Software Testing

Gordon Fraser and José Miguel Rojas

Abstract Any nontrivial program contains some errors in the source code. These "bugs" are annoying for users if they lead to application crashes and data loss, and they are worrisome if they lead to privacy leaks and security exploits. The economic damage caused by software bugs can be huge, and when software controls safety critical systems such as automotive software, then bugs can kill people. The primary tool to reveal and eliminate bugs is software testing: Testing a program means executing it with a selected set of inputs and checking whether the program behaves in the expected way; if it does not, then a bug has been detected. The aim of testing is to find as many bugs as possible, but it is a difficult task as it is impossible to run *all* possible tests on a program. The challenge of being a good tester is thus to identify which are the best tests that help us find bugs, and to execute them as efficiently as possible. In this chapter, we explore different ways to measure how "good" a set of tests is, as well as techniques to generate good sets of tests.

All authors have contributed equally to this chapter.

G. Fraser
University of Passau, Passau, Germany
e-mail: gordon.fraser@uni-passau.de

J. M. Rojas
University of Leicester, Leicester, UK
e-mail: j.rojas@leicester.ac.uk

S. Cha et al. (eds.), *Handbook of Software Engineering*,
https://doi.org/10.1007/978-3-030-00262-6_4

1 Introduction

Software has bugs. This is unavoidable: Code is written by humans, and humans make mistakes. Requirements can be ambiguous or wrong, requirements can be misunderstood, software components can be misused, developers can make mistakes when writing code, and even code that was once working may no longer be correct when once previously valid assumptions no longer hold after changes. Software testing is an intuitive response to this problem: We build a system, then we run the system, and check if it is working as we expected.

While the principle is easy, testing *well* is not so easy. One main reason for this is the sheer number of possible tests that exists for any nontrivial system. Even a simple system that takes a single 32 bit integer number as input already has 2^{32} possible tests. Which of these do we need to execute in order to ensure that we find all bugs? Unfortunately, the answer to this is that we would need to execute *all* of them—only if we execute all of the inputs and observe that the system produced the expected outputs do we know for sure that there are no bugs. It is easy to see that executing all tests simply does not scale. This fact is well captured by Dijkstra's famous observation that testing can never prove the absence of bugs, it can only show the presence of bugs. The challenge thus lies in finding a subset of the possible inputs to a system that will give us reasonable certainty that we have found the main bugs.

If we do not select good tests, then the consequences can range from mild user annoyance to catastrophic effects for users. History offers many famous examples of the consequences of software bugs, and with increased dependence on software systems in our daily lives, we can expect that the influence of software bugs will only increase. Some of the most infamous examples where software bugs caused problems are the Therac-25 radiation therapy device (Leveson and Turner 1993), which gave six patients a radiation overdose, resulting in serious injury and death; the Ariane 5 maiden flight (Dowson 1997), which exploded 40 s after lift-off because reused software from the Ariane 4 system had not been tested sufficiently on the new hardware, or the unintended acceleration problem in Toyota cars (Kane et al. 2010), leading to fatal accidents and huge economic impact.

In this chapter, we will explore different approaches that address the problem of how to select good tests. Which one of them is best suited will depend on many different factors: What sources of information are available for test generation? Generally, the more information about the system we have, the better we can guide the selection of tests. In the best case, we have the source code at hand while selecting tests—this is known as *white box* testing. However, we don't always have access to the full source code, for example, when the program under test accesses web services. Indeed, as we will see there can be scenarios where we will want to guide testing not (only) by the source code, but by its intended functionality as captured by a specification—this is known as *black box* testing. We may even face a scenario where we have neither source code, nor specification of the system, and even for this case we will see techniques that help us selecting tests.

Even if we have a good technique to select tests, there remain related questions such as who does the testing and when is the testing done? It is generally accepted that fixing software bugs gets more expensive the later they are detected (more people get affected, applying a fix becomes more difficult, etc.); hence the answer to the question of when to test is usually "as soon as possible." Indeed, some tests can be generated even before code has been written, based on a specification of the system, and developers nowadays often apply a test-driven development methodology, where they first write some tests, and then follow up with code that makes these tests pass. This illustrates that the question of who does the testing is not so obvious to answer: For many years, it has been common wisdom that developers will be less effective at testing their own code as external people who did not put their own effort into building the software. Developers will test their own code gentler, and may make the same wrong assumptions when creating tests as when creating code. Indeed, testing is a somewhat destructive activity, whereby one aims to find a way to break the system, and an external person may be better suited for this. Traditionally, software testing was therefore done by dedicated QA teams, or even outsourced to external testing companies. However, there appears to be a recent shift toward increased developer testing, inspired by a proliferation of tools and techniques around test-driven and behavior-driven development. This chapter aims to provide a holistic overview of testing suitable for a traditional QA perspective as well as a developer-driven testing perspective.

Although many introductions to software testing begin by explaining the life-threatening effects that software bugs can have (this chapter being no exception), it is often simply economic considerations that drive testing. Bugs cost money. However, testing also costs money. Indeed, there is a piece of common wisdom that says that testing amounts to half of the costs of producing a software system (Tassey 2002). Thus, the question of testing well does not only mean to select the best tests, but it also means to do testing well with as few as necessary tests and to use these tests as efficiently as possible. With increased automation in software testing, tests are nowadays often executed automatically: Tests implemented using standard xUnit frameworks such as JUnit are often executed many times a day, over and over again. Thus, the costs of software testing do not only lie in creating a good test set, but also being efficient about running these tests. After discussing the fundamentals of testing, in the final part of this chapter, we will also look at considerations regarding the efficiency of running tests.

2 Concepts and Principles

Software testing is the process of exercising a software system using a variety of inputs with the intention of validating its behavior and discovering faults. These faults, also known as *bugs* or *defects*, can cause failures in their software systems. A fault can be due to errors in logic or coding, often related to misinterpretation of user requirements, and they are in general inevitable due to the inherent human nature of

software developers. Failures are the manifestation of faults as misbehaviors of the software system.

A *test case* is a combination of input values that exercises some behavior of the software under test. A program "input" can be any means of interacting with the program. If we are testing a function, then the inputs to the function are its parameters. If we are testing an Android app, then the inputs will be events on the user interface. If we are testing a network protocol, the inputs are network packets. If we are testing a word processor application, the inputs can either be user interactions or (XML) documents. To validate the observed behavior of the software under test, a test case consists not only of these inputs, but also of a *test oracle*, which specifies an output value (e.g., return value, console output, file, network packet, etc.) or behavior that is expected to match the actual output produced by the software. A collection of test cases is referred to as a *test suite* (or simply a test set).

Example 1 (Power Function) Let us introduce a running example for the remainder of this chapter. The example presented in Fig. 1 consists of a single method `power` which takes two integer arguments `b` (base) and `e` (exponent) as input and returns the result of computing `b` to the power of `e`. Two exceptional cases are checked at the beginning of the function: first, the exponent `e` cannot be negative and, second, the base `b` and the exponent `e` cannot equal zero at the same time. The resulting variable `r` is initialized to one and multiplied `e` times by `b`.

A test case for the `power` function consists of values for the two input parameters, `b` and `e`; for example, the test input $(0, 0)$ would represent the case of 0^0. Besides the test input, a test case needs to check the output of the system when exercised with the input. In the case of input $(0, 0)$, we expect the system to throw an exception saying that the behavior is undefined (see lines 4 and 5 of the example), so the test oracle consists of checking whether the exception actually occurs. If the exception does occur, then the test has *passed*, else it has *failed* and has revealed a bug that requires debugging and fixing.

