CHAPTER

11 Detecting Anomalies

Although one program run can tell you quite a great deal, having multiple runs to compare offers several opportunities for locating *commonalities* and *anomalies*—anomalies that frequently help to locate defects. In this chapter, we discuss how to detect anomalies in code coverage and anomalies in data accesses. We also demonstrate how to infer invariants from multiple test runs automatically, in order to flag later invariant violations. All of these anomalies are good candidates for infection sites.

11.1 CAPTURING NORMAL BEHAVIOR

If we have a full specification of a program's behavior, we can easily narrow down an infection (as discussed in Chapter 10). So why not simply use assertions all the way through? The following are some reasons:

- Assertions take time to write, especially in otherwise unspecified or undocumented code.
- Assertions over temporal properties ["Don't call `close()` unless you have called `open()`"] or control flow ("If you've taken this branch, you can't take this other branch") are difficult to specify.
- For practical purposes, assertions cannot cover all properties of the entire state at all times—because this would imply a specification as complex as the original program (and just as likely to contain defects). Thus, there will always be gaps between assertions that must be explored.

All of these limits come from the fact that assertions verify against *correct* behavior, which has to be specified by humans. However, there is an alternative: rather than having assertions compare against the correct behavior we could also have assertions compare against *normal* behavior, and thus detect behavior that is *abnormal* (i.e., deviates from the average in some way). Such behavior is characterized by certain *properties* of the program run, such as the following:

- *Code coverage:* Code that is executed in ("abnormal") failing runs but not in ("normal") passing runs.

- *Call sequences:* Sequences of function calls that occur only in ("abnormal") failing runs.
- *Variable values:* Variables that take certain ("abnormal") values in failing runs only.

Of course, knowing about *abnormal* behavior is not as useful as knowing about *incorrect* behavior. Incorrect behavior implies a defect, whereas abnormal behavior implies—well, formally nothing, just abnormal behavior. However, abnormal behavior is often a good *indicator* of defects, meaning that abnormal properties of a program run are more likely to indicate defects than normal properties of the run. Consequently, it is wise to first search for anomalies and then to focus on anomalies for further observation or assertion.

So, how does one capture normal behavior of a program? This is done using *induction* techniques—inferring an *abstraction* from multiple concrete events. In our case, the concrete events are program runs. The abstractions are general rules that apply to the runs taken into account. Typically, such techniques are applied on runs that pass a given test, generating abstractions for "normal" runs. A run that fails that test is then examined for those properties where these abstractions are not met, resulting in anomalies that should be focused on.

In this chapter, we explore a number of automated techniques that use induction to infer abstractions from runs—and then leverage these abstractions to detect anomalies and potential defects. Many of these techniques are fairly recent, and are thus not necessarily ready for industrial primetime. Nonetheless, they should serve as food for thought on how debugging can be further automated. Here, we ask:

HOW CAN WE FIND OUT WHERE A FAILING RUN DEVIATES FROM PASSING RUNS?

11.2 COMPARING COVERAGE

One of the simplest methods for detecting anomalies operates per the following logic:

1. Every failure is caused by an infection, which again is caused by a defect.
2. The defect must be *executed* in order to start the infection.
3. Thus, code that is executed only in failing runs is more likely to contain the defect than code that is always executed.

To explore this hypothesis, we need a means of checking whether code has been executed or not. This can easily be done using *coverage tools*, which instrument code such that the execution keeps track of all lines being executed. Such coverage tools are typically used for assessing the quality of a test suite.

A test suite should execute each statement at least once, because otherwise a defect may not be executed. (More advanced *coverage criteria* demand that each

EXAMPLE 11.1: The `middle` program returns the middle number of three

```
 1  // Return the middle of x, y, z
 2  int middle(int x, int y, int z) {
 3      int m = z;
 4      if (y < z) {
 5         if (x < y)
 6             m = y;
 7         else if (x < z)
 8                m = y;
 9      } else {
10          if (x > y)
11              m = y;
12          else if (x > z)
13              m = x;
14      }
15      return m;
16  }
17
18  // Test driver
19  int main(int arc, char *argv[])
20  {
21      int x = atoi(argv[1]);
22      int y = atoi(argv[2]);
23      int z = atoi(argv[3]);
24      int m = middle(x, y, z);
25
26      printf("middle: %d\n", m);
27
28      return 0;
29  }
```

transition in the control flow graph be executed at least once.) In our case, though, we want to use such coverage tools to compare the coverage of a passing and a failing run.

