CHAPTER

Reproducing Problems

4

The first step in debugging is to *reproduce* the problem in question—that is, to create a test case that causes the program to fail in the specified way. The first reason is to bring it under control, such that it can be observed. The second reason is to verify the success of the fix. This chapter discusses typical strategies for reproducing the operating environment, the history, and the problem symptoms.

4.1 THE FIRST TASK IN DEBUGGING

Once a problem report is in the problem database, or once a test has failed, the problem will eventually be processed by some programmer in order to fix it. The programmer's first task (or, more precisely, the first task of any debugging activity) is to *reproduce the problem*—that is, the problem must occur in the very same way as stated in the original problem report. Reproducing the problem is important for two reasons:

1. *To observe the problem.* If you are not able to reproduce the problem, you cannot observe what is going on. This thoroughly limits your ability to reason about the program, as pointed out in Chapter 6. Basically, your only chance is to *deduce* from the program code what *might* have happened to cause the problem.
2. *To check whether the problem is fixed.* You may be able to deduce a potential problem cause—and even to design a change that would fix this potential cause. But how do you show that your change actually fixes the problem in question? You can do so only by reproducing the initial scenario with the changed product, and showing that the problem now no longer occurs. Without reproducing the problem, you can never know that the problem has been fixed.

Of course, if the problem occurred in a test in the local environment we are already set and done, because we can reproduce the test at the touch of a button—assuming the test was automated and deterministic. In other cases, and if the problem was reported by a user, we must first *create a test case* that reproduces the problem. This is the key issue in this chapter:

HOW CAN A TEST REPRODUCE A SPECIFIC PROBLEM?

4.2 REPRODUCING THE PROBLEM ENVIRONMENT

Whereas creating test cases per se is well understood (see Chapter 3), reproducing a specific problem can be one of the toughest problems in debugging. The process consists of two main steps:

1. Reproducing the *problem environment*—that is, the setting in which the problem occurs.
2. Reproducing the *problem history*—the steps necessary to create the problem.

We first focus on reproducing the problem environment. If a problem occurs in a specific environment, the best place to study the problem is exactly this environment. Thus, if Olaf has trouble with the Perfect Publishing Program we should simply knock on Olaf's door and ask him whether we could take a brief look at his machine—or in a more tech-savvy environment ask him for permission to log on to his machine.

Working in the problem environment offers the best chance of reproducing the problem. However, for numerous reasons working in the problem environment typically does not happen:

- *Privacy:* The most important reason is *privacy*—users may simply not wish others to operate their machines; the same goes for large corporations.
- *Ease of development:* To examine the problem, programmers may require a complete development environment, involving diagnostic software such as debuggers, which are not typically found on customer's machines.
- *Cost of maintenance:* Users may depend on their machines being operational. In many cases, you cannot simply take a machine out of production to maintain the software.
- *Travel cost:* When physical access is required, having maintainers move to the user's environment is expensive.
- *Risk of experiments:* Debugging typically involves experiments, and running experiments on a user's machine may cause damage.

Thus, unless the problem environment is already prepared for diagnostic actions your *local environment* involves the least cost in reproducing the problem. For these reasons, as a maintainer you typically attempt to reproduce the problem using as much of the local environment as possible. This is an iterative process, as follows:

1. Attempt to reproduce the problem in *your environment*, using the product release as specified in the problem report (see Section 2.2 in Chapter 2). If the problem occurs, you are done—and you are lucky.

   However, do not cry "success" simply because you experience *a* problem—cry "success" only if you experience *the* problem. Here, "the" problem means the problem exactly as specified in the problem report. Every deviation from the specified symptoms increases the risk of you working on a problem that is different from the user's problem. Thus, be sure to check every single symptom that is specified. If it does not occur in your environment, you may want to refine your efforts.

2. If the problem does not occur yet, adopt more and more circumstances from the problem environment—one after the other. This applies to configuration files, drivers, hardware, or anything else that might possibly influence the execution of the product. Start with those circumstances

   - that are the most likely to cause problems (as inferred from previous problem reports), and
   - that are easy to change (and to be undone).

   For instance, if the problem environment includes a specific set of user preferences first try using this set of preferences. If, however, the problem environment uses LemonyOS 1.0, but you use LemonyOS 1.1, you may want to downgrade your machine to LemonyOS 1.0 only after adopting all less-expensive aspects—or better yet have quality assurance keep a LemonyOS 1.0 machine for testing.

3. Adopt one circumstance after the other until

   - you could reproduce the problem, or
   - your environment is identical to the problem environment (as far as specified by the user).

   In the latter case, there are two alternatives:

   - The first alternative is that because the problem does not occur in your environment it *cannot* have occurred in the problem environment. Consider the fact that the problem report is incomplete or wrong.
   - The second alternative is that the problem report is accurate but there still *is* a difference between your environment and the problem environment—because otherwise the problem would occur in your environment as well. Try to find that difference by querying further facts about the problem environment.

   In both cases, it is wise to query further facts about the problem environment. As illustrated in Bug Story 4, even otherwise insignificant details can have an impact on whether one can reproduce earlier runs.

This process of getting nearer and nearer to the problem environment has a beneficial side effect: you also isolate some circumstances that are relevant in producing the problem. Let's say the problem environment includes a Helquist graphics card. Your environment does not yet include a Helquist graphics card and the problem does not occur. However, as soon as you add a Helquist graphics card the problem *does* occur. This means that the Helquist graphics card is relevant for reproducing the problem; more precisely, the Helquist graphics card is a *problem cause*. Chapter 13 discusses systematic ways of isolating such causes—manually and automatically.

BUG STORY 4

Mad Laptop

In 2002, I went to a conference to give a laptop demonstration of a complex software system. Before I left, we had taken every precaution that the software would run perfectly. But the evening before the demonstration, I sat in my hotel room and nothing worked. The software would run into time-out errors anytime, everywhere. I phoned my department at home. They had the same software on the same machine and everything worked.

After three hours of trials, I decided to start from scratch. To my amazement, the demo now ran just fine. What had happened? Within these three hours, the battery had run out of power and I had connected my laptop to a power plug. I disconnected the laptop, repeated the demo, and the problem occurred again. It turned out that my laptop ran slower when running on batteries, which saved energy but also introduced the time-out errors. Needless to say, all of our previous tests had been conducted on AC power, and this was how I gave the demo.

4.3 REPRODUCING PROGRAM EXECUTION

Finding an environment in which the problem occurs is not enough. We must also be able to recreate the individual steps that lead to the problem. Let's assume that we have recreated the problem environment, as discussed in Section 4.2. Let's also assume that the problem history, as stated in the problem report (see Section 2.2 in Chapter 2), is accurate and complete. Still, we may not be able to reproduce the problem. In fact, even the user on the original machine may not be able to reproduce the problem.

Why is it that for the same program a problem may or may not occur? The execution of a program is determined by its code (which we assume constant) and its *input.* If the input differs, so does the execution of the program. To reproduce a specific execution, we must thus *reproduce the program input.*

To reproduce an input, the programmer must *observe* it and *control* it. Only when the input is under control does the execution become *deterministic*—that is, the outcome of the execution does not change when repeated. Without such control, the execution becomes *nondeterministic*—that is, the problem may occur or not, regardless of the will of the programmer.