Although software testing can be, and often is, performed manually, it is generally desirable to automate testing. First, this means that test execution can be repeated as many times as desired; second, it reduces a source of error (the human) during repeated executions; and third, it is usually just much quicker. In

```
1  int power(int b, int e){
2     if (e < 0)
3        throw new Exception("Negative exponent");
4     if ((b == 0) && (e == 0))
5        throw new Exception("Undefined");
6     int r = 1;
7     while (e > 0){
8        r = r * b;
9        e = e - 1;
10    }
11    return r;
12 }
```

Fig. 1 Running example

order to automate execution, a test case requires a *test driver*, which is responsible for bringing the system under test (SUT) into the desired state, applying the input, and validating the output using the test oracle. A common way to create test drivers is to use programming languages; for example, scripting languages like Python are well suited and often used to create complex testing frameworks. A further popular approach lies in reusing existing testing frameworks, in particular the *x*Unit frameworks. We use the name *x*Unit to represent unit testing frameworks that exist for many different programming languages; for example, the Java variant is called JUnit, the Python variant is pyUnit, etc. The following snippet shows two example test cases for the `power` function using JUnit:

```
@Test
public void testPowerOf0() {
  try {
    power(0, 0);
    fail("Expected exception");
  } catch(Exception e) {
    // Expected exception occurred, all is fine
  }
}

@Test
public void testPowerOf2() {
  int result = power(2, 2);
  assertEquals(4, result);
}
```

The first test (`testPowerOf0`) is an implementation of our $(0, 0)$ example, and uses a common pattern with which expected exceptions are specified. If the exception does not occur, then the execution continues after the call to `power` to the `fail` statement, which is a JUnit assertion that declares this test case to have failed. For reference, the snippet includes a second test (`testPowerOf2`) which exemplifies a test that checks for regular behavior, with inputs $(2, 2)$.

The name of testing frameworks like JUnit derives from the original intention to apply them to *unit testing*. This refers to the different *levels* at which testing can be applied. Figure 2 summarizes the main test levels: At the lowest level, individual units of code (e.g., methods, functions, classes, components) are tested individually; in our case this would, for example, be the `power` function. This is now often done by the developer who implemented the same code. If individual units of code access other units, it may happen that these other dependency units have not yet been implemented. In this case, to unit test a component, its dependencies need to be replaced with *stubs*. A stub behaves like the component it replaces, but only for the particular scenario that is of relevance for the test. For example, when "stubbing out" the `power` function, we might replace it with a dummy function that always returns 0, if we know that in our test scenario the `power` function will always be called with `e=0`. In practice, stubbing is supported by the many different *mocking* frameworks, such as Mockito or EasyMock for Java. A *mock* takes the idea of a stub a step further, and also verifies how the code under test interacted with the stub implementation. This makes it possible to identify changes and errors in behavior.

Technically, in a unit test the unit under test should be accessed completely isolated from everything else in the code. There are many followers of an approach

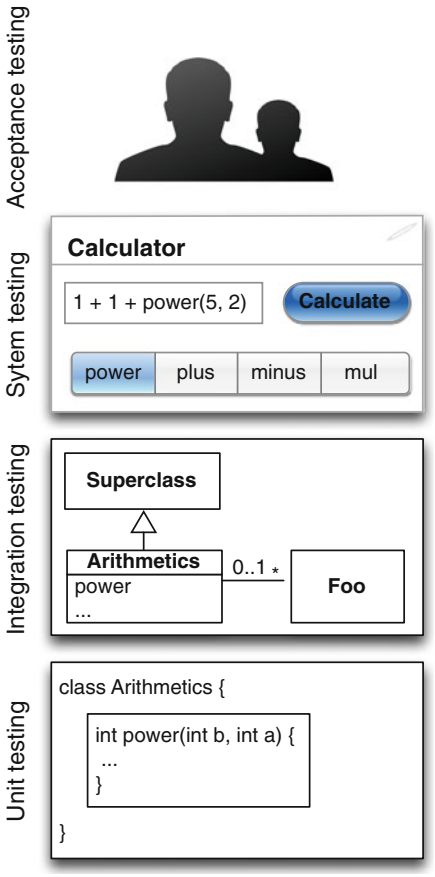

Fig. 2 Test levels illustrated: The `power` function is an individual unit; integration testing considers the interactions between different classes, such as the one containing the `power` function. System tests interact with the user interface, for example, a calculator application that makes use of the integrated units. Acceptance testing is similar to system testing, but it is done by the customer, to validate whether the requirements are satisfied

of "pure" unit testing, which means that everything the unit under test accesses needs to be replaced with mocks or stubs. However, this approach has a downside in that the mocks need to be maintained in order to make sure their behavior is in sync with the actual implementations. Therefore, unit testing is often also implemented in a way that the unit under test may access other units. If tests explicitly target the interactions between different components, then they are no longer unit tests but *integration* tests. In Fig. 2, the integration tests consider interactions between the class containing our `power` function and other classes in the same application. In practice, these are also often implemented using frameworks like JUnit as test drivers, and resort to stubs and mocks in order to replace components that are not yet implemented or irrelevant for the test.

If the system as a whole is tested, this is known as *system testing*. The concept of a test driver still applies, but what constitutes an input usually changes. In our example (Fig. 2), the program under test has a graphical user interface, a calculator application, and test inputs are interactions with this user interface. Nevertheless, it

is still possible to automate system tests using test frameworks such as JUnit; for example, a common way to automate system tests for web application is to define sequences of actions using the Selenium[1] driver and its JUnit bindings. Frameworks like Selenium make it possible to programmatically access user interface elements (e.g., click buttons, enter text, etc.) A very closely related activity is *acceptance testing*, which typically also consists of interacting with the fully assembled system. The main difference is that the aim of system testing is to find faults in the system, whereas the aim of acceptance testing is to determine whether the system satisfies the user requirements. As such, acceptance testing is often performed by the customer (see Fig. 2).

One of the advantages of automating test execution lies in the possibility to frequently re-execute tests. This is particularly done as part of *regression testing*, which is applied every time a system has been changed in order to check that there are no unintended side effects to the changes, and to make sure old, fixed bugs are not unintentionally reintroduced. Regression testing is often performed on a daily basis, as part of continuous integration, but unit tests are also frequently executed by developers while writing code. Because executing *all* tests can be computationally expensive—and even unnecessary—there exist various approaches that aim to reduce the executions costs by *prioritizing* tests, *minimizing* test sets, and *selecting* tests for re-execution.

Even though the existence of convenient testing frameworks and strategies to reduce the execution costs allows us to create large test suites, Dijkstra's quote mentioned in the introduction still holds: We cannot show the absence of bugs with testing, only the presence. This is because for any nontrivial system the number of test cases is so large that we cannot feasibly execute all of them. Consider the `power` example with its two integer inputs. If we assume that we are testing the implementation on a system where integers are 32 bit numbers, this means that there are $2^{32^2} = 1.8 \times 10^{19}$ possible inputs. To exhaustively test the program, we would need to execute every single of these inputs and check every single of the outputs. If we had a computer able to execute and check one billion tests per second (which would be quite a computer!), then running all these tests would still take 585 years. Because executing all tests clearly is not feasible, the main challenge in testing lies in identifying what constitutes an adequate test suite and how to generate it. The main part of this chapter will focus on these two central problems.