As an ongoing example, consider the piece of code shown in Example 11.1, computing the middle value of three numbers. This program works nicely on a number of inputs.

```
$ ./middle 3 3 5
middle: 3
$ _
```

It fails, though, on specific inputs—the middle number of 2, 1, and 3 is 2, not 1.

```
$ ./middle 2 1 3
middle: 1
$ _
```

We can now examine the code coverage of these runs, as well as a few more, shown in Example 11.2. Each column stands for a run (with the input values at

EXAMPLE 11.2: Comparing coverage of multiple test runs

		3	1	3	5	5	2
		3	2	2	5	3	1
2	`int middle(int x, int y, int z) {`	5	3	1	5	4	3
3	`    int m = z;`	•	•	•	•	•	•
4	`    if (y < z) {`	•	•	•	•	•	•
5	`        if (x < y)`		•				
6	`            m = y;`		•				
7	`        else if (x < z)`	•				•	•
8	`            m = y;`	•					•
9	`    } else {`	•		•	•		
10	`        if (x > y)`			•			
11	`            m = y;`			•			
12	`        else if (x > z)`						
13	`            m = x;`						
14	`    }`						
15	`    return m;`	•	•	•	•	•	•
16	`}`	✔	✔	✔	✔	✔	✘

•, •: covered statements

Source: Jones et al. (2002).

the top), and each circle stands for a line being executed in the run. The `return` statement in line 15, for instance, has been executed in every single run, whereas the assignment `m = y` in line 8 has been executed in both test cases `3 3 5` and `2 1 3` (shown previously).

This line is somewhat special, too. Every statement that has been executed in the failing run `2 1 3` has also been executed in passing runs. However, line 8 has been executed in only *one* passing test, whereas all other lines have been executed in at least two passing tests. Assuming that a statement is more normal the more often it is executed in passing runs (which indicates it has a low chance of starting an infection), line 8 is the least-normal statement.

(You may also have noticed that lines 12 and 13 are never executed—neither in the failing nor in the passing run. Is this an anomaly we are looking for? No, because the same effect applies in passing as well as in failing runs.)

If we actually focus on line 8 as an anomaly, we could take a look at the conditions under which line 8 is being executed. These are $y < z$ (line 4), $x \geqslant y$ (line 5), and $x < z$ (line 7). In other words, $y \leqslant x < z$ holds. Thus, the middle value is x, and in line 8, m should be assigned x rather than y. Line 8, the greatest anomaly regarding coverage, is indeed the defect that causes the failure.

Such coverage information can be visualized to guide the user in detecting anomalies. Figure 11.1 shows TARANTULA, a tool for visualizing coverage anomalies. In TARANTULA, each nonblank character of the code is shown as a pixel. Each line is assigned a color hue and a brightness, indicating the *anomaly level*.

FIGURE 11.1

Visualizing coverage anomalies with TARANTULA. Each line of code is assigned a specific color. The "redder" a statement the stronger its execution correlates with failure. (*Source:* Jones et al., 2002.)

- *Color.* The redder a statement, the higher the percentage of failing test cases in the test cases that executed this statement. In Example 11.2, line 8 would get the highest amount of red, as 50 percent of the test cases that executed this statement failed. Lines 5, 6, and 9–11, though, would appear in green, as they were only executed in passing test cases.
- *Brightness.* The brighter a statement, the higher the percentage of test cases executing this statement in all test cases. In Example 11.2, of lines 5–11, line 9 would obtain the highest brightness, as it was executed in the most test cases.

What a TARANTULA user would be searching for, then, is bright red lines—statements that are executed in failing test cases only. In the best of all worlds, only a few of these lines would show up—limiting the number of lines to be examined. In a case study, Jones et al. (2002) showed that an anomaly does not necessarily indicate a defect, but that a defect is frequently an anomaly.

- For some defects, the percentage of abnormal code was as low as 3 percent. That is, starting the search at the most abnormal category, the programmer would find the defect after looking at 3 percent of the code.
- The most abnormal category (the 20 percent "reddest") contained at most 20 percent of the code. That is, comparing coverage actually yields small significant differences.
- Eighteen of the 20 defects were correctly classified in the most abnormal category—that is, the one containing 20 percent (at most) of the code.

Thus, in this particular case study, focusing on the abnormal statements allowed programmers to ignore 80 percent of the code or more. Yet, one can still improve this. In past years, researchers have focused on the following extensions to coverage comparison.