All of this sounds rather trivial when thinking of the program input in a classical sense—that is, data read from a file or a keyboard. Unfortunately, the input of a program can be more than that—in particular, if you take the view that the input comprises anything that influences the execution. The following are possible inputs, as sketched in Figure 4.1:

- *Data:* As stored in files and databases is the least problematic input, as it can easily be reproduced.

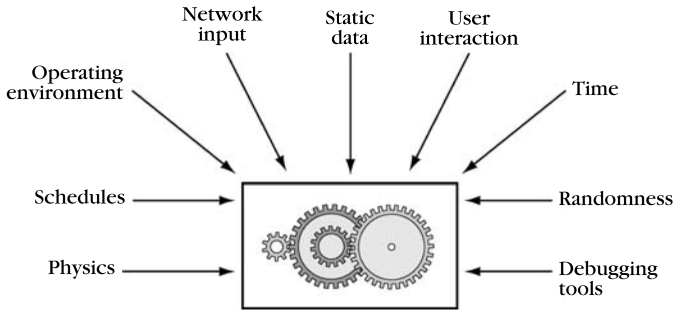

FIGURE 4.1

Input that might influence a program's execution.

- *User inputs:* Can differ in minor details, which may be relevant for a problem.
- *Communications:* Between threads, processes, or machines, offer several challenges for reproduction.
- *Time:* Can influence the program execution in various ways.
- *Random numbers:* By definition make every program execution different.
- *Operating environments:* Provide services beyond those listed previously that can heavily interact with the program, all of which may or may not influence the execution.
- *Process and thread schedules:* Normally should not interfere with the program's execution. However, if the program has a defect they may.

Most of these inputs have *intended* effects on the program (which is why they should be reproduced). Other inputs, though, have unintended effects, such as the following inputs:

- *Physics:* Is typically abstracted away at the hardware layer, but cosmic rays, electrical discharges, or quantum effects can influence a program's execution.
- *Debugging tools:* Typically interfere with the program execution, and thus may uncover but also mask problems.

The general pattern for controlling these inputs is to set up a *control layer* between the real input and the input as perceived by the program, as sketched in Figure 4.2. This control layer *isolates* the program under observation from its environment. The program input becomes controllable, and thus the execution becomes deterministic. Any of the automated techniques discussed in Chapter 3 can be used for actually controlling the program.

In the remainder of this chapter, we will focus on applying this pattern to make a run deterministic, organized by input source. In Section 4.2, we have already discussed how to reproduce the environment. Start with your own environment and reproduce one input source after the other until the problem is

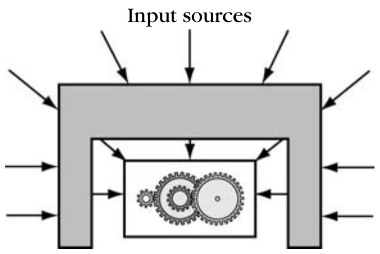

FIGURE 4.2

Controlling a program's input. To control program input, one sets up a control layer between the real input and the input as perceived by the program.

reproduced. In the process, you will narrow down the input sources relevant to the problem.

4.3.1 Reproducing Data

Regarding reproduction, data as stored in files and/or databases are seldom an issue. Files and databases can easily be transferred from one machine to another, and can be easily replicated. The following are the only three issues to be aware of:

1. *Get all the data you need*. Be sure to reproduce all data your application accesses and all data under the user's control. This also includes *configuration data* such as registries or configuration files.

   Most of the data required to reproduce the problem is typically already included in the problem report. As discussed in Section 2.2 in Chapter 2, on reporting problems, it is helpful to set up a specific tool or functionality to collect such data.
2. *Get only the data you need*. Some programs operate on enormous amounts of data, which makes it difficult to examine the problem. Chapter 5 discusses how to simplify input data.
3. *Take care of privacy issues*. Data under the user's control may contain private or confidential information. Be sure not to find entire hard disks with sensitive information in your mailbox (this has happened!).

4.3.2 Reproducing User Interaction

In the good old days, programs would read their input from data files only, making reproduction an easy task. Modern programs, though, use complex user interfaces, and these make it difficult to observe and control the user's input.

The standard approach to reproducing user interaction is to use a *capture/replay* tool. Such a tool comes in two modes (Figure 4.3):

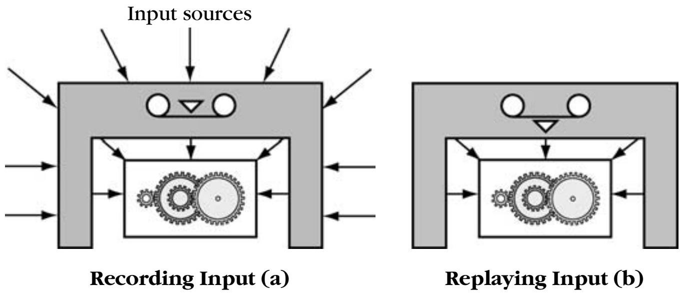

FIGURE 4.3

Capturing and replaying program input. During a normal execution, the controlling layer records the interaction (a). Later, it replays it (b).

1. *Capturing interaction.* The program executes normally, interacting with its environment. However, the tool *records* all input from the environment to a *script* before forwarding it to the program.
2. *Replaying interaction.* The program executes under control of the tool. The tool redirects the program input such that it no longer gets its input from the environment but rather from the previously recorded script. Obviously, this makes the input controllable—and thus allows for reproducible runs.

Technically, a tool realizes capture/replay by intercepting calls to library or system functions providing input. Capture takes place after the execution of the input function. The tool records the returned input and passes it on to the program. During replay, the tool no longer executes the input function. Instead, it returns the previously recorded input.

What does the recorded information look like? In Chapter 3 we saw examples of scripts that automate program interaction. Depending on the layer at which input capturing occurs, the scripts simulate user input either

- as a "low-level" stream of events (Section 3.3.1), or
- as a "higher-level" sequence of interactions with user controls (Section 3.3.3).

Capturing user interaction can also take place at the following two layers:

1. As a stream of events, a captured user interaction looks like the one shown in Example 3.4, except that there would be no comments and the script would include real-time aspects (such as waiting for, say, 376 milliseconds until releasing a key).
2. As a sequence of interactions with user controls, a captured user interaction looks like that shown in Example 3.6, except that (again) there would be no comments.

As discussed in Section 3.3, on testing at the presentation level, a script at the "low level" is *fragile*. That is, the slightest change in user interface layout, font size, or even interaction speed will make them unusable. Therefore, a low-level script should not be used beyond a single short-term debugging session.

To make the test reusable, one should at least aim to automate input at the higher level—or test at the functionality layer, as described in Section 3.4. Alas, few tools are available that allow programmers to capture and replay interaction at these layers. For a single testing or debugging session, though, it may already suffice to capture and replay just the basic user interaction in terms of key presses and mouse clicks.

4.3.3 Reproducing Communications

The techniques used for capturing and replaying user interaction (Section 4.3.2) can be extended to any type of communication. For instance, specialized tools can capture, inspect, reconstruct, and replay network traffic.

A general problem with such tools is that they may alter the performance of your program and its communications, especially if there are much data to be captured and replayed. (This is not the case with user interactions, which are typically low bandwidth.) The fact that tools alter the performance is not so much a problem in itself, but this change in the environment may mask problems that would occur otherwise (see Section 4.3.9 for the effects of debugging tools).