Strategies to reason about test adequacy as well as strategies to select tests are often categorized into *white box* and *black box* techniques. In black box testing, we assume no knowledge about the internal implementation of a system, but only about its interfaces and specification. In contrast, white box testing assumes that we have access to the internal components of the system (e.g., its source code), and can use that to measure test adequacy and guide test generation. The latter approach often reduces to considering the structure of the source code; for example, one strategy might be to ensure that all statements in the program are executed by some test. This

[1] Selenium—a suite of browser automation tools. http://seleniumhq.org. Accessed March 2017.

type of testing is therefore often also referred to as *structural testing*. Black box testing, on the other hand, is usually concerned with how the specified functionality has been implemented, and is thus referred to as *functional testing*.

In this chapter, we are mainly considering structural and functional testing techniques, and also assume that the aim of structural testing lies in checking functional behavior. We discuss techniques to manually and automatically derive tests based on both viewpoints. However, in practice there are many further dimensions to testing besides checking the functionality: We might be interested in checking whether the performance of the system is adequate, whether there are problems with its energy consumption, whether the system response time is adequately short, and so on. These are nonfunctional properties, and if the aim of testing is to validate such properties, we speak of *nonfunctional testing*. While nonfunctional tests are similar to functional tests in that they consist of test inputs and oracles, they differ in what constitutes a test oracle, and in the strategies to devise good tests. However, the dominant testing activity is functional testing, and most systematic approaches are related to functional testing, and hence the remainder of this chapter will focus on functional testing.

3 Organized Tour: Genealogy and Seminal Works

3.1 Analyzing Tests

Since we cannot exhaustively test a software system, the two central questions in software testing are (a) what constitutes an adequate test suite and (b) how do we generate a finite test suite that satisfies these adequacy criteria. Since these questions were first posed by Goodenough and Gerhart (1975), they have featured prominently in software testing research. While Goodenough and Gerhard defined a test suite as adequate if its correct execution implies no errors in the program, more pragmatic solutions are based on a basic insight: If some unit of code is not executed, i.e., *covered*, then by definition, testing cannot reveal any faults contained in it. The adequacy of a test suite can therefore be measured by how much of a program is *covered* by the test suite. While ideally one would like to know how much of the possible behavior of the program is covered, there is no easy way to quantify coverage of behavior, and therefore the majority of adequacy criteria revolve around proxy measurements related to program code or specifications.

Many *coverage criteria* have been proposed over time, and a primary source of information is the extensive survey by Zhu et al. (1997). A coverage criterion in software testing serves three main purposes. First, it answers the question of adequacy: Have we tested enough? If so, we can stop adding more tests, and the coverage criterion thus serves as a stopping criterion. Second, it provides a way to quantify adequacy: We can measure not only whether we have tested enough based on our adequacy criterion, but also *how much* of the underlying proxy measurement

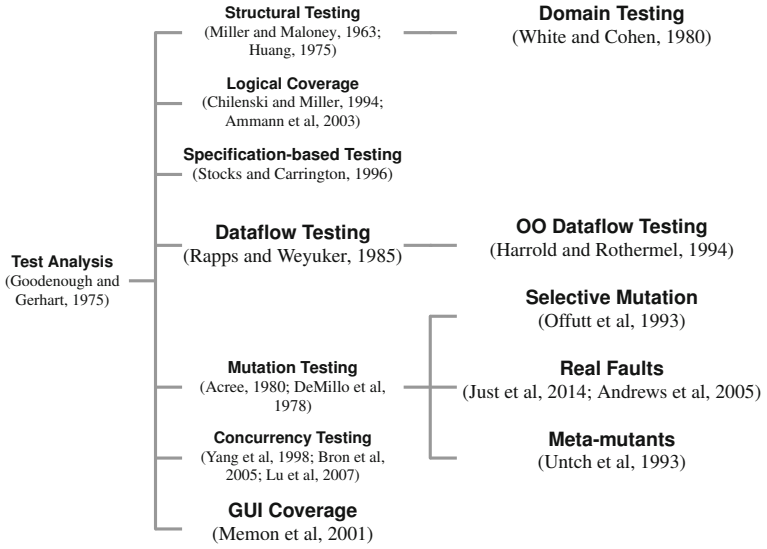

Fig. 3 Analyzing tests: genealogy tree

we have covered. Even if we have not fully covered a program, does a given test suite represent a decent effort, or has the program not been tested at all? Third, coverage criteria can serve as generation criteria that help a tester decide what test to add next.

Coverage can be measured on source code or specifications. Measurement on specifications is highly dependent on the underlying specification formalism, and we refer to the survey of Zhu et al. (1997) for a more detailed discussion. In this section, we focus on the more general notion of *code* coverage. Figure 3 presents a genealogy tree of the work described in this section.

3.1.1 Structural Testing

The idea of code coverage is intuitive and simple: If no test executes a buggy statement, then the bug cannot be found; hence every statement should be covered by some test. This idea dates back to early work by Miller and Maloney (1963), and later work defines criteria with respect to coverage of various aspects of the control flow of a program under test; these are thoroughly summarized by Zhu et al. (1997).

Statement coverage is the most basic criterion; a test suite is considered to be adequate according to statement coverage if all statements have been executed. For example, to adequately cover all statements in Fig. 1, we would need three tests: One where $e < 0$, one where $b == 0$ and $e == 0$, and one where $e > 0$. Because full coverage is not always feasible, one can quantify the degree of coverage as the percentage of statements covered. For example, if we only took the first two tests

from above example, i.e., only the tests that cover the exceptional cases, then we would cover 4 statements (lines 2–5), but we would not cover the statements in lines 6–9 and 11 (the closing brace in line 10 does not count as a statement). This would give us 4/9 statements covered or, in other words, 44.4% statement coverage.

An interesting aspect of statement coverage is that it is quite easy to visualize achieved coverage, in order to help developers improve the code coverage of their tests. For example, there are many popular code coverage frameworks for the Java language, such as Cobertura[2] or JaCoCo.[3] Covered statements are typically highlighted in a different color to those not covered, as in the following example:

```
1  int power(int b, int e){
2  ,  if (e < 0)
3  ,     throw new Exception("Negative exponent");
4  ,  if ((b == 0) && (e == 0))
5  ,     throw new Exception("Undefined");
6     int r = 1;
7     while (e > 0){
8        r = r * b;
9        e = e - 1;
10    }
11    return r;
12 }
```

Statement coverage is generally seen as a weak criterion, although in practice it is one of the most common criteria used. One reason for this is that it is very intuitive and easy to understand. Stronger criteria, as, for example, summarized by Zhu et al. (1997), are often based on the control flow graph of the program under test. Consider Fig. 4, which shows the control flow graph of the `power` function; a test suite that achieves statement coverage would cover all nodes of the graph. However, a test suite that covers all statements does not necessarily cover all edges of the graph. For example, consider the following snippet:

```
1 if(e < 0)
2    e = 0;
3 if(b == 0)
4    b = 1;
```

It is possible to achieve 100% statement coverage of this snippet with a single test where e is less than 0 and b equals 0. This test case would make both if conditions evaluate to true, but there would be no test case where either of the conditions evaluates to false. *Branch coverage* captures the notion of coverage of all edges in the control flow graph, which means that each if condition requires at least one test where it evaluates to true, and at least one test where it evaluates to false. In the case of above snippet, we would need at least two test cases to achieve this, such that one test sets $e < 0$ and the other $e \geq 0$ and one test sets $b == 0$ and the other $b \neq 0$.

[2]Cobertura—a code coverage utility for Java. http://cobertura.github.io/cobertura. Accessed March 2017.

[3]JaCoCo—Java Code Coverage Library. http://www.eclemma.org/jacoco. Accessed March 2017.

