More precisely, the following are the strategies that WHYLINE uses:

"Why did...?" questions. For a "*Why did...?*" question, the WHYLINE shows the *dynamic backward slice* from the queried statement *S*. That is, it would show all statements *P* is dependent on. The slice is limited to two such statements. If needed, the programmer can query again about a specific statement.

As an example, consider the slices shown in Example 9.1 and the question "Why did `s = 2` in line 15?" The WHYLINE strategy would point to

- "`s = 1`" from line 12 (`s = s + x`), the direct origin of `s`, as well as
- "`i = 2`" from line 8 (`i <= n`), which controls execution of line 12.

(Instead of line 8, the WHYLINE strategy could also point to line 3 (`x = 1`), the origin of `x`.) If needed, the programmer can further explore the origins by querying why one of these statements was executed.

"Why didn't...?" questions. For a "*Why didn't...?*" question, the WHYLINE would use the *static backward slice* of the statement *S*, and by following back the

control dependences, retrieve those statements that directly prevented execution. It then performs the "*Why did…?*" question on each.

As an example, let's ask the question "Why didn't `x = 2` in line 11?" Following the static dependences, we have three control dependences from line 11 to lines 8, 9, and 10, respectively. Assume the WHYLINE asks the "*Why did…?*" question only for the closest one, line 10 (`if (a > 1)`). As an answer, it would then present

- "`a = 1`" from line 5 (`a = a + 1`), as well as
- "`b = 1`" from line 9 (`if (b > 0)`), the statement directly controlling line 5.

Again, the programmer could interactively query the WHYLINE about how these values came to be.

Overall, the WHYLINE demonstrates what can be done in a modern debugger. It incorporates random access in time (as in omniscient debugging) as well as static and dynamic slicing to trace origins. By limiting the slice length, it prevents the programmer from having to deal with too many possible origins, and allows exploration along the dependences. Finally, it shows how to package it all in a nice user interface, avoiding the use of program analysis jargon such as "dependences" or "slices." A study conducted by Ko and Myers (2004) showed that the WHYLINE could decrease debugging time by nearly a factor of 8, highlighting the potential of modern debugging environments.

As of 2008, the WHYLINE is also available for JAVA programs, targeting expert developers, and achieving all of the original WHYLINE contributions "without limitations on the target program, other than that it uses standard I/O mechanisms and that the program does not run too long" (Ko and Myers, 2008). The WHYLINE for JAVA allows programmers to ask specific questions on the output of the program: "Why is this line blue?" or "Why did this error message show up?" In a user study, the WHYLINE participants were more than twice as fast as the experts without the WHYLINE—showing off the potential of program-analysis techniques when coming in an easy-to-use and intuitive package.

9.5 TRACKING DOWN INFECTIONS

Even with all of the advanced observation tools discussed in this chapter, we still need a strategy for using them systematically. The following is a *general strategy* that combines observation and dependences to narrow down arbitrary infection sites—that is, a strategy for *locating arbitrary defects*.

1. Start with the infected value as reported by the failure. In the `sample` program (Example 1.1), this would be `a[0]`.
2. Follow back the dependences to potential origins. This can be done using
   - static dependences, as discussed in Chapter 7, or
   - dynamic dependences, as discussed in Section 9.3.

Following the data dependences in `sample`, we can trace back the value of `a[0]` to other values in `a[]` as well as to `size`.

3. Observe the origins and judge whether the individual origins are infected or not. In the `sample` run, we find that `size` is infected (it has the wrong value).
4. If you find an earlier infected value, repeat steps 2 and 3 to track its origins. In `sample`, the value of `size` depends on `argc` (and only on `argc`).
5. When you find an infected value *V* where all values that *V* depends on are sane, you have found the infection site—in other words, the defect. `argc` is sane, but `size` is not. Thus, the infection must have taken place at the assignment of `argc` to `size`—at the function call to `shell_sort()`.
6. Fix the defect, and verify that the failure no longer occurs. This ensures that you have found the defect that caused the failure in question.

This strategy is in fact an application of scientific method (see Chapter 6) that creates hypotheses along the dependences, and uses observation to assess possible origins. It also guarantees that you *will* find the infection site, just by observation and judgment. It even works when dependences are imprecise or even unknown. In such cases, there is more to observe and more to judge, of course.

However, the amount of data to be observed and to be judged can still be enormous. We must still ease the task. In particular, we can:

- Help the programmer *judge* whether some origin is infected. This can be done using *assertion* techniques, discussed in Chapter 10.
- Help the programmer *focus* on specific origins. Such origins include *anomalies* (discussed in Chapter 11) and actual *failure causes* (discussed in Chapters 13 and 14).