Nearest neighbor. Rather than comparing against a combination of all passing runs, it may be wiser to compare only against one passing run—the so-called "nearest neighbor."This nearest neighbor is the passing run the coverage of which is most similar to the failing run. Obviously, the few remaining differences are those that are most strongly correlated with failure—and thus likely to point to the defect.

In a case study, Renieris and Reiss (2003) found that the nearest-neighbor approach predicts defect locations better than any other method based on coverage. In 17 percent of all test runs of the so-called *Siemens* test suite, the defect location could be narrowed down to 10 percent or less of the code. (In Chapter 14, we see how to improve on this result.)

Sequences. As coverage comparison points out, the code of a single method can be correlated with failure. Some failures, though, occur only through a *sequence* of method calls tied to a *specific object.* As an example, consider streams in JAVA. If a stream is not explicitly closed after use, its destructor will eventually close it. However, if too many files are left open before the garbage collector destroys the unused streams, file handles will run out and a failure will occur. This problem is indicated by a sequence of method calls. If the last access (say, `read()`) is followed by `finalize()`, but not `close()`, we have a defect.

In a case study, Dallmeier et al. (2005) applied this technique on a test suite based on the JAVA *NanoXML* parser. They found that sequences of calls always predicted defects better than simply comparing coverage. Overall, the technique pinpointed the defective class in 36 percent of all test runs, with the same low cost as capturing and comparing coverage.

As all of these figures were obtained on a small set of test programs, they do not generalize to larger programs. Nonetheless, these test programs serve as *benchmarks*. If a specific technique works better on a benchmark, it is likely to perform better on other programs. In the future, we will see more and more advanced coverage-comparison tools, demonstrating their advantage on benchmarks as well as on real-life programs.

11.3 STATISTICAL DEBUGGING

In addition to simple coverage, there are other aspects to collect from actual runs. One interesting aspect is exceptional behavior of functions—as indicated by exceptions being raised, or unusual values being returned. If such events frequently occur together with failures, we might have important anomalies that point us to the defect.

The following is an example of how such a technique works, developed by Liblit et al. (2003). Release 1.2 of the CCRYPT encryption tool has a defect: When it comes to overwriting a file, CCRYPT asks the user for confirmation. If the user responds with EOF instead of `yes` or `no`, CCRYPT crashes.

Liblit et al. attempted to use remote sampling to isolate the defect. They instrumented CCRYPT such that at each call site of a function it would maintain three counters for positive, negative, or zero values returned by the function. (In C functions such arithmetic sign values often differentiate whether the call was successful or not.) When CCRYPT terminated, they would be able to tell how often each function had returned a positive, negative, or zero value. This data from 570 call sites, or $570 \times 3 = 1{,}710$ counters, would then be collected for statistical evaluation.

To gather data from actual runs, Liblit et al. generated 2,990 *random runs* from random present or absent files, randomized command-line flags, and randomized user responses (including occasional EOF conditions). These runs were classified into *failing* (crashing) and *passing* runs.

Liblit et al. would then examine the counters. Just as when comparing coverage (Section 11.2), they would search for functions executed only in failing runs, but never in passing runs—or more specifically, for functions returning a specific value category only in failing runs. In other words, the appropriate counter is positive for all failing runs, but always zero for passing runs.

It turns out that in the 2,990 CCRYPT test runs, only 2 out of the 1,170 counters satisfy the following conditions:

- `traverse.c`:320: `file_exists()` > 0
- `traverse.c`:122: `xreadline()` == 0

In other words, the failure occurs if and only if these 2 conditions are met: `file_exists()` returns true, because a file (to be overwritten) exists, and `xreadline()` returns null, because the user did not provide any input.

Such a result is an *anomaly* because it occurs only in failing runs. It is not a defect, though, because both return values are perfectly legal. However, using this knowledge we can easily follow the forward dependences from the call sites and see where the returned values are used—and we will quickly find that CCRYPT does not expect `xreadline()` to return null.

If we use real user input rather than random strings, we should even expect some runs where a file exists (i.e., `file_exists()` > 0 holds) but where the user provides a valid input (such as `yes`), resulting in `xreadline()` $\neq 0$ and a passing

run. This would imply that the predicate `file_exists()>0` is true for at least one passing run—and thus only `xreadline()==0` would remain as the single anomaly correlated with the failure.

11.4 COLLECTING DATA IN THE FIELD

Detecting anomalies from actual executions may require a large set of runs, which is why these are typically *generated* randomly. A far better approach, though, is to use data *collected in the field*—that is, from executions at users' sites.