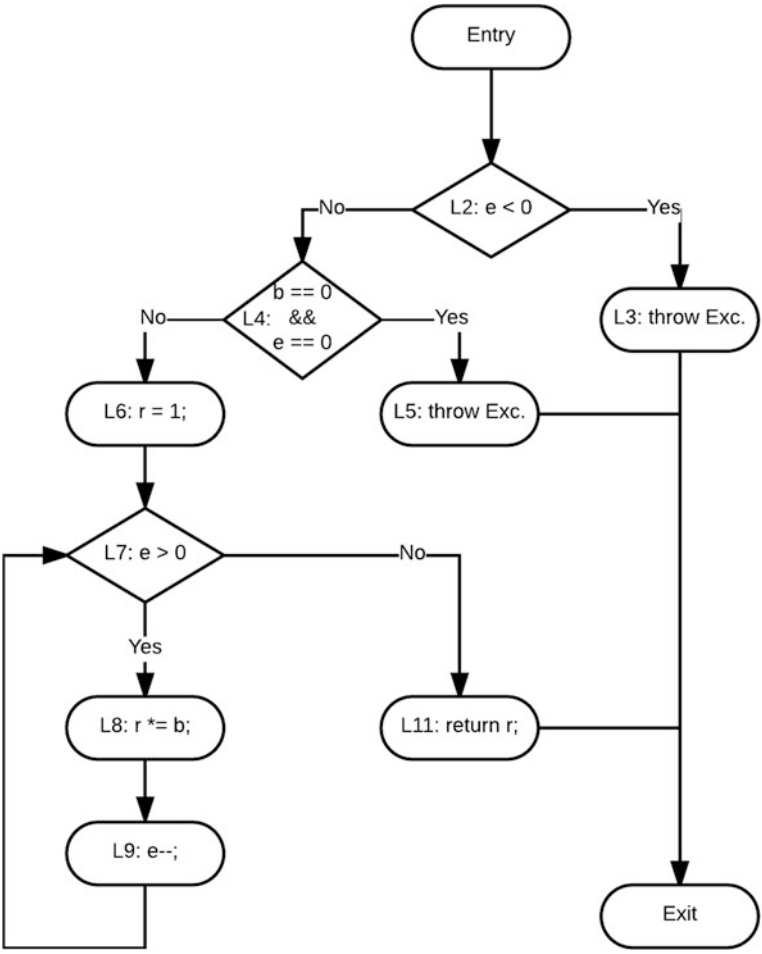

Fig. 4 Control flow graph of the `power` function

Further criteria can be defined on the control flow graph, and the strongest criterion is *path coverage*, which requires that all paths in the control flow graph are covered. This, however, is rarely a practical criterion. Consider Fig. 4: There are two paths defined by the two error checking conditions that end with throwing an exception, but then there is a loop depending on the input parameter e. Every different positive value of e results in a different path through the control flow graph, as it leads to a different number of loop executions. Assuming that e is a 32 bit number (i.e., 2^{31} possible positive values), the number of paths through the `power` function is thus $2^{31} + 2$.

Note that these path-oriented adequacy criteria mainly focus on what is known as *computation errors* (Howden 1976), where the input of the program usually follows

the correct path, but some error in the computation leads to an incorrect result. As *domain errors*, where the computation is correct but the wrong path is chosen, may not always be detected by these criteria, the alternative of *domain testing* (White and Cohen 1980) explores values in particular around the boundaries of domains to check that these are correctly implemented. For more details about this approach to testing, we refer to Sect. 3.3.1.

3.1.2 Logical Coverage Criteria

To reduce the number of possible paths to consider for a coverage criterion compared to the impractical path coverage, but to still achieve a more rigorous testing than the basic statement and branch coverage criteria, a common approach lies in defining advanced criteria on the logical conditions contained in the code. For example, the second if condition in Fig. 1 (`if ((b == 0) && (e == 0))`) consists of a conjunction of two basic conditions. We can achieve full branch coverage on this program without ever setting b to a nonzero value, and so we would be missing out on potentially interesting program behavior. *Condition coverage* requires that each of the basic conditions evaluates to true and to false, such that we could cover the case where $b == 0$ as well as $b \neq 0$. However, covering all conditions in this way does not guarantee that all branches are covered. For example, the pair of tests $b = 0, e = 1, b = 1, e = 0$ covers all conditions yet does not cover the branch where the overall conjunction evaluates to true. A possible alternative lies in *multiple condition coverage*, which requires that all possible combinations of truth value assignments to basic conditions are covered; in this case that means four tests.

As multiple condition coverage may lead to large numbers of tests for complex conditions, *modified condition/decision coverage* (MC/DC[4]) has been proposed as an alternative, and achieving full MC/DC is also required in domains such as avionics by standards like the DOI-178b. MC/DC was defined by Chilenski and Miller (1994) and requires that for each basic condition there is a pair of tests that shows how this condition affects the overall outcome of the decision. MC/DC subsumes branch and condition coverage and selects all interesting cases out of those described by multiple condition coverage. However, there is some ambiguity in the definition of MC/DC, and so Chilenski provides a discussion of three different interpretations (Chilenski 2001). Note that the use of short-circuiting operators in many programming languages has an influence on criteria like multiple condition coverage or MC/DC, as not every combination of truth values of the basic conditions is actually feasible. For example, consider again the example expression `if ((b == 0) && (e == 0))`: To cover the full decision, MC/DC would

[4]There is a slight controversy about whether the acronym should be "MC/DC" or "MCDC." Anecdotal evidence suggests that Chilenski had not intended to use the version with slash. However, this is the version dominantly used in the literature nowadays, so we will also use it.

require covering the case where `b == 0` and `e == 0` are true, and the two cases where either `b != 0` or `e != 0` is true and the other condition if false. However, if `b != 0`, then with short-circuiting the expression `e == 0` would never be evaluated since the overall decision can only be false. This, however, means that a test case (`b != 0`, `e == 0`) cannot be executed.

However, logical conditions are not only found in conditional statements in source code, but can also a part of specifications, ranging from natural language specifications and requirements to formal specifications languages and models. In these settings short-circuit evaluation is rare, and thus criteria like MC/DC are particularly important there. Since there are various slightly differing definitions of MC/DC and the many other criteria for logical conditions, Ammann et al. (2003) summarized these criteria using precise definitions. These definitions also form the basis of their well-known book (Ammann and Offutt 2016).

There are also specific logical coverage criteria for purely Boolean specifications. For example, Weyuker et al. (1994) investigated different fault classes for Boolean specifications written in disjunctive normal form, as well as techniques to automatically generate test data to detect these fault classes. This has opened up a wide range of work on testing of Boolean specifications, for example, criteria such as MUMCUT (Chen et al. 1999) and automated test generation strategies such as the use of model checkers (Gargantini and Heitmeyer 1999) to derive Boolean value assignments to logical formulas so that different logical coverage criteria are satisfied. These approaches are generally related to the wider area of test adequacy criteria based on formal specifications, for example, described in the Test Template Framework by Stocks and Carrington (1996), which defines general strategies on which tests to derive from specifications.

3.1.3 Dataflow Testing

An alternative approach to defining coverage is based on the insight that many errors result from how variables are assigned new data and how these data are used later in the program; this is known as *dataflow testing*. When a variable is assigned a value, this is a *definition* of the variable. For example, Fig. 5a shows the definitions of the `power` function from Fig. 1. Variable r is defined in lines 6 and 8. Variables b and e have an implicit definition at function entry in line 1, and e has a further definition in line 9. There are two different ways variables can be *used*, as shown in Fig. 5b: They can be used as part of a new computation; for example, line 8 contains a use of variable b, and line 9 contains a use of variable e. Alternatively, variables can be used as part of predicates, such as e in lines 2, 4, and 7, and b in line 4. In essence, dataflow testing aims to cover different combinations of definitions and uses.