Stay tuned—there is more to come.

9.6 CONCEPTS

How To

To explore execution history, use an *omniscient debugger*, which records the entire execution of a program and grants random access to every aspect of the execution.

To isolate value origins for a specific run, use *dynamic slicing.*

Dynamic slices apply only to a single run of the program, but are far more precise than static slices.

The best available interactive debuggers leverage omniscient debugging, static slicing, and dynamic slicing to provide diagnoses about why things happen and why they do not.

To track down an infection, follow back the dependences and observe the origins, repeating the process for infected origins (see Section 7.3 in Chapter 7).

9.7 TOOLS

Dynamic Slicers. Very few dynamic slicers are publicly available. Abhik Roychoudhury at the University of Singapore and his team have published JSLICE, a dynamic slicer for JAVA programs: *http://jslice.sourceforge.net/*. Clemens Hammacher at Saarland University has implemented JAVASLICER, a dynamic slicing framework for JAVA. JAVASLICER specifically focuses on usability, separation of tracing and slicing, and accuracy: *http://www.st.cs.uni-saarland.de/javaslicer/*.

ODB. The ODB debugger was developed by Lewis (2003), who also coined the term *omniscient debugging*. ODB is available at: *http://www.lambdacs.com/debugger/debugger.html*. Commercial implementations of omniscient debugging include Visicomp's RETROVUE and Omnicore's CODEGUIDE.

WHYLINE. The WHYLINE for JAVA is available for download via Andrew Ko's home page at *http://faculty.washington.edu/ajko/whyline-java.shtml*.

9.8 FURTHER READING

Dynamic slicing was invented independently by Korel and Laski (1990) and by Agrawal and Horgan (1990). The computation method in this chapter follows Gyimóthy et al. (1999). Still, the main challenge of dynamic slicing is *efficiency*. Zhang and Gupta (2004) offer several interesting approaches to the subject, as well as an extensive overview on the literature.

An interesting variant of dynamic slicing is *critical slicing,* as realized by DeMillo et al. (1996) in the SPYDER debugger. Critical slicing is based on the idea of removing individual statements from the program and to leave only those statements relevant to the error. As DeMillo et al. (1996) point out, this reduces the average program by 64 percent.

The original WHYLINE version was presented by Ko and Myers (2004); the JAVA version is discussed in Ko and Myers (2008). The paper by Ko and Myers (2009) focuses on the design rationales and gives substantial insights on how to make a debugger user friendly for real developers.

EXERCISES

9.1 In the exercises of Chapter 5, we used a JAVA implementation of the delta debugging algorithm (Example 5.10) to simplify inputs. Download and use ODB to debug the run.

9.2 Compute the dynamic slices for the run `sample 11 14` for the `sample` program (Example 1.1).

9.3 Using the WHYLINE strategies, answer the following questions for the run `sample 11 14`:

(a) Why is `a[0] = 0` at line 38?

(b) Why is line 22 not executed in the first iteration of the `for` loop in line 16?

9.4 The following program is supposed to determine the greatest common divisor.

```
01   int gcd(int a, int b)
02   {
03       int d = b;                [S] [D1] [D2] [Δ]
04       int r = a % b;            [S] [D1] [D2] [Δ]
05
06       while(r > 0)              [S] [D1] [D2] [Δ]
07       {
08           int n = d;            [S] [D1] [D2] [Δ]
09           d = r;                [S] [D1] [D2] [Δ]
10           r = n / d;            [S] [D1] [D2] [Δ]
11       }
12
13       return d;                 [S] [D1] [D2] [Δ]
14   }
```

This program has a defect: `gcd(2, 6)` returns `3`, but `3` is not a divisor of `2`. The defect is in line 10: it should read `r = n % d`.

(a) Determine the static backward slice of `d` (in line 13). Check $\boxed{\text{S}}$ for all statements that are part of the slice.

(b) Determine the dynamic backward slice of `d` in the run `gcd(2, 6)`. Check $\boxed{\text{D1}}$ for all statements that are part of the slice.

(c) Determine the dynamic backward slice of `d` in the run `gcd(0, 5)`. Check $\boxed{\text{D2}}$ for all statements that are part of the slice.

(d) Determine the difference between the slices in steps 2 and 3. Check $\boxed{\Delta}$ for all statements that are part of the difference.

(e) "A difference as obtained in step 4 always contains the faulty statement." Give a program and two slices that contradict this claim.

Life can only be understood backwards, but it must be lived forwards.

–SØREN AABYE KIERKEGAARD
Diary (1843)