- The number of executions at users' sites is typically *far higher* than the number of executions during testing.
- Real-life executions produce a greater *variety*. In the CCRYPT example, for instance, the typical behavior of entering "yes" or "no" at a prompt was not covered by the random input.
- In our networked world, collecting and gathering data can easily be automated, as well as the analysis.
- Gathering information from users' runs gives a firsthand indication about how the software is being used—information otherwise difficult to obtain.
- As a side effect, the makers of a software product learn which features are most frequently used and which are not—an important factor when determining the impact of a problem (see Section 2.4 in Chapter 2).

In principle, there is no limit to the information that can be collected. *Exceptional behavior* such as crashes is certainly valuable to the provider (see Section 2.2 in Chapter 2). However, to statistically correlate such exceptional behavior with other aspects of the program run one may also want to monitor function execution or data values.

It is not wise to log everything, though. Two issues have to be considered:

- *Privacy.* Users have a right to privacy, and are very conscious of privacy issues. Section 2.2 in Chapter 2 has details on these issues and how to address them.
- *Performance.* Collecting data impacts the local performance of the system. In addition, forwarding large amounts of collected data over the network entails costs. To improve performance, one can *focus* on a specific part of the product—for instance, collect data only from a few components rather than the entire system. Instead of collecting *all* data, one can also *sample* the logs such that each user executes only a small fraction of the collecting statements.

This sampling approach is actually quite effective. In the CCRYPT example from Section 11.3, Liblit et al. would conduct an experiment in which only 1 out of 1,000 return values of a function was sampled. The impact on performance is less than 4 percent. Of course, sampling only 1 out of 1,000 function returns requires a large number of runs.

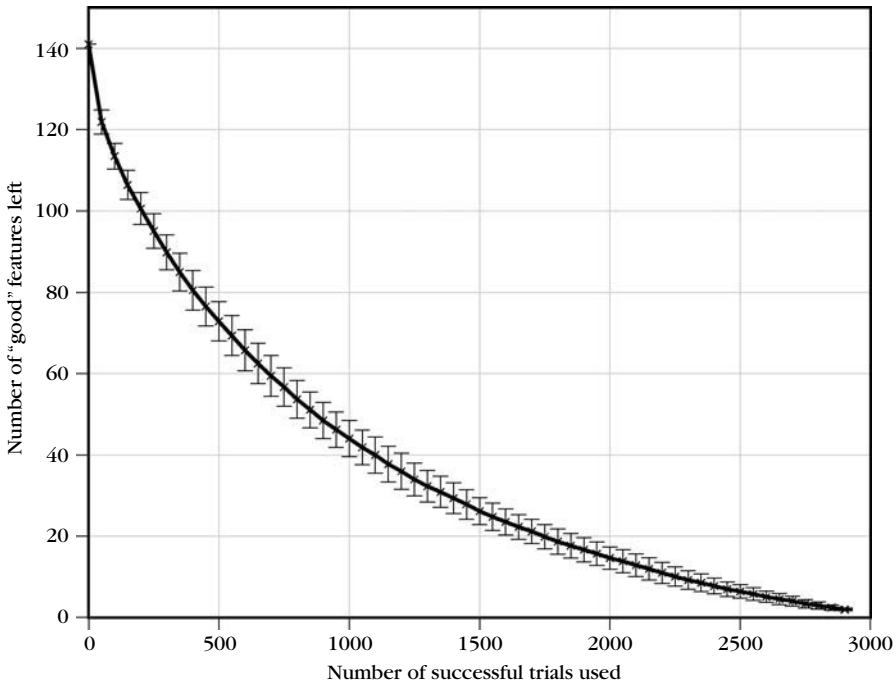

FIGURE 11.2

Narrowing down predicates. Crosses mark means; error bars mark one standard deviation.

Figure 11.2 shows how the function counters discussed in Section 11.3 are eliminated as the number of runs increases. The process starts with 141 out of 1,710 counters that are ever nonzero. One then adds data from one random run after another. After having considered 1,750 runs, the set of remaining predicates has gone down to 20, and after 2,600 runs just 5 are left. Again, these 5 predicates are strongly correlated with failure.

Overall, the results so far demonstrate that statistical sampling can be a powerful, low-overhead tool for detecting anomalies in the field. Now all one needs is users who are willing to have their runs sampled. Liblit et al. (2005) state that "relatively few runs (we used 32,000) are sufficient to isolate all of the bugs described in this paper."

There are situations, though, where several thousand runs are easy to sample. In particular, a centralized *Web service* may be called thousands of times a day—and since there is just one program instance doing the actual work behind the scenes, instrumenting a sample of runs is pretty straightforward, as is collecting data of actual failures. In practice, this means that you get anomalies almost for free—and the higher the number of runs, the more significant the correlation of features and failures will be.