Note that one does not necessarily have to capture the entire communication since the start of the program. If the program goes into a reproducible state while

BUG STORY 5

Press Play on Tape

As a student, I worked in a computer store. This was in the mid-1980s, and we had HP calculators, Commodore PETs, and Ataris. One day, a customer walked in and asked for help. He was not able to enter a program on his Commodore 64 home computer. Whenever he entered the first line, the computer would issue a syntax error message. We asked him to show us the problem on a C-64 nearby. He entered the first line of the program and got the message:

```
10 print "Hello World"
?syntax  error
ready.
```

We were amazed, and tried to reproduce the problem. However, when one of us would enter the same line it would work properly. Only if the customer entered the line did the error occur. Finally, one of us asked the customer to enter just the number 10. He did so, and got:

```
10
press play on tape
```

Now we understood. We asked the customer, "How do you type ones and zeros?" He replied, "I use a lowercase l and a capital letter O, as on my old typewriter." The customer had just entered lowercase l and capital O instead of ten, and the C-64 interpreted this as an abbreviation for the `LOAD` command. "Oh, you mean I should use the digit keys instead?" "Yes," we said, and off went another happy customer.

operating, it suffices to capture (and replay) only the communication since that reproducible state. Databases, for instance, reach such a reproducible state after each transaction.

Failure causes are likelier to be found in the later communications than in earlier communications. Thus, it may frequently suffice to capture just the latest communication (say, the database transactions that are not yet completed). This saves space, does not hamper performance too much, and still may be helpful for reproducing the problem.

4.3.4 Reproducing Time

Many programs require the current time of day for executing correctly. Problems that depend on the time of day are difficult to reproduce and difficult to trace (see Bug Story 6, for example).

If the program is supposed to depend on the time of day, or if there is some indication that the problem depends on the time of day, one must be able to execute the program under an arbitrary time of day. One way to do so is to change the system time before each execution—but this is cumbersome and error prone. A much more comfortable option, though, is to make time a *configurable item*: the program is set up such that it can be executed with an arbitrary time of day. This is useful for reproducing problems and very helpful for creating automatic tests (Chapter 3).

As in Section 4.3.2, on reproducing user input, the basic idea here is to obtain *control over input sources* such that they can be reproduced at will. The time of day is just a specific instance of how to obtain such control.

4.3.5 Reproducing Randomness

Specific programs, notably games and cryptographic applications, depend on *randomness*. That is, they are supposed to behave differently in every single execution. Here, nondeterminism is part of the design. That is, the program is set up in such a way as to prohibit reproduction of individual runs.

BUG STORY 6

Program Only Works on Wednesday

I once had a program that worked properly only on Wednesdays. The documentation claimed the day of the week was returned in a double word (8 bytes). In fact, Wednesday is nine characters long, and the system routine actually expected 12 bytes of space for the day of the week. Because I was supplying only 8 bytes, it was writing 4 bytes on top of the storage area intended for another purpose. As it turned out, that space was where a *y* was supposed to be stored for comparison with the user's answer. Six days a week the system would wipe out the *y* with blanks, but on Wednesdays a *y* would be stored in its correct place.

Source: Eisenstadt (1997).

When testing such applications, such randomness must also be controlled and made reproducible. The most efficient way to do so depends on the source of randomness. If randomness is obtained from a series of pseudorandom numbers, it may suffice to capture (and replay) the initial *seed* of the random number generator. Many games allow one to explicitly specify a seed such that individual executions can be reproduced.

Cryptographic applications typically depend on more than just a pseudorandom number generator. They obtain randomness from several sources (such as user input, network events, thermal noise, or others). These sources must be captured and replayed like input, as discussed in Section 4.3.2. If needed, organize your program such that developers can replace the source of randomness by a deterministic source. (Be cautious about enabling end users to turn randomness off, though—especially in cryptographic applications!)

4.3.6 Reproducing Operating Environments

User interaction, communications, time, and randomness all have one thing in common: a *program interacts with its environment*, using services provided by this environment. Such an operating environment typically consists of further libraries, maybe a virtual machine, an operating system, and eventually the entire world to which the particular machine may be connected.

The entire interaction between a program and its environment is typically handled by the *operating system*. More precisely, the operating system controls all inputs and outputs of a program. Thus, the boundary between program and operating system comes as a natural place at which to monitor, control, and possibly record and replay program executions.

As an example, consider the simple C++ program shown in Example 4.1. When executed, it reads in a password from standard inputs and outputs "access granted" if the correct password is entered. (A real application would at least care not to echo the input.)

What does the interaction between this program and its environment look like? On a Linux box, the STRACE tool monitors the interaction of a program with the operating system by printing out all system calls, their arguments, and their return values. (Similar tools are available on all UNIX-like systems.) After compiling `password.C`, we run STRACE on the resulting `password` binary, diverting the STRACE output into a file named `LOG`.

```
$ c++ -o password password.C
$ strace ./password 2> LOG
Please enter your password: secret
Access granted.
$ _
```

What does the STRACE output look like? Example 4.2 shows an excerpt from the `LOG` file. The `LOG` file lists the *system calls*—function invocations by which

EXAMPLE 4.1: `password.C`—a simple C++ password requester

```
#include <string>
#include <iostream>

using namespace std;

string secret_password = "secret";

int main()
{
    string given_password;

    cout « "Please enter your password: ";
    cin » given_password;
    if (given_password == secret_password)
        cout « "Access granted." « endl;
    else
        cout « "Access denied." « endl;
}
```

EXAMPLE 4.2: The STRACE log from `password.C` (excerpt)

```
⟨Clutter produced by shared libraries ...⟩
write(1, "Please enter your password: ", 28) = 28
read(0, "secret\n", 1024)                 = 7
write(1, "Access granted.\n", 16)         = 16
exit_group(0)                             = ?
```

the program requests services from the operating system. The `write()` system call, for instance, writes a string to stream number 1, the standard output stream on Linux (and other POSIX environments). STRACE also logs the *data* returned by the system calls. For instance, it reports the return value of `write()` (the number of written characters). For the following `read()` call, it reports the actual characters (`"secret\n"`) read. Obviously, a tool such as STRACE is great for monitoring the interaction between a program and its operating system.

STRACE basically works by *diverting* the calls to the operating system to *wrapper functions* that log the incoming and outgoing data. There are various ways of achieving this. For instance, STRACE could link the program with its own "faked" versions of `read()`, `write()`, and so on that would all do the logging before and after invoking the "real" `read()` and `write()` functions. STRACE goes a more general way, which does not require relinking. On a Linux system, all system calls use one single functionality—a specific *interrupt* routine that transfers control from the program to the system kernel. STRACE diverts this interrupt routine to do the logging.

The same mechanism that STRACE and like tools use for *reporting* the interaction can also be used for *recording* and *replaying* the interaction (actually, this is how recording and replaying input works). For instance, a log as generated by STRACE could be processed by a replay tool. Such a replay tool would no longer invoke the "real" functions but simply have its "fake" functions return the values as found in the STRACE log file.

In Chapter 8 we learn more about obtaining such logs. In particular, *aspect-oriented programming* (see Section 8.2.3) offers elegant and system-independent ways of adding monitoring code to large sets of functions.