A pair of a definition of a variable and its use is called a *def-use pair*. There is, however, an important aspect to consider: the execution path leading from the definition to the use must not contain a further definition of the same variable, as the value that would be used would no longer be the one originally defined; the redefinition of the variable is said to *kill* the first definition. For example, the def-use

```
1  int power(int b, int e){
2     if (e < 0)
3        throw new Exception("...");
4     if ((b == 0) && (e == 0))
5        throw new Exception("...");
6     int r = 1;
7     while (e > 0){
8        r = r * b;
9        e = e - 1;
10    }
11    return r;
12 }
```

(a)

```
1  int power(int b, int e){
2     if (e < 0)
3        throw new Exception("...");
4     if ((b == 0) && (e == 0))
5        throw new Exception("...");
6     int r = 1;
7     while (e > 0){
8        r = r * b;
9        e = e - 1;
10    }
11    return r;
12 }
```

(b)

Fig. 5 (**a**) Definitions and (**b**) uses for the variables `b`, `e`, and `r` in the `power` function

Table 1 Definition-use pairs for the `power` function

Variable	Def-use pairs
`b`	(1, 4), (1, 8)
`e`	(1, 2), (1, 4), (1, 7), (1, 9), (9, 7), (9, 9)
`r`	(6, 8), (6, 11), (8, 8), (8, 11)

pair consisting of the definition of r in line 6 of Fig. 1 and the use in line 11 is only covered by a test case that leads to the while loop not being executed at all, for example, by setting e to 0 and b to a nonzero value. Any test where e is greater than 0 would execute line 8 before reaching line 11, and so the definition in line 6 would always be killed by the definition in line 8. Table 1 lists all the definition-use pairs of our example `power` function.

There are different families of dataflow coverage criteria, and these are discussed in depth in the survey paper by Zhu et al. (1997). The seminal paper introducing dataflow coverage criteria was written by Rapps and Weyuker (1985), who define some basic criteria like All-Defs coverage, which requires that for every definition of a variable, there is a test such that the definition is used; All-Uses coverage requires that each possible def-use pair is covered. A related set of criteria (k-tuples) is presented by Ntafos (1988) and discussed in detail by Zhu et al. (1997). However, all these discussions refer only to basic intra-procedural dataflow, whereas object orientation provides new possibilities for dataflow through object and class states; dataflow testing is extended to the object-oriented setting by Harrold and Rothermel (1994).

The dataflow concepts used in dataflow testing date back to the early days of compiler optimizations, and the analysis of dataflow in order to identify faults in source code was popularized in the 1970s (Fosdick and Osterweil 1976). For example, the seminal DAVE framework (Osterweil and Fosdick 1976) enabled detection of various dataflow analyses and was used for a variety of follow-up work. In contrast to the approaches described in this section, however, classical dataflow analysis is a *static* analysis, i.e., it does not rely on program executions, like dataflow testing does.

3.1.4 Mutation Testing

Mutation testing, also known as fault-based testing, was independently introduced by Acree (1980) in his PhD thesis and by DeMillo et al. (1978). The idea is to use the ability to detect seeded artificial faults in the system under test as a means to guide and evaluate testing, rather than relying on structural properties. The overall approach is summarized in Fig. 6: the starting point of mutation testing is a program and a set of tests, which all pass on the program. The first step is to generate artificial faults, which are called *mutants*. Each mutant differs from the original system in only one small syntactical change. Mutants are generated systematically and automatically using *mutation operators*, which represent standard types of mutations and apply these changes at all possible locations in the source code. Many different mutation operators have been proposed in the literature for various different programming languages, and a convention is to give these operators three-letter acronyms. Figure 7 shows an example mutant for the `power` function of Fig. 1 in detail; this mutant is the result of applying the COR (conditional operator replacement) mutation operator to line number 4 of the original version of the program.

Once a collection of mutants has been produced, every test in the test suite under evaluation (i.e., both tests in Fig. 6) is executed on the original system and on each of the mutants. Since a prerequisite for mutation analysis is that all tests pass on the original program, a mutant is "killed" when the test suite contains at least one test that fails when executed on the mutant. To inform the testing process, the *mutation score* is computed to indicate how many of the mutants can be detected (read *killed*) by the test suite. While a mutant that is killed may increase confidence in the test suite and increase the mutation score, the true value may lie in the live mutants,

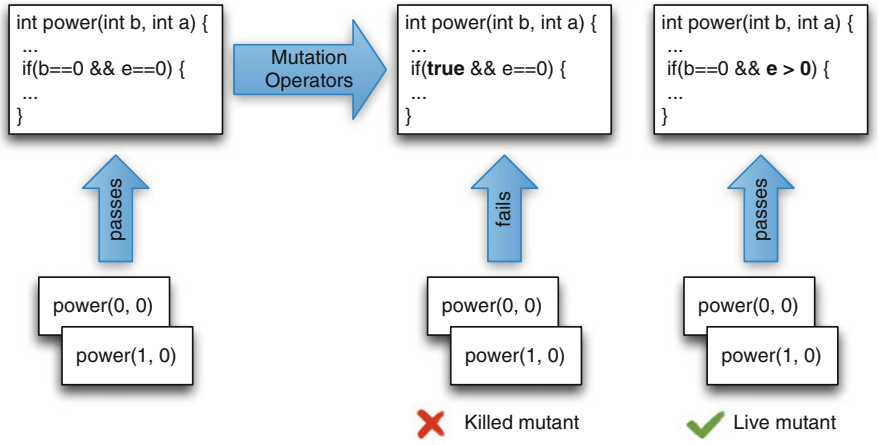

Fig. 6 Overview of the mutation testing process: Mutation operators are applied to the program under test to produce mutants. Tests are executed on all mutants; if a test fails on a mutant but passes on the original program, then the mutant is killed. If there is no test that kills the mutant, the mutant is alive, and likely reveals a weakness in the test suite

```
1  int power(int b, int e){
2     if (e < 0)
3        throw new Exception("Negative exponent");
4     if ((true) && (e == 0))
5        throw new Exception("Undefined");
6     int r = 1;
7     while (e > 0){
8        r = r * b;
9        e = e - 1;
10    }
11    return r;
12 }
```

Fig. 7 Mutant for the `power` function, result of applying the COR operator to line number 4

i.e., those that were not detected by any test: live mutants can guide the developers toward insufficiently tested parts of the system.

Depending on the number of mutation operators applied and the program under test, the number of mutants generated can be substantial, making mutation analysis a computationally expensive process. Several different optimizations have been proposed in order to reduce the computational costs, and Offutt and Untch (2001) nicely summarize some of the main ideas. For example, rather than compiling each mutant individually to a distinct binary, *meta-mutants* (Untch et al. 1993) merge all mutants into a single program, where individual mutants can be activated/de-activated programmatically; as a result, compilation only needs to be done once. Selective mutation is a further optimization, where the insight that many mutants are killed by the same tests is used to reduce the number of mutants that is considered, either by randomly sampling mutants or by using fewer mutation operators. In particular, work has been done to determine which operators are *sufficient*, such that if all the resulting mutants are killed, then also (almost) all mutants of the remaining operators are killed (Offutt et al. 1993).