11.5 DYNAMIC INVARIANTS

So far, we have seen how specific aspects of multiple runs can be collected to detect anomalies. Another approach for leveraging multiple runs is to generate *likely specifications* that hold for all runs and to see whether these can be turned into general *assertions*—which can then be used to detect anomalies.

How does one generate specifications from actual runs? A simple yet highly effective approach has been implemented in the DAIKON tool by Ernst et al. (2001). The main idea behind DAIKON is to discover *invariants* that hold for all observed runs. These invariants come in the form of pre- and postconditions. They can be converted into assertions to check for abnormal behavior. To see what DAIKON does, consider the following piece of code.

```
public static int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

What does this code do? We create a set of concrete runs, processing 100 randomly generated arrays with a length from 7 to 13 and elements from [−100, +100]. Running DAIKON on these runs yields two invariants. First is the inferred precondition for `ex1511()`:

```
Ex.ex1511(int[], int):::ENTER
n == size(b[])
b != null
n <= 13
n >= 7
...
```

Obviously, `n` is the size of the array `b[]`. This could serve as an assertion, assuming we can access the size of `b[]`. In addition, in the runs observed `n` is always in the range from 7 to 13—but this is obviously an artifact of the runs we observed. The second invariant is the postcondition for `ex1511()`:

```
Ex.ex1511(int[], int):::EXIT
b[] == orig(b[])
return == sum(b[])
...
```

In the first invariant, the `orig(b[])` clause stands for the "original" value of `b[]`—that is, as the function was entered—and the invariant properly states that `ex1511()` did not change `b[]`'s values. The second invariant states that the return value of `ex1511()` is always the sum (`sum()`) of the elements of `b[]`—and this is precisely what `ex1511()` does.

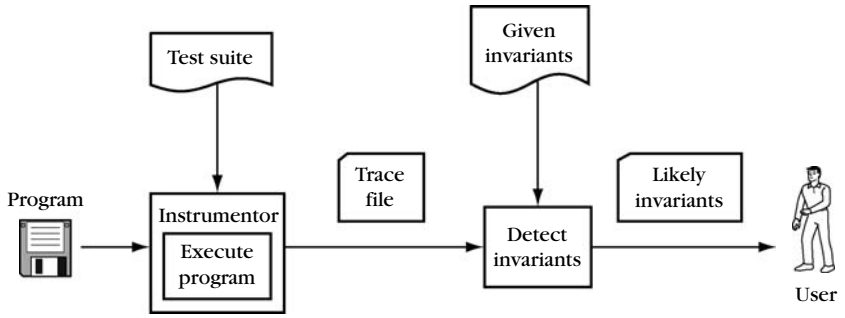

FIGURE 11.3

How DAIKON works. The instrumented program generates a trace file, from which DAIKON extracts the invariants.

How does DAIKON detect these invariants? The general process, as follows, is shown in Figure 11.3.

1. The program to be observed is instrumented at runtime such that all values of all variables at all entries and exits of all functions are logged to a trace file. For C and JAVA programs, DAIKON uses binary instrumentation techniques built on top of VALGRIND (see Section 10.8.3 in Chapter 10). For PERL programs, DAIKON adds instrumentation to the source code in a preprocessing step.
2. When executing the program under a test suite, the instrumented code generates a trace file.
3. DAIKON processes this trace file. DAIKON maintains a library of invariant *patterns* over variables and constants. Each of these patterns can be instantiated with different variables and constants.
   - *Method specifications* come as pre- and postconditions. They can apply to:
     - *Primitive data* such as integers. They compare at most three variables with constants and other variables, as in:

```
x = 6;      x ∈ {2, 5, -30}
x < y;      y = 5 * x + 10;
z = 4 * x + 12 * y + 3;
z = fn(x, y).
```

     - *Composite data* such as sequences, arrays, or lists. For instance, `A subsequence B; x ∈ A; A is sorted.`
   - *Object invariants* such as the following.