However, the true technical problem is less the capturing (or even replaying) of logs than the sheer *amount of data* we have to cope with. As an example, consider a Web server that serves 10 requests per second. Say that each of these requests results in a trace log of about 10 kilobytes. Every hour will thus result in $10 \times 3{,}600 \times 10\,\text{KB} = 360\,\text{MB}$ of trace. A single day will result in 8,640 MB of trace. Given the advances in storage capacity, this may sound feasible. However, you should also consider that whenever you have to reproduce a problem you also have to replay all of this interaction.

An alternative to tracing all of the interaction from scratch is to use *checkpoints*. A checkpoint basically records the entire state of a program such that it can be restored later. This is typically done when the program has reached some *stable state*. In the case of the Web server, for instance, this may be a pause between two requests or transactions. To reproduce a problem, it then suffices to restore the latest checkpoint, and then to replay the interaction since that checkpoint.

There is an obvious trade-off here. States are huge (see Figure 1.3, for instance) and capturing states into checkpoints may take time, and thus one must decide when to prefer checkpoints over recorded interaction. Chapter 14 discusses how to capture program states into checkpoints and how to restore them.

4.3.7 Reproducing Schedules

Modern programs typically consist of several concurrent threads or processes. In general, the *schedule* in which these individual parts are executed is defined by the runtime or operating system, thus abstracting away details that would otherwise burden the programmer. Indeed, a program is supposed to behave identically, whatever the schedule is. Consequently, although the schedule is nondeterministic the program execution should stay deterministic and the programmer need not care about parallelism and nondeterminism (if the program is correct, that is).

Nondeterminism introduced by thread or process schedules is among the worst problems to face in debugging. The following is a simple example. The APACHE Web server provides a number of authentication mechanisms to make sure that only authorized clients can access specific Web pages. One of these authentication mechanisms is the `htaccess` mechanism. If a directory contains a `.htaccess` file, the access to this directory is *restricted*—typically to a set of users with passwords stored in a separate `.htpasswd` file.

To maintain .htpasswd files, APACHE provides a helper program named htpasswd. For instance, the invocation

```
$ htpasswd .htpasswd jsmith
New password: _
```

adds or modifies the password for user jsmith. The htpasswd program prompts the user for the password and stores the user name and the encrypted password in the file .htpasswd.

How can htpasswd ever be nondeterministic? The trouble occurs when multiple users (or processes or threads) are invoking htpasswd at the same time. Normally, each invocation of htpasswd reads the .htpasswd file, modifies it, writes it, and closes it again. Multiple sequential invocations cause no problem, as illustrated in Figure 4.4(a).

However, if two htpasswd processes are running in parallel, bad things can happen, as illustrated in Figure 4.4(b). Some htpasswd process *A* begins and opens the .htpasswd file. Another process *B* does so at the same time and reads in the content. Now, process *A* modifies the content and writes back the .htpasswd file. However, process *B* does so, too, effectively overwriting and undoing the changes made by *A*. As long as write accesses to .htpasswd are scarce, it is unlikely that such a schedule would ever occur, but if it does it will be difficult to reproduce the failure.

There are several solutions to the htpasswd problem (none of which the htpasswd program implements at the time of this writing). The htpasswd could *lock* the .htpasswd file and thus protect it against multiple concurrent updates. It may also *retrieve the last update time* when reading the file, and *check it again*

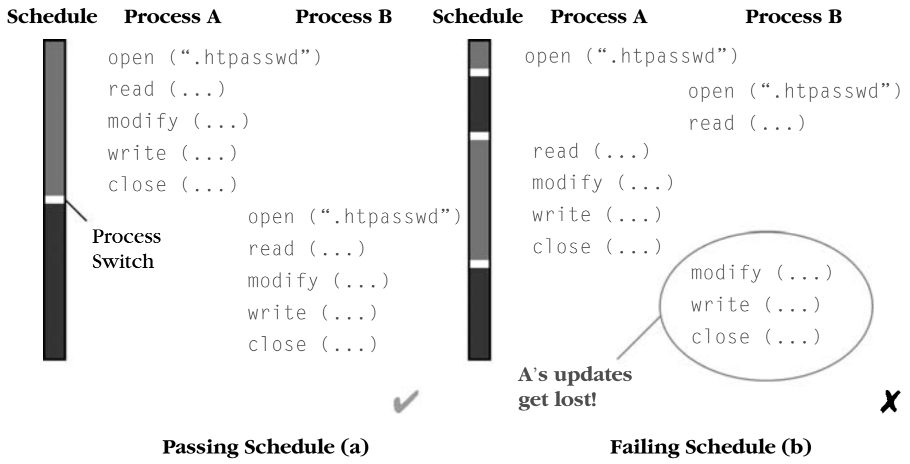

FIGURE 4.4

Differences in schedules may cause problems. If a process switch occurs in the middle of processing a file the second process (b) may undo updates made by the first process (a).

before writing. If the update time has changed in between, the file has changed and must be reread again.

Similar problems (with similar solutions) exist for all resources shared across multiple threads or processes. If multiple threads all write into a shared variable, the thread that writes last wins. Again, the solution would be to introduce locks or language-specific synchronization mechanisms, as described in any good textbook on operating systems or parallel processes.

However, all of these are solutions to a *known* problem, but to identify a problem we must first reproduce it. The general solution is to treat a schedule just like another input source, recording it and replaying it at will. However, as with communications (see Section 4.3.3, on reproducing communication), the amount of data to collect can greatly affect the performance of a system. Another issue is scalability: recording and replaying the thread schedule of a single program has been shown to be practically feasible. However, recording and replaying an entire set of communicating processes, including the schedules as imposed by the operating system, is still a challenge for today's tools.

As programs should behave identically under all possible thread and process schedules, one may attempt to uncover *differences in execution* as induced by schedule differences. Such differences may be uncovered by program analysis (see Chapter 7). For instance, one can verify that all access to shared variables happens in a synchronized fashion. Likewise, massive random testing (see Chapter 3) can uncover problems due to differing schedules.

4.3.8 Physical Influences

Nondeterminism, as induced by thread or process schedules, is just one of the aspects of the real world that programmers deliberately abstract away. Why do they abstract these aspects away? Because they are not supposed to influence the execution of the program.

However, there are many ways to influence the machine on which our program executes. Energy impulses, for instance, can cause bits to flip. Such energy impulses can come over power lines or network communications, but also from alpha particles emitted from the Earth's crust. (*Cosmic rays,* on the other side, have been shown to not influence programs in any way—except maybe in space-borne computers.) Quantum effects may also account for a certain amount of unpredictability. Real-life bugs can also cause failures (recall the tale of the moth caught in a relay, as told in Bug Story 1).

Although computers are typically designed to withstand physical influence, there is a very small chance that such influences may actually cause the program to fail—and as they are extremely rare, physical influences are difficult to reproduce. However, physical influences are also so rare that they can hardly be blamed for a nonreproducible problem. Yet it is common among programmers to blame, say, cosmic rays for an inexplicable problem, to shrug the shoulders, and go away.

Professional programmers should take such physical influences into account only if all other alternatives have been proven to be irrelevant—and if the physical influences can actually be proven. One exception remains, though: If physical influences are likelier than expected, because the physical environment is different from average, then (and only then) are you allowed to take such causes into account. Thus, if the problem occurs in the hot chamber of some nuclear research facility feel free to have someone check for sources of strong magnetic fields or alpha particles.