A limitation of mutation testing lies in the existence of *equivalent mutants*. A mutant is equivalent when, although syntactically different, it is semantically equivalent to the original program. The problem of determining if a mutant is equivalent or not is undecidable in general; hence, human effort is needed to decide when a mutant is equivalent and should not be considered or when a mutant is actually not equivalent and a test should be created to detect it. It is commonly assumed that equivalent mutants are among the reasons why mutation testing has not been adopted by most practitioners yet. Figure 8 presents an example of equivalent mutant for the `power` function. The mutation applied is the ROR (relational operator replacement) operator; the condition of the while loop is changed from `>` to `!=`. The resulting mutant is equivalent because the evaluations of `e>0` and `e!=0` are identical, given that the first **`if`** condition already handles the case `e<0`, and therefore `e` will never be less than 0 when the mutated expression is evaluated.

Two theoretical assumptions are the foundation of mutation testing: the *coupling effect* and the *competent programmer hypothesis*. The coupling effect states that small faults are *coupled* with more complex ones. The implications of this assumption are that by explicitly testing for simple faults, mutation testing implicitly

```
1  int power(int b, int e){
2     if (e < 0)
3        throw new Exception("Negative exponent");
4     if ((b == 0) && (e == 0))
5        throw new Exception("Undefined");
6     int r = 1;
7     while (e != 0){
8        r = r * b;
9        e = e - 1;
10    }
11    return r;
12 }
```

Fig. 8 Equivalent mutant for the `power` function, result of applying ROR to line number 4

tackles more complex ones as well. Furthermore, according to the competent programmer hypothesis, software developers tend to write almost-correct programs, i.e., programs whose faults are due to small syntactical mistakes, which can in practice be simulated during mutation testing. There are two influential empirical studies suggesting that mutants are indeed coupled and thus representative of real faults: The first one is by Andrews et al. (2005), and more recently Just et al. (2014) empirically validated the coupling effect by showing that a strong correlation exists between mutant detection and real fault detection. There is now a substantial body of literature on mutation testing, which has been comprehensively surveyed by Jia and Harman (2011).

3.1.5 Coverage Criteria for Concurrent Programs

With the increasing use of multi-core architectures, software is increasingly making use of concurrency features of modern programming languages. While the testing concepts discussed so far all also apply to concurrent programs, the concurrency provides an additional source of errors, and thus requires dedicated testing effort. In particular, errors arise if multiple parallel threads or processes access some shared memory, then a common problem lies in *race conditions*: If one process writes to a shared variable and another process reads the shared variable, then depending on how the operating system schedules the processes the order of the read and write operation may change and, with it, the result of the program execution. Similarly, *atomicity violations*, where a critical section that needs to be protected is interrupted by the thread scheduler, may cause a program to behave incorrectly. To avoid such sources of errors, various protection mechanisms such as locks have been devised. A lock protects a critical section, and only one process at a time can access it, while all others need to wait for the lock to be released. This, however, offers additional sources of errors. For example, a process may request to access a critical section but never receive the lock, in which it *starves*. If two processes acquire two different locks but then get stuck waiting for the lock held by the other process, the program is in a *deadlock*.

These and other types of concurrency issues are difficult to test for, since the interleaving of processes is difficult to control and the number of possible

interleavings in practice can be huge. However, Lu et al. (2008) analyzed 105 concurrency bugs in open-source applications and found that, despite the large number of possible memory accesses and interleavings, in practice errors manifest with only small numbers of accesses and interleavings. For example, 96% of the examined concurrency bugs analyzed by Lu et al. (2008) manifested if a certain partial order between only two threads was enforced. Similarly, 92% of the examined concurrency bugs were guaranteed to manifest if a certain partial order among no more than four memory accesses is enforced, and 97% of the examined deadlock bugs involved at most two variables. Since it is feasible to find many concurrency issues by covering combinations of low numbers of memory access points, processes, and variables, several coverage criteria have been devised specifically for concurrent programs.

While Yang et al. (1998) showed that the all-definition-use pair coverage can be adapted to explore shared data accesses, there are also more targeted coverage criteria. In particular, synchronization coverage (Bron et al. 2005) was designed with the aim of being as easy to understand and actionable as statement coverage is in for sequential programs. The coverage criterion requires that for each synchronization statement in the program (e.g., synchronized blocks, wait(), notify(), notifyAll() of an object instance), there exists at least one test where a thread is blocked from progressing, and at least one where a thread is allowed to continue. Lu et al. (2007) defined a hierarchy of more rigorous criteria that also consider shared data accesses that are not guarded by synchronization statements. For example, all-interleaving coverage requires that all feasible interleavings of shared access from all threads are covered. Thread pair interleaving coverage requires, for a pair of threads, that all feasible interleavings are covered. Single variable interleaving coverage requires that for one variable all accesses from any pair of threads are covered. Finally, partial interleaving requires either coverage of definition-use pairs or consecutive execution of pairs of access points. Since many of these criteria are either too weak or too complex to be of practical value in a testing context, Steenbuck and Fraser (2013) defined concurrency coverage, which requires covering all possible schedules for sets of threads, variables, and synchronization points, for a parameterized degree of synchronization points, variables, and threads.

3.1.6 Coverage Criteria for Graphical User Interfaces

While many adequacy criteria are defined on the program's source code, the source code often cannot capture program behavior at higher levels of abstraction. When programs have graphical user interfaces (GUIs), then these can be used to capture such aspects. In particular, GUIs consist of different types of user interface elements, i.e., widgets, and users interact with these widgets (e.g., by clicking buttons or entering text). Interactions with the program thus consist of sequences of such interactions, and it is possible to test programs with such sequences of interactions. This idea is captured by the concept of GUI testing, where such sequences are used as tests.

The adequacy of test sets consisting of sequences of GUI events can be captured in models of the user interface. Memon et al. (2001) introduced the notion of an *event flow graph* (EFG). The EFG of a GUI component defines the relation between different interactions, such that nodes of the graph represent events and edges between nodes represent dependencies. For example, a menu-open event for menu `File` would have an edge to the system-interaction events `Open` or `Save`, which are possible events that follow from opening the menu. The example user interface shown in the system testing level of Fig. 2 would contain system interaction events for the buttons *power*, *plus*, *minus*, *mul*, and *calculate* and an event for setting the text of the text field. Based on this graph, different types of coverage criteria can be defined. For example, *event coverage* would require that each event is included in at least one test case, and *event-interaction coverage* would require that all possible pairs of events that can be executed in sequence are covered. Since some events are related by the modal windows they are contained in, criteria like *invocation coverage* require that event opening and closing components (e.g., opening and closing a dialog window) are covered. Covering individual events or pairs of events provides a basic notion of coverage, but ideally one would want to cover longer sequences of events. However, since the number of possible event sequences quickly explodes when considering more than pairs of events, Yuan et al. (2011) applied ideas from combinatorial interaction testing (see Sect. 3.3.1) to identify relevant subsets. In contrast to the more basic criteria defined earlier (Memon et al. 2001), the criteria based on combinatorial interaction testing allow for combinations of any strength and consider the sequential ordering of events (i.e., not only all combinations but also their orderings are considered).

3.2 Generating Tests

Manually generating tests can be a tedious process, and considering the importance of a strong test suite, researchers have long sought to support testers by generating tests automatically. In many cases, automated generation only considers test *data*, i.e., the input data to the system that satisfies a chosen adequacy criterion. Determining whether the tests reveal errors is then deferred to test oracles that have to be added by the testers manually. Figure 9 gives a structural overview of the work covered in this section.