```
string.content[string.length] = '\0';
node.left.value ≤ node.right.value
this.next.last = this
```

   Just as method specifications, object invariants can apply to primitive as well as composite data.

For each variable (or tuple of variables), DAIKON maintains a set of potential invariants, initially consisting of all invariants. At each execution point under consideration, for each value recorded in the trace, DAIKON checks each invariant in the set to determine whether it still holds. If it does not, DAIKON removes it from the set.

During this process, DAIKON checks individual variables as well as *derived variables*, such as `orig(b[])` or `sum(b[])` in the previous example. Derived variables also include the return values of functions, to be used in postconditions.

4. While detecting invariants, DAIKON makes some optimizations to make the remaining invariants as relevant as possible. In particular, if some invariant A implies an invariant B, then B need not be reported.
5. After DAIKON has fully processed the trace file, the invariants that remain are those that held for all execution points.
6. DAIKON ranks the invariants by the number of times they actually occurred. An invariant detected 100 times is less likely to be a random effect than an invariant detected 3 times.
7. DAIKON reports the relevant and ranked invariants to the user. The invariants can be fed into another tool for further processing.

The benefits of this technique are as clear as its drawbacks. The most obvious drawback of DAIKON is that the invariants it detects are those built into its library. DAIKON cannot generate new abstractions on its own. For instance, DAIKON can discover that "at the end of `shell_sort()`, the value is sorted" (`sample.c` in Example 1.1), but not that "the value returned is the middle number" (`middle.c` in Example 11.1). This is because DAIKON knows about "sorted" things but not about "middle" numbers. It just lacks the appropriate vocabulary.

It is not too difficult to extend DAIKON 's vocabulary by concepts such as middle elements. In general, as long as some property can be observed it can be added to DAIKON's invariant library. However, the more properties there are to be checked (as well as possible combinations thereof) the longer DAIKON has to run. In regard to the current library, invariant detection time for each program point is cubic in the number of variables that are in scope at this point (as patterns involve three variables at most). Thus, a large number of invariants on a large number of variables will quickly bring DAIKON to its knees. Thus, users need to apply DAIKON to the portion of the code that interests them most.

For the user, the central problem is to *assess* the reported invariants. Are these true facts that hold for every possible execution (which is what users are typically interested in), or do the invariants just hold for the examined runs? On the other hand, if the examined runs are typical for the general behavior, all reported invariants will be helpful for understanding this general behavior.

Keeping these issues in mind, DAIKON is a great tool for program understanding and for summarizing the properties of several runs in a few conditions. DAIKON can output its assertions in JML format, which JML tools (see Section 10.6 in Chapter 10) can thus check for other runs—for failing runs, for instance, as well as for new runs

of a changed version. Whenever a specification is required but a test suite is available, the invariants inferred by DAIKON come in handy as a first take.

11.6 INVARIANTS ON-THE-FLY

A question not yet answered is whether dynamic invariants can be directly used for anomaly detection—without first selecting those that are useful as general specifications. In the CCRYPT example from Section 11.3, we showed how function return values can be correlated with actual failures. Such return values can also be detected and summarized, making dynamic invariants a potential tool for automatic anomaly detection.

This idea has been explored in the DIDUCE prototype of Hangal and Lam (2002). DIDUCE is built for efficiency, and primarily geared toward anomaly detection. It works for a very specific set of invariants only, but has been demonstrated to be effective in detecting anomalies.

Just like DAIKON, DIDUCE instruments the code of the program in question. In contrast to DAIKON, though, DIDUCE works on-the-fly—that is, invariants are computed while the program is executed. For each instrumented place in the program, DIDUCE stores three items:

Coverage. DIDUCE counts the number of times the place was executed.

Values. For each accessed variable, DIDUCE stores the found *value* of the variable read or written. This value is converted to an integer, if necessary, and then stored as a pair (V, M), where

- V is the *initial value* first written to the variable, and
- M is a *mask* representing the *range of values* (the ith bit is 0 if a difference was found in the ith bit, and 1 if the same value has always been observed for that bit).

Formally, if the first value of a variable is W, then $M := \neg 0$ and $V := W$ hold. With each new assignment W', the mask M becomes $M := M \wedge \neg(W' \otimes V)$, where $\otimes$ is the exclusive-or operation.

The following is an example. If some variable i is first assigned a value of 16, then $V = 16 = 10000$ (in binary representation) holds (M is initially $\neg 0 = 11111$). If i is later assigned a value of 18, V is still unchanged, but in M the second bit is cleared because the difference between V and 18 is the second bit. Thus, M becomes 11101.