4.3.9 Effects of Debugging Tools

Another source that can alter a program's behavior is the *debugging process itself.* In fact, simply observing or examining the problem can cause the problem to disappear—or to be replaced by another problem. Problems that behave in this manner are known as *Heisenbugs*—in analogy to Heisenberg's famous uncertainty principle, which states that you cannot observe position and momentum of a particle at the same time with arbitrary precision (the observation itself alters the situation to be observed).

One major source for Heisenbugs are differences between the debugging (i.e., observation) environment and the production environment, combined with undefined behavior of the program. As a simple example, consider the following short C program.

```
int f() {
    int i;
    return i;
}
```

Note that `i` is not initialized. Thus, the value of `i` as returned by `f()` is undefined. "Undefined" means "undefined by the C standard," which again means that whatever `f()` returns it all conforms to the C standard. In practice, though, `f()` will return a definite value on many systems. Multiuser operating systems take care to start processes only with initialized memory (typically zeroed), such that a process may not examine the leftovers of other processes. Thus, in such an environment if `f()` is the first function being called it is likely to always return zero.

Now consider `f()` being executed within an interactive debugger (see Chapter 8). The debugger has no need to clear the leftovers of earlier processes. In particular, if you run the program multiple times, `f()` may return a random leftover from a previous run of the program, which may or may not be zero. Thus, running the program in a debugger alters the program's behavior—which may result in the original problem being masked by another problem, or (worse) the problem not occurring anymore at all.

If you experience a Heisenbug, think about *undefined behavior* in your program. Undefined behavior is almost always caused by a defect. In particular, consider the following:

- Check the data flow of your program to identify uninitialized variables (see Section 7.5 in Chapter 7, on code smells).
- Use system assertions to identify memory that is read before being written (see Section 10.8 in Chapter 10, on system assertions).

BUG STORY 7

A `print` Statement Introduces a Heisenbug

In the midst of a debugging session, I inserted a `print` statement such that I could observe what was going on. To my great surprise, the problem no longer occurred after I had inserted the `print` statement. Even more puzzling, after I removed the `print` statement the problem was still gone, although the program had been reverted to its original state. Well, the problem was gone, but I remained suspicious.

When the problem resurfaced on our production machine, I went on investigating what had gone on. It turned out that there *was* a difference between the original and the reverted program: the executables were different. The problem was caused by a bug in the initial linker: a symbol had been resolved to a bad address. To insert the `print` statement, though, an alternate *incremental linker* had been used—and using this incremental linker fixed the problem.

Some languages are more prone to Heisenbugs than others (in particular, languages where undefined behavior is part of the semantics, such as C and C++). In more modern languages, such as JAVA and C#, almost every single aspect of the program execution is well defined, including forced initialization of variables. Furthermore, these languages come with *managed memory,* giving every memory access a predictable outcome.

You do not necessarily need a debugger to trigger a Heisenbug (any of the debugging techniques discussed in this book can trigger differences in behavior). Among the most obvious issues is that examining a program (interactively or not) can introduce *timing issues*, as discussed in Section 4.3.4. Recompilation for debugging might trigger bugs in the tool chain. Even something as innocuous as a `print` statement can alter the behavior of a program (Bug Story 7). The debugging tools, of course, may themselves be buggy, and this can lead programmers far astray from the actual problem cause.

For these reasons, whenever there is the least suspicion the problem may be a Heisenbug it is always useful to *double check* and to observe the program by at least two independent means. In addition to Heisenbugs, computer jargon also introduced Schroedinbugs, Bohr bugs, and others (List 4.1 lists them all).

4.4 REPRODUCING SYSTEM INTERACTION

As seen in the previous section, the interface between a program and its environment becomes more and more difficult to control the tighter the program is *coupled* to its environment. In other words, the more information the program and environment exchange and the more the program depends on this information, the more difficult it will be to reproduce a given problem.

LIST 4.1: Jargon about Reproducible and Less-Reproducible Problems

Bohr bug (*from quantum physics):* A repeatable bug—one that manifests reliably under a possibly unknown but well-defined set of conditions.

Heisenbug (from Heisenberg's Uncertainty Principle in quantum physics): A bug that disappears or alters its behavior when one attempts to probe or isolate it.

Mandelbug (from the Mandelbrot set): A bug the underlying causes of which are so complex and obscure as to make its behavior appear chaotic or even nondeterministic. This term implies that the speaker thinks it is a Bohr bug, rather than a Heisenbug.

Schroedinbug (MIT: from the Schrödinger's cat thought experiment in quantum physics): A design or implementation bug in a program that does not manifest until someone reading source code or using the program in an unusual way notices that it never should have worked, at which point the program promptly stops working for everybody until fixed.

Source: Raymond (1996).

One way to overcome this issue is to replay not only the program but its environment—in fact, to record and replay every single aspect of the physical machine executing the program. To do so, *virtual machines* (as discussed in Section 3.3.2 in Chapter 3) come in handy.

The REVIRT system uses a virtual machine called UMLinux (not to be confounded with the similarly named User-Mode-Linux) to record and replay all interaction of a virtual machine. It uses a specially modified guest operating system to reduce the overhead of virtualization. Compared to the computation directly on the host, UMLinux virtualization adds an overhead of 58 percent to execution time. REVIRT recording adds another overhead of 8 percent. This indicates that virtual machines are feasible one-size-fits-all solutions when it comes to control, record, and replay program executions—if such a machine is available for the problem at hand. The single major drawback is that the recorded scripts are difficult to read, let alone maintain.

4.5 FOCUSING ON UNITS

Another way of dealing with the coupling between program and environment is to search for *alternative interfaces* that may be controlled, recorded, and replayed. In Section 3.5 in Chapter 3 we saw how to control individual *units* of the program—that is, subprograms, functions, libraries, modules, abstract data types, objects, classes, packages, components, beans, or whatever decomposition the design and the language provide.

The idea now is not to reproduce the execution of the entire program but only *the execution of a specific unit.* This has the advantage that controlling the unit in isolation may be easier than controlling the entire program. The disadvantage,

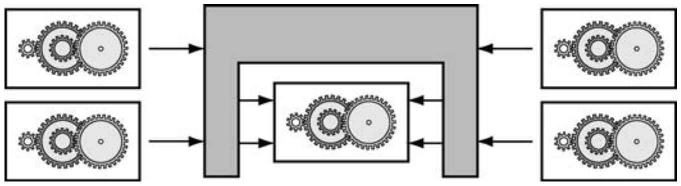

FIGURE 4.5

Controlling a unit's interaction. Setting up a layer for a single unit controls its interaction with the other units.

of course, is that you can only reproduce the behavior of the given unit—and thus must count on the unit producing the problem in isolation.

4.5.1 Setting Up a Control Layer

The basic scheme for controlling a single unit is sketched in Figure 4.5. Again, we set up a layer that isolates the unit from its other units. This *unit control layer* can be used to monitor, record, and replay the input.

A control layer is a generalization of the STRACE tool (discussed in Section 4.3.6). Rather than setting up a layer between program and environment (operating system), we attempt to isolate arbitrary program units from their environment (the remainder of the program). This technique is typically used to isolate *bottom-level* units—that is, units

- with services that are being used frequently by higher-level units, and
- that do not rely on services provided by other units.