3.2.1 Random Testing

Technically, random testing is may be the simplest test generation technique one could think of: Given the input domain of a program, test data is generated by randomly sampling data points. This is convenient for an engineer who implements the test generation system, as the technical challenges are limited. This is also convenient for a tester, as the test generator requires no further information besides

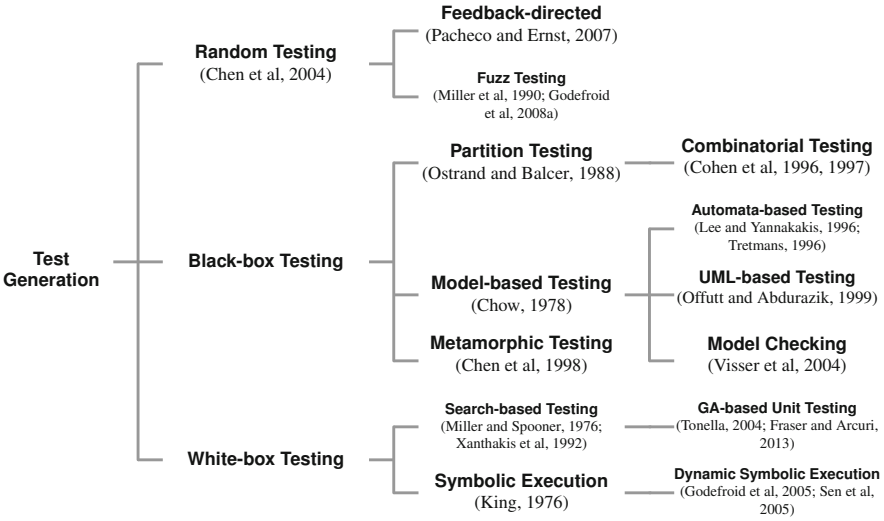

Fig. 9 Generating tests: genealogy tree

a definition of the program's input space. However, as with any form of random sampling, the number of samples typically has to be substantial, and so one typically assumes that there is an automated test oracle to make random testing possible.

While more systematic approaches, like discussed in the subsequence sections, are expected to require fewer tests, there is an often overlooked fundamental difference between random and systematic testing: Systematic test generation may use knowledge about the program in order to select tests likely to expose faults, but only random testing allows derivation of conclusions about the *reliability* of a system under test. In particular, Hamlet's entry on random testing in the *Encyclopedia of Software Engineering* (Hamlet 1994) describes the underlying assumptions: Given an *operational profile*, i.e., a distribution describing user inputs, random testing can be used to calculate reliability properties such as the mean time to failure.

When the aim is not to gain confidence, but to find faults, then random testing can be improved in several ways. A popular version of random testing is *adaptive random testing* (ART), as proposed by Chen et al. (2004): Whereas standard random testing samples input values using an underlying distribution (e.g., a uniform distribution or an operational profile), the idea of ART is to maximize the diversity of the sampled values. Given a random sample, the next test to execute is selected such that it is as different as possible from the previous one, then the next one is selected such that it is as different as possible from the previous two, and so on. An easy way to achieve this is to uniformly sample a number of test inputs and then to select the best one out of the sampled values (e.g., using Euclidian distance if inputs are numeric). For example, consider we started creating a random set of tests T with a single random input $T = \{(3, 8)\}$. To select the next text, with ART we would first

create a set of candidate random inputs, e.g.,

$$t_1 = (10, 0)$$
$$t_2 = (-20, 4)$$
$$t_3 = (7, 29)$$

For each of these candidate tests, the distance to the original test t would be calculated, for example, using the Euclidean distance:

$$d(t, t_1) = \sqrt{(3-10)^2 + (8-0)^2} = 10.63$$
$$d(t, t_2) = 23.35$$
$$d(t, t_3) = 21.38$$

Since the distance to t_2 is largest, this would be the selected test, resulting in $T = \{(3, 8), (-20, 4)\}$. To select a further test input, again a set of candidate inputs would be generated randomly. Then, for each of the candidate inputs, the minimum distance to any test in T is calculated, and the test that is furthest away from all tests in T is then selected as the next test, and so on.

A range of studies building on the work of Chen et al. (2004) have shown that ART produces tests that are more effective at detecting faults. However, this approach loses the advantages of being able to estimate reliability. Furthermore, as Arcuri and Briand (2011) showed, it only improves over random testing if there is a cost to the number of tests (e.g., by requiring a manual test oracle), as otherwise the time spent on selecting tests could be more effectively used to execute random tests. In particular, with increasing numbers of tests selected, calculating the distance quickly becomes more computationally expensive.

A further way to improve the effectiveness of random testing lies in the idea of interleaving test execution and random sampling. This is particularly useful when test inputs are not just basic primitive values, but are incrementally extended. For example, Pacheco et al. (2007) describe what they call "feedback-directed random testing," where sequences of API calls are generated by incrementally extending and executing sequences. As some sequences of calls are better suited to be extended than others, the results of the execution are considered. Those resulting in exceptional behavior are not extended, whereas sequences that represent regular behavior are candidates for extension. Again, this increases effectiveness but at the price of losing confidence.

3.3 Fuzz Testing

A popular, and may be even *the* most popular, application of random testing lies in *fuzz testing*. The idea of fuzz testing goes back to Miller et al. (1990), who discovered that sending random characters as input to Unix command-line utilities can reveal program crashes. Fuzz testing has many applications, and it is particularly used in security-related domains, where discovering exploitable faults is a main concern (Sutton et al. 2007).

Whereas Miller et al. (1990) applied fuzz testing to simple programs receiving raw textual input, programs often require more complex, structured input (e.g., program code). A common approach to fuzz testing in this case lies in randomly modifying well-formed inputs and using the resulting variants as tests (Sirer and Bershad 1999; Forrester and Miller 2000).

Alternatively, when the complex input format is described with a grammar, this grammar can be used to synthesize valid input. Grammar-based testing has been used since the early 1970s for testing compilers and interpreters (Hanford 1970; Bird and Munoz 1983) and now is used for fuzzing complex data. For example, Microsoft's SAGE tool (Godefroid et al. 2008a) is reported to have saved the company millions of dollars' worth of software bugs.

In their series of studies on fuzz testing, Miller's group also explored testing of applications that rely on user interactions (Forrester and Miller 2000). Here, fuzzing consists of sending random events (mouse events, keyboard events). This is now a common approach to testing applications with GUIs, i.e., GUI testing. Since the interactions with the program resemble those of a monkey hitting a keyboard and a mouse, this type of testing is sometimes also referred to as *monkey testing*. A hot topic in the testing research field is GUI testing of Android apps, with the aim to find app crashes. Interestingly, the simple "monkey" tool, which is included in the Android development kit and naively sends random interactions at random screen positions, is still among the most effective automated testing tools for Android apps (Choudhary et al. 2015).

3.3.1 Black Box Techniques

As already indicated in the previous section on random testing, there is a long-standing debate on whether or not random testing is preferable to more systematic approaches. However, in practice the number of tests often is a major concern, in particular when the test oracle needs to be provided by a human but also when test execution itself is expensive. Furthermore, often the objective of testing is not to increase confidence but to find faults. In these cases, more systematic approaches may be more successful. For a detailed discussion of the ongoing debate, the reader may consult the comparisons by Gutjahr (1999), Duran and Ntafos (1984), and Hamlet and Taylor (1990). The main baseline against which random testing is compared in these works is partition testing, and this is also what we consider

first in this section. However, since the start of the random vs. systematic debate, other alternative approaches have been presented, and in particular combinatorial testing techniques have recently gained momentum. To round off this section, we will finally also consider specification- and model-based testing techniques as well as metamorphic testing, which covers the spectrum of black box testing techniques.