Difference. For each variable, DIDUCE additionally stores the *difference* between the previous value and the new value. These are again stored as a pair (V, M), as described previously.

If i's value changes from 16 to 18, as described previously, the initial difference V is 2. In the mask M, all bits are set ($M = \neg 0$). If i's value now increases to 21, the new difference between old and new value is $21 - 18 = 3$. The first bit in M is cleared because the increases 2 and 3 differ in the first bit.

Table 11.1 Collecting Invariants in DIDUCE

Code	i	Value V	Value M	Difference V	Difference M	Invariant
`i = 10;`	1010	1010	...11111	–/–	–/–	$i = 10$
`i += 1;`	1011	1010	...11110	1	...11111	$10 \leqslant i \leqslant 11 \wedge \lvert i' - i \rvert = 1$
`i += 1;`	1100	1010	...11000	1	...11111	$8 \leqslant i \leqslant 15 \wedge \lvert i' - i \rvert = 1$
`i += 1;`	1101	1010	...11000	1	...11111	$8 \leqslant i \leqslant 15 \wedge \lvert i' - i \rvert = 1$
`i += 2;`	1111	1010	...11000	1	...11101	$8 \leqslant i \leqslant 15 \wedge \lvert i' - i \rvert \leqslant 2$

The masks, as collected during a program run, imply *ranges* of values and differences that can easily be translated into invariants over values and differences. Table 11.1 outlines how the mask bits become more and more cleared as value variation progresses. Due to the representation, the ranges are not as exact as could be. The representation, though, is very cost effective. The runtime overhead is limited to a few memory operations and simple logical operations for each instrumentation point, and the slowdown factor reported by Hangal and Lam (2002) lies between 6 and 20.

Once one has collected the invariants and finalized the M masks, this representation is just as effective for reporting invariant violations. Whenever DIDUCE observes further variation of M, it reports a violation. Thus, each value or difference out of the range observed previously becomes an anomaly.

Although far more limited than DAIKON, the invariant violations reported by DIDUCE have successfully uncovered defects in a number of programs. The following is an example reported by Hangal and Lam (2002). A multiprocessor simulator exhibited rare and presumably random cache errors. Running DIDUCE in a time interval where no failures occurred resulted in a set of invariants. These invariants were then checked in an overnight run of DIDUCE. It turned out that one violation was produced: a status line, which was usually 0 or 1, suddenly turned out to be 2, and a failure occurred. It turned out that the programmer had not checked this condition properly, and the anomaly reported by DIDUCE quickly pointed him to the defect.

As DIDUCE accumulates invariants during a run, and can be switched from "learning" to "detection" mode without interrupting the application, it is particularly easy to use. In particular, users can start using DIDUCE at the start of the debugging process and can switch between inferring and checking during a run.

11.7 FROM ANOMALIES TO DEFECTS

An anomaly is not a defect, and it is not a failure cause. Yet, an anomaly can be a good starting point for reasoning.

- *Does the anomaly indicate an infection?* If so, we can trace the dependences back to the origins.
- *Could the anomaly cause a failure?* If so, we must understand the effects of the anomaly—for instance, by following the forward dependences through the program.
- *Could the anomaly be a side effect of a defect?* If so, we must trace back the anomaly to the common origin of failure and anomaly.

Overall, the case studies in this chapter have shown that abnormal properties of a program run are more likely to indicate defects than normal properties of the run. Therefore, whenever we have the choice of multiple events we should first focus on the abnormal ones—and using scientific method set up an experiment that checks whether the anomaly causes the failure.

11.8 CONCEPTS

As defects are likely to cause abnormal behavior, anomalies frequently point to defects.

Anomalies are neither defects nor failure causes, but can strongly correlate with either.

How To

To determine abnormal behavior, determine the normal behavior of passing runs and see how the failing run(s) differ. This can be done

- by *comparing* the summarized properties directly, or
- by turning the properties into *assertions*, which can then be used to detect anomalies in failing runs.

To summarize behavior, use inductive techniques that summarize the properties of several program runs into an *abstraction* that holds for all of these runs.

To detect anomalies, researchers so far have focused on *coverage, function return values*, and *data invariants.*

To compare coverage, instrument a program and summarize the coverage for the passing and for the failing runs. Concentrate on statements executed in failing runs only.

To sample return values, at each call site count the numbers within each category of return values. Focus on those categories that occur only in the failing runs.

To collect data from the field, use a sampling strategy such that the impact on performance is minimized.

To determine invariants, use DAIKON or a similar tool to check whether given invariants hold at the instrumentation points.

Whenever we have the choice of multiple events, we should first focus on the abnormal ones.

The techniques discussed in this chapter are fairly recent and not yet fully evaluated.

11.9 TOOLS