Most such bottom-level units are concerned with elementary services such as storage and communication. To reproduce a problem in such a bottom-level unit, it is usually easier to record and replay the interaction at the unit boundary rather than reproducing the entire behavior of the application that uses the unit. The following are some examples:

- *Databases:* To reproduce a problem in a database, record and replay the SQL transactions as issued by an application—rather than reexecuting the entire application.
- *Compilers:* To reproduce a problem in a compiler, record and restore the intermediate data structures—rather than reexecuting the entire front end.
- *Networking:* To reproduce a networking problem, record and replay the basic communication calls as issued by the application—rather than reexecuting the entire application.

4.5.2 A Control Example

As an example of a unit control layer, imagine a simple C++ class that realizes a *mapping* from strings to integers, as follows.

```
class Map {
public:
    virtual void add(string key, int value);
    virtual void del(string key);
    virtual int lookup(string key);
};
```

Our aim is to create a control layer that *logs* all input and output of such a map. We also want to use the log to *reproduce* the interaction between the map and the other units. That is, we need a means of reading the log and invoking the appropriate `Map` methods.

A very simple way of creating such means is to create the log as a stand-alone *program file.* If, for instance, first `add("onions", 4)` and then `del("truffels")` is called, and finally `lookup("onions")` is called, the log file should read as follows.

```
#include "Map.h"
#include <assert>

int main() {
    Map map;
    map.add("onions", 4);
    map.del("truffels");
    assert(map.lookup("onions") == 4);
    return 0;
}
```

This does not look like the log files you are used to, right? This log file can be compiled and executed—and thus reproduces the interaction of a `Map` object with its environment. Note that we use an assertion both to log and to verify the output of the `lookup()` method. This way, the resulting log can also be used for regression testing.

To implement the logging functions, we have to overwrite the original `Map` methods. In an object-oriented language such as C++, a simple way of doing so is to create a *subclass* of `Map` with redefined methods. (A more elegant alternative would be *aspects*, discussed in Section 8.2.3 in Chapter 8.)

```
class ControlledMap: public Map {
public:
    typedef Map super;

    virtual void add(string key, int value);
    virtual void del(string key);
    virtual int lookup(string key);

    ControlledMap();            // Constructor
    ControlledMap();            // Destructor
};
```

Each of the `ControlledMap()` methods actually invokes the method of the `Map` superclass, but also logs the invocation to the `clog` stream. As an example, consider the `add()` method, as follows.

```
void ControlledMap::add(string key, int value)
{
    clog « "map.add(\"" « key « "\", "
         « value « ");" « endl;
    Map::add(key, value);
}
```

We do the same for the deletion method:

```
void ControlledMap::del(string key)
{
    clog « "map.del(\"" « key « "\");" « endl;
    Map::del(key);
}
```

For the `lookup()` methods, we also log the return value and enclose the whole into an assertion.

```
virtual int ControlledMap::lookup(string key)
{
    clog « "assert(map.lookup(\"" « key « "\") == ";
    int ret = Map::lookup(key);
    clog « ret « ");" « endl;
    return ret;
}
```

All three methods have a slight problem: If the key contains a character that cannot be enclosed literally in a C++ string, the resulting program will not compile. A better implementation would thus take care to translate such characters properly.

The constructor and destructors of `ControlledMap()`, called when a `ControlledMap()` object is created and deleted, respectively, add some framework to the `clog` stream such that the output becomes a stand-alone compilation unit.

```
ControlledMap::ControlledMap()
{
    clog « "#include \"Map.h\"" « endl
         « "#include <assert>" « endl
         « "" « endl
         « "int main() {" « endl
         « "    Map map;" « endl;
}

ControlledMap:: ~ControlledMap()
{
    clog « "    return 0;" « endl;
         « "}" « endl;
}
```

How do we use this layer? We simply replace `Map` with `ControlledMap()` for some object in the program and thus have all interaction logged to `clog`. By diverting the log into a program file, we can always reproduce all `Map` interaction simply by compiling and executing the program file. As a side effect, the resulting program files can also be used as *test cases* and thus protect against regression. Such recorded test cases are more abstract than recorded user interaction (see Section 4.3.2 on reproducing input), and are thus more valuable for long-term use (integrating them into unit test tools such as JUNIT is straightforward).

4.5.3 Mock Objects

In the `ControlledMap()` example, we have seen how to set up an object such that it records all of its interaction. In addition to recording, though, we could also set up an object such that it *replays* a previously recorded interaction. This way, we could *replace* an original object with a *mock object*—an object that does nothing but *simulate* the original object by replaying previously recorded interaction. This basic idea is illustrated in Figure 4.6. During capture, a special tool logs all interactions between the original object and its clients. On replay, a mock object replaces the original object, expecting and replaying the previously recorded interaction.

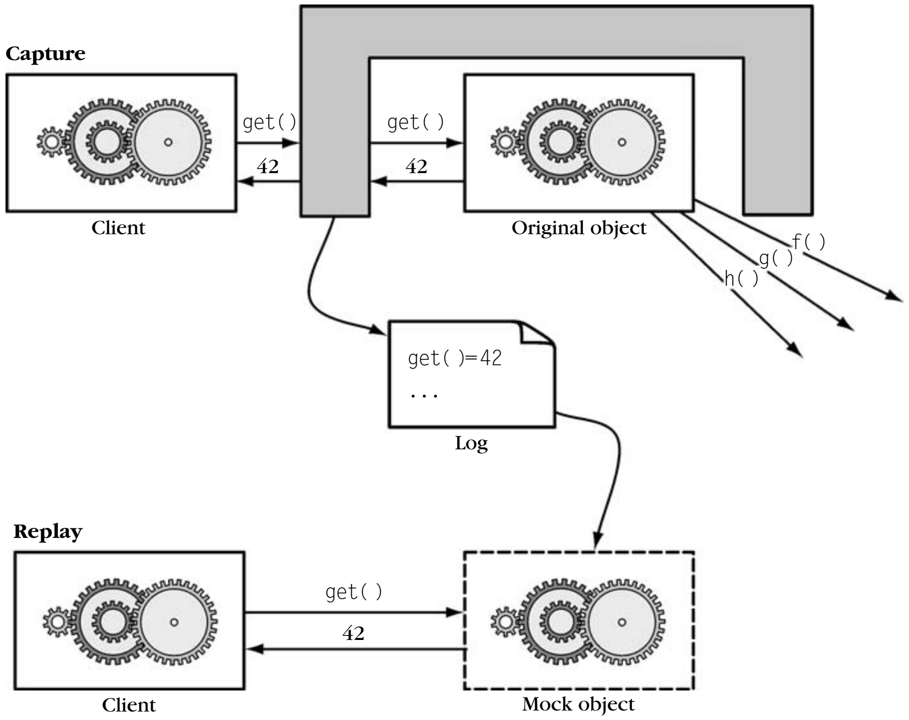

FIGURE 4.6

Replaying unit interaction with mock objects. On replay, the mock object replays the behavior of the original object.

Assume we have a `MockMap` available, which is able to replay interactions recorded earlier by `ControlledMap()`. Replaying the interaction from Section 4.5.2, such a `MockMap` object would:

- Expect a call `add("onions", 4)` and do nothing
- Expect a call `del("truffels")` and do nothing
- Expect a call `lookup("onions")` and return `4`

Note that the `MockMap` does not actually store items; it simply faithfully replays the earlier behavior. For a simple container such as a map, this does not make that much of a difference. For complex items that are difficult to move from one setting to another, however, a mock object can make a huge difference. As an example, consider a *database* installed at a user's site. To reproduce a problem, you normally need to install and replicate the user's database.