DAIKON. The DAIKON tool by Ernst et al. (2001) has had a tremendous impact on the field of dynamic program analysis—not only because of its features but because it is available for download. The DAIKON project page offers software, instructions, and papers on the subject. It is found at *http://groups.csail.mit.edu/pag/daikon/*.

DIDUCE. The DIDUCE tool by Hangal and Lam (2002) is no longer available for download. Its old home page is available at *http://diduce.sourceforge.net/*.

11.10 FURTHER READING

Dynamic program analysis has taken off as a discipline only in the last decade—an explosion largely due to the wealth of computing power we have today ("Why not simply run the program 2,000 times?") and to the presence of cheap communication ("Let's collect all data from all users"). As becomes clear in this chapter, the individual approaches are yet isolated, and it is unclear which approach is best suited for which situation. Yet, the tools and techniques merit to be experimented with—to see how they can help with user's programs.

The TARANTULA tool, developed by Jones et al. (2002), was the first tool to visualize and leverage code coverage for detecting faults. Jones et al. offer several details on the tool and a conducted case study.

Renieris and Reiss (2003) introduced the "nearest-neighbor" concept. I also recommend their paper because of the careful and credible evaluation method. Sequences of method calls were investigated by Dallmeier et al. (2005).

Liblit et al. (2003) were the first to introduce sampling for detecting failures. They describe details of the approach as well as additional case studies. At the time of writing, you could download instrumented versions of common programs that would report sampled coverage data—thus helping researchers to isolate defects in these applications. See *http://sample.cs.berkeley.edu/*.

Remote sampling is also addressed in the GAMMA project under the name of *software tomography* (Orso et al., 2003). GAMMA primarily focuses on sampling coverage information such that the coverage comparison (as described in Section 11.2) can be deployed on a large number of runs.

Podgurski et al. (2003) apply statistical feature selection, clustering, and multivariate visualization techniques to the task of classifying software failure reports. The idea is to bucket each report into an equivalence group believed to share the same underlying cause. As in GAMMA, features are derived from execution traces.

In addition to DAIKON and DIDUCE, other approaches for extracting behavior models from actual runs have been researched—although not necessarily with the purpose of finding defects. Ammons et al. (2002), for instance, describe how to

construct state machines from observed runs that can be used as specifications for verification purposes. Such state machines may also be helpful in uncovering anomalies.

Finally, it is also possible to create abstractions both for passing and failing runs—and then to check a new run as to whether it is closer to one category than the other, effectively *predicting* whether the run will pass or fail. This can be useful for discovering latent defects or otherwise undesirable behavior. Dickinson et al. (2001) describe how to use cluster analysis to predict failures. Brun and Ernst (2004) show how to classify the features of program properties to "learn from fixes." If a program shows properties similar to those that have been fixed in the past, these properties are also likely to be fixed.

EXERCISES

11.1 Using your own words, compare (1) anomaly detection by comparing coverage and (2) anomaly detection by dynamically determined invariants, in terms of:
(a) Efficiency
(b) Applicability to a wide range of programs.
(c) Effectiveness in locating defects.

11.2 Sometimes, bugs reappear that have been fixed before. How can regression testing be used to prevent this? How can regression testing help to detect anomalies?

11.3 Discuss:
- How assertions could be used to detect anomalies.
- Why and why not to use them throughout the program run.

11.4 We have seen that detected anomalies do not necessarily indicate defects (false positives). Explain this phenomenon. What would be false negatives, and how can we explain those?

11.5 What is the basic idea of invariant analysis? What are the advantages and disadvantages of dynamic compared to static techniques?

11.6 Use DAIKON to detect invariants in the `bigbang` program (Example 8.3). Note that you may need to resolve memory issues first.

11.7 DAIKON generates a huge number of invariants. What techniques does it use to reduce that number to *relevant* invariants? Explain the effectiveness of the techniques.

11.8 Consider the `sample` code from Example 1.1 in explaining the effectiveness of DAIKON and DIDUCE. Discuss strengths and limitations of both tools.

11.9 Compare DAIKON and DIDUCE regarding efficiency, usability, scalability, performance, and reliability of the results. Discuss when to use which approach.

11.10 Use the PYTHON `sys.settrace()` facility discussed in Section 8.5 to compare coverage, as in TARANTULA (Section 11.2). Test your approach on the `middle` program (which you reimplement in PYTHON) and the runs shown in Figure 11.2. When comparing coverage, do you observe the same anomalies?

11.11 Use the PYTHON `sys.settrace()` facility discussed in Section 8.5 to check variable ranges, as in DIDUCE (Section 11.6). Apply it on a PYTHON program of your choice and compute the induced overhead.

If you never know failure, how can you know success?

– *The Matrix* (1999)