This problem can be addressed by turning the database into a capture/replay mock object, as follows:

1. We record the database interaction at the user's site.
2. We then forward the mock object (rather than the database) to the developer.
3. Using the mock object, the developer can reproduce and examine the original run—without going into the hassle of reproducing the entire environment.

Creating mock objects *manually* can be a large amount of programming work, especially for objects with complex interfaces. Recently, though, tools have begun to emerge that allow us to turn arbitrary objects into mock objects. These tools automatically examine the object interfaces and add recording and playback facilities. The tools also take care of the following issues:

- *Return values.* The tool must generate mock objects for *returned values*. A query to a database typically returns a query result, which comes as an object. If the database is turned into a mock object, the query must also return a mock result.
- *Outgoing calls.* The tool must capture and replay *outgoing* calls as well—that is, the mock object calls other objects using previously recorded calls. A mock object for a database, for instance, may call back methods of other objects as soon as a query result is available. Such outgoing calls must also be recorded.
- *Arguments.* The tool must provide mock objects for *arguments* of outgoing calls. In the previous example, a method called back by a mock database must be provided with (mock) arguments.
- *Variables.* The tool must monitor direct read and write access to object variables, such that these accesses can also be mocked.

At the time of this writing, such capture/replay mock objects are still research prototypes. However, the approach can be applied to arbitrary objects (or units) and thus nicely generalizes to all problems recording and reproducing unit behavior.

4.5.4 Controlling More Unit Interaction

Layers as sketched in the previous examples monitor and reproduce only function calls. However, there may be more ways in which the unit depends on its environment. If a unit works in isolation, but not within the application, there must be some *interaction* that is not yet adequately monitored.

- *Variables.* Some units provide variables the application can access to alter the unit's behavior, or to retrieve query results. Such *implicit communication* via variables must also be monitored and controlled, which requires a lot of work unless you use a capture/replay mock tool (as discussed in Section 4.5.3).
- *Other units.* Some units depend on other units, which may also be controlled by the application. Be sure to capture and restore the state of these units as well. If needed, break the dependence (see Section 3.6 in Chapter 3, on isolating units).
- *Time.* Some units (or more precisely, some problems within units) depend on a specific amount of time that must elapse between function calls. If needed, you may wish to record and replay these time intervals.

Obviously, the more possibilities there are for an application to alter the behavior of a unit the more difficult it becomes to isolate the unit from the application. At this point, it becomes clear how a good program design (as discussed in Section 3.7 in Chapter 3) has a positive impact on debugging efforts. The less information being exchanged between units, and the less dependences there are between units, the easier it becomes to control and reproduce unit behavior. Thus, a good design not only makes a program easier to understand, maintain, restructure, and reuse, but also to debug.

4.6 REPRODUCING CRASHES

While recording and replaying arbitrary runs remains a technical challenge, there is a special case in which recording and replaying has shown to be both efficient and effective. When a program *crashes,* its state at the moment of the crash is available for analysis—including all currently active functions, all arguments, and all variables. Could one leverage this state to reproduce the failure? This is the aim of *test case extraction,* which takes a crashing program run and automatically generates a test case that reproduces the crash.

To illustrate the concept of test case extraction, consider the `BankAccount` class in Example 4.3. When the `main()` method is executed with assertions enabled, we obtain the following error message.

```
$ javac BankAccount.java
$ java -ea BankAccount
Exception in thread "main" java.lang.AssertionError: negative
balance
        at BankAccount.withdraw(BankAccount.java:22)
        at BankAccount.main(BankAccount.java:31)
$ _
```

EXAMPLE 4.3: `BankAccount.java`—a simple bank account with assertions

```
class BankAccount {
    public int balance;

    BankAccount(int initial_balance) {
        balance = initial_balance;
    }

    public void deposit(int amount) {
        int old_balance = balance;
        balance = balance + amount;

        assert balance > old_balance: "negative amount";
        assert balance >= 0: "negative balance";
    }

    public void withdraw(int amount) {
        int old_balance = balance;
        balance = balance - amount;

        assert balance < old_balance: "negative amount";
        assert balance >= 0: "negative balance";
    }

    public int balance() {
        return this.balance;
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount(100);
        account.withdraw(1000);

        System.out.println(account.balance());
    }
}
```

The error message reports the stack of functions that were active at the moment of the crash: `main()` invoked `withdraw()`, which in turn threw an `AssertionError` as the `"negative balance"` assertion was violated.

How would we reproduce this failure? While the program state is still available, we can call arbitrary methods on arbitrary objects. In particular, we can *repeat the method calls that just failed*, and see whether this would reproduce the failure. Suppose we repeat the failing method call `account.withdraw(1000)`: We will experience the same failure again and again. All we need is a snapshot of the state (or, more precisely, of the `account` object) at the moment of invocation—and we can easily reproduce calls again and again.

Unfortunately, when the program has crashed, the original state at the moment of invocation is already lost. In the `BankAccount` example, when the error is detected, the balance already is altered. (This is a defect:The `BankAccount` assertions should not only check their postconditions, but also check the arguments as part of the precondition. See Chapter 10 for details.) We therefore must *monitor* the original calls and the state of the moment of the call—a technique that can quickly get expensive. Such monitoring techniques have been explored by two research groups, one at ETH Zurich for the Eiffel programming language in a tool called *Cdd*, and one at MIT for Java programs in a tool called *ReCrash*. They came up with the following three alternatives:

Keep a copy of the calling stack. This technique keeps a *shadow copy* of the call stack and all arguments at invocation time. In our example, the shadow stack would contain the call `account.withdraw(1000)`, keeping a copy of both the argument (1000) as well as the `account` object with its original balance of 100. With this copied information, one can always reproduce the original call.

The advantage of this approach is that it allows for faithful reproduction of arbitrary method calls. Its disadvantage is that keeping a shadow copy induces *overhead*—with the overhead being the lowest if just references are copied, and getting higher the deeper the shadow copy is. In our example, just keeping a reference to `account` would not suffice, as `account` changes its state.

In their experiments, the MIT team experienced overheads between 11 and 42 percent (for just references) and 12,000 and 638,000 percent (for deep state copies). The best compromise was achieved by a *used fields* mode, performing deeper copying on fields that are read or written in the method—as the `account.balance` field in our example—and a shallow copying for all other objects. The *used fields* mode had a overhead of 13–50 percent, and was able to reproduce all crashes faithfully.

Use the failing state instead of the invocation state. To reduce the monitoring overhead, the ETH team explored another technique: Rather than trying to monitor (and later reproduce) the original state, they explored whether one could reproduce the failure in the (possibly modified) state at the moment of the crash.

The advantage of this "failing state" approach is that no monitoring at all is required; the only overhead occurs at the time of the crash, when the entire state is copied for later reproduction. The disadvantage is that the failing state may not be a surrogate for the invocation state. In the `BankAccount` example, this would mean to invoke `account.withdraw(1000)` on an account in the failing state—that is, on an account that already has an invalid balance of −900:

```
account = new BankAccount(-900);
account.withdraw(1000);
```

If we execute this extracted test case, the program fails on the same assertion as the original failure; although the state is different, the essential features of the crash are preserved. However, the example also illustrates how brittle the technique can be: Any assertion against an illegal state in the `BankAccount()`

constructor would yield the approach unusable. Yet, in their experiments, the ETH team was able to reproduce 90 percent of the failures using the state at failure time.

Wait for a second chance. As another alternative to reduce the monitoring overhead, the MIT team suggested a so-called second chance mode. In this setting, the original program is not monitored at all; only after the crash, monitoring is activated only for those very locations involved in the crash. Thus, when the failure occurs a second time, it will be monitored. In our example, for instance, the `withdraw()` method would be monitored as soon as it fails once.

The obvious disadvantage of second chance mode is that it can only reproduce a failure the second time it occurred; also, activating monitoring in an already deployed program may be complicated. The advantage of this approach is that it has zero monitoring overhead in the original (failing) run. Even after monitoring is activated, the overhead stays low, as it focuses only on specific parts of program and state. In their experiments, the MIT team observed an overhead of around 1 percent, which is very tolerable in practice.

A possible synthesis of these three approaches looks as follows. First, try to reproduce the failure with just the failing state, which provides a high chance of reproducing the failure without any monitoring. In case reproduction is not possible, deploy a light weight monitoring and wait for the failure to occur for a second time.

Overall, though, one should keep in mind that these techniques work better the sooner an infection is detected. If an infection is not detected for a long time, the crash may be reproduced, but the origin of the infection is lost. As an example, consider the following code.

```
Record record = getDatabaseRecord("1234");
processRecord(record);
```

Now assume that the `record` returned by `getDatabaseRecord()` is invalid (say, a `null` reference). If we pass it to `processRecord()`, we will experience a crash. *ReCrash* and *Cdd* will be able to reproduce this crash, by rebuilding the invalid `record`, and passing it to `processRecord()`. For the developer, though, the crucial point is not to be able to reproduce just the crash—it is to reproduce the entire chain of events that led to the creation of the invalid record. In the remainder of the book (and particularly in Chapter 13), we explore how to identify such event chains.

On the other hand, if the infection is detected soon enough (say, as a runtime check in `getDatabaseRecord()`), the reproduced crash will point developers directly to the place the defect occurred. It is therefore no surprise that the original concept of reproducing crashes was developed as part of *design by contract,* meaning that every single function of the program is checked for its pre- and postconditions—an approach with several benefits for debugging, and, as we see here, also for reproducing errors. Design by contract is covered further in Chapter 10.

4.7 CONCEPTS

Once a problem is tracked, the next step is to *reproduce* it in *your environment.*

How To

To reproduce a problem:

- Reproduce its environment (Section 4.2)
- Reproduce the execution (Section 4.3 and later)

To reproduce the problem environment:

- Start with *your environment*
- Adopt one circumstance of the *problem environment* after the other

For details, see Section 4.2.

To reproduce the problem execution, place a *control layer* between the program's input and the input as perceived by the program (Figure 4.2). Such a control layer can be used to monitor, control, capture, and replay the program's input.

Technically, one realizes a control layer by intercepting calls to input functions.

Inputs that can be (and frequently must be) controlled include:

- Data (Section 4.3.1)
- User inputs (Section 4.3.2)
- Communications (Section 4.3.3)
- Time (Section 4.3.4)
- Randomness (Section 4.3.5)
- Operating environment (Section 4.3.6)
- Process schedules (Section 4.3.7)

Physics (Section 4.3.8) and debugging tools (Section 4.3.9) can influence a program's behavior in unintended ways.

Executing on a *virtual machine* gives the best possibilities for recording and replaying interaction.

To reproduce unit behavior, place a *control layer* between the unit's input and the input as perceived by the unit (Figure 4.5).

Mock objects can provide a general means of recording and replaying the interaction of arbitrary units.

To reproduce a crash, record the program state at the moment of the crash and attempt to replay the failing methods.

4.8 TOOLS

Winrunner. Tools that record and replay user input are commonly used for testing. HP provides the *Quick Test Professional* testing tool that provides record/replay facilities for Windows. An introduction can be found at *http://mercuryquicktestprofessional.blogspot.com/*.

Android. The ANDROID package, unfortunately, is no longer available for download; an Internet search may help you out.

Revirt. The REVIRT system by Dunlap et al. (2002) allows you to record and replay the interaction of an entire machine. This is found at *http://www.eecs.umich.edu/virtual/*.

Checkpointing Tools

Regarding *checkpointing,* a variety of research tools are available. Such tools allow "freezing" a running process such that it can be resumed later, even on a different machine. For details, see *http://www.checkpointing.org/*.

ReCrash. The *ReCrash* tool for Java, allowing for recording and replaying of crashes, is available at *http://pag.csail.mit.edu/ReCrash*.

Cdd. The *Cdd* prototype, providing automatic test extraction for crashing EIFFEL programs, is now part of the EIFFEL studio development environment, freely available at *http://www.eiffel.com/*.

4.9 FURTHER READING

When it comes to capturing and replaying more than just user interaction, tools are less in a product than in a research prototype stage. Ronsse et al. (2003) give an excellent introduction to the field of making program executions deterministic by recording and replaying their interaction with the environment. They also give pointers on how to replay message passing and shared memory.

Choi and Srinivasan (1998) and Konuru et al. (2000) describe the DEJAVU tool that allows for deterministic replay of multithreaded and even distributed JAVA applications. Among others, DEJAVU records and replays input as well as thread schedules. Mock objects for capturing and replaying object interaction are discussed by Saff and Ernst (2004a). The *Cdd* tool is described in Leitner et al. (2007). Artzi et al. (2008) describe *ReCrash* and their experiments with it.

EXERCISES

4.1 Use ANDROID (or a similar tool) to record and replay a user session with a Web browser. Can you use the script on a different machine or a different window manager?

4.2 A recorded user interaction script typically simply records the *delays* between events, rather than *synchronizing* with the program output. An example of synchronization might be waiting for a dialog window to appear before continuing to simulate input. Discuss the advantages and disadvantages of synchronization.

4.3 Use STRACE (or a similar tool) to monitor the interaction of `ls` (or a similar simple command). For each of the calls reported:

(a) Look it up in the manual.
(b) Estimate the effort for recording and replaying the information passed.

Start with calls you know (say, `open`, `close`, `read`, and `write`) and proceed to lesser-known calls.

4.4 Extend the unit capture scheme of Section 4.5.2 such that the generated log becomes a test case for a unit test framework such as CPPUNIT or JUNIT.

4.5 Which events should be recorded to create a capture/replay tool for:

(a) Random numbers and time events?
(b) Kernel interaction?

How can program design support use this capture/replay tool?

4.6 "If I cannot reproduce a problem, it must be the user's fault." Discuss this statement, given a program with nondeterministic behavior and an environment that is difficult to reproduce.

4.7 "Not every infection of a program state needs to stem from a defect. There could be a bit flip in a memory cell, due to an energy impulse, that caused the infection." Discuss.

4.8 If one uses a tool like *Cdd* or *ReCrash* to reproduce crashes, one can choose an arbitrary function on the call stack to be replayed. What are the advantages and disadvantages of replaying the most recently called functions versus replaying older functions (say, the original invocation to `main()`)?

Here also are huge men having horns four feet long,
and there are serpents also of such magnitude
that they can eat an ox whole.

– MAP INSCRIPTION
Biblioteca Apostolica Vaticana (1430)

For every fact, there is an *infinity* of hypotheses.

– ROBERT PIRSIG
Zen and the Art of Motorcycle Maintenance (1974)