Partition Testing

In their discussion of test adequacy, Goodenough and Gerhart (1975) describe the idea that the input space of a program can be partitioned into equivalence classes, such that for each class choosing any test input will test the entire class. This is known as *equivalence partitioning*, and a common assumption is that there is a specification from which these equivalence classes, or partitions, can be derived. For example, if we consider once again to be testing a `power` function implementation, then the specification tells us that the behavior for negative exponents is different from positive exponents, but it is the same for all positive exponents, and the same for all negative exponents. Thus we can, for example, define one input partition for negative exponents and one for exponents greater or equal to 0.

Partitions can, in principle, also be based on source code information, as, for example, described by White and Cohen (1980) in their work on domain testing. Of particular interest in this article is the insight that the choice of any value within an equivalence partition is sufficient to detect if the implementation of the functionality represented by this domain contains errors, but it is not sufficient to determine whether the domain itself has been implemented correctly—i.e., a representative value from inside the partition will not detect *domain errors*.

To avoid this problem, it is important to choose values that are at the *boundaries* of the domains (White and Cohen 1980), an approach known as *boundary value analysis*. There are various interpretations of which boundary values to choose. Given a numerical domain $A \leq x \leq B$, a typical choice would include a value on the lower boundary of the domain (A), at the upper boundary of the domain (B), and a representative value from inside the partition ($A < x < B$). Furthermore, one would typically also include negative examples, i.e., input values on the other side of the boundaries ($A-1$, $B+1$). In the example of our `power` function program, we selected two partitions for the exponent above (negative exponents and exponents greater or equal to 0). Figure 10 illustrates these two partitions, where -1 and 0 are the boundary values, and we selected -10 and $+10$ as representative values

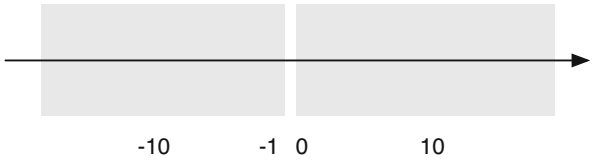

Fig. 10 Partitions with boundary values for the exponent of the `power` function

from within the partitions. Indeed, the implementation for the value 0 has special conditions for this case.

This leaves two main challenges in boundary value testing: The first one is deriving the equivalence classes in the first place, and the second one lies in combining values if there are multiple input parameters for a program. For example, in the `power` program, we have input partitions $[MIN, -1]$, 0, and $[1, MAX]$ for parameters b and e. Boundary values are at MIN, -1, 0, 1, and MAX, and we might choose to also include some representative values from the middle of the domains, for example, -100 and 100. This means we have seven values for each of the two parameters and thus 49 combinations.

To overcome these issues, Ostrand and Balcer (1988) introduced the *category-partition method*, a systematic approach that guides a tester to derive a set of *test frames*, i.e., value assignments to the inputs of the system under test, from the specification of the system. The first step is to identify individual functional units of the system. For each of these functional units, the characteristics of their parameters are analyzed in order to define categories. For each category, the main choices (partitions) are then selected. As the approach is based on a systematic combination of all possible choices for the different parameters and categories, a central aspect of the approach is to restrict possible combinations by defining constraints between choices. For example, a particular choice for one category might lead to an error condition irrespectively of the choices for other categories, and so it is not necessary to build all combinations. A particularly nice aspect of the category-partition method is that the test frames can be generated automatically. However, for each test frame, a test oracle is still required, which most likely is provided manually. This makes the definition of good constraints a central important aspect.

For example, when applying the category-partition method to the `power` function, we might identify the same three partitions as well as seven representative values for each of the inputs as described above. Ostrand and Balcer (1988) used the "Test Specification Language" (TSL) to express these values, which results in a specification as follows:

```
Parameter e:
  MIN.
  -100.
  -1.
  0.
  1.
  100.
  MAX.

Parameter b:
  MIN.
  -100.
  -1.
  0.
  1.
  100.
  MAX.
```

The TSL specification serves as input to automated tools such as TSL generator,[5] which re-implements the original implementation by Ostrand and Balcer (1988). TSL generator will create test frames representing all combinations of different values. In this case, this results in $7 \times 7 = 49$ test frames, each selecting a combination of the values for `e` and `b` (e.g., (`MIN`, -100).

We can add further constraints to reduce the number of combinations. For example, we would declare error cases for `e` for all three values less than 0, which in TSL is done by adding a `[error]` clause. The partition denoted as "error" partition will not be combined with other values but results in a single test frame consisting of only the error value. Under the assumption that the other values do not influence the error case, it is then the tester's choice which other values to combine with this. We might further make the decision to only combine $e = 0$ with $b = 0$, which can be done by declaring a `property` on one partition, and adding a constraint with an `if` condition on the other. Finally, we could avoid exhaustively combining all extreme values (maximum, minimum) as well as the value 1 by declaring them as `single` values. Similarly to the `error` cases, `single` values are only included in a single combination where the choice of other values is up to the tester. Applying these constraints results in the following TSL specification:

```
Parameter e:
  MIN. [error]
  -100. [error]
  -1. [error]
  0. [property e0]
  1. [single]
  100.
  MAX. [single]

Parameter b:
  MIN. [single]
  -100.
  -1.
  0. [if e0]
  1. [single]
  100.
  MAX. [single]
```

This, overall, reduces the number of combinations from 49 to 15, in this example:

```
Test Case 1   <error>
   Parameter e :  MIN

...

Test Case 4   <single>
   Parameter e :  1

...

Test Case 9   (Key = 4.2.)
   Parameter e :  0
   Parameter b :  -100
```

[5] TSL generator for the category-partition method. https://github.com/alexorso/tslgenerator. Accessed February 2017.

```
Test Case 10  (Key = 4.3.)
   Parameter e :  0
   Parameter b :  -1
...
```

Combinatorial Interaction Testing

Combinatorial explosion means that testing systems with many parameters and categories is a real challenge, even when applying systematic approaches like the category-partition method. The example of the `power` function illustrated that even simple programs can result in many different combinations. Often, the number of parameters is larger than that, and environmental factors have a further influence. For example, our `power` function might be compiled on different target platforms with different maximum values (e.g., 16, 32, 64 bit platforms), and just by adding that parameter the total number of possible combinations increases from 49 to 147. It is easy to see how this number can further increase. An illustrative example is that of a standard desktop application which has a properties dialog with many individual options (e.g., think of the Microsoft Word options dialog). Any test generated for Word would, in theory, need to be executed on all exhaustively possible combinations of these options.

An important insight has shaped research on testing methods in the light of the combinatorial problem: In most cases it is not necessary to test *all* combinations, as errors are often exposed by a combination of only a few of the parameters. Kuhn et al. (2004) report empirical data on this phenomenon; for example, just looking at all pairwise combinations of parameter values is already sufficient to detect 77% of the faults reported for the considered real software systems.

For example, assume we want to test our `power` function on different operating systems (Windows, Mac, Linux), each with 32 bit and 64 bit versions and for `b` and `e` three partitions each (<0, 0, >0). In total, this would give us 54 combinations. However, if only considering pairwise combinations, we can get a much smaller test set, as shown in Table 2. In this table, for example, each value of the "Windows" operating system is combined with all possible values for platform, *b* and *e*; the

Table 2 Pairwise test suite for the `power` function example

OS	Platform	*b*	*e*
Windows	64bit	<0	0
Windows	32bit	0	>0
Windows	64bit	>0	<0
Mac	32bit	<0	>0
Mac	64bit	0	<0
Mac	32bit	>0	0
Linux	32bit	<0	<0
Linux	64bit	0	0
Linux	64bit	>0	>0

same holds for all other pairwise combinations. In total, only nine test cases are necessary in order to cover all pairwise combinations.

The challenge for the tester lies in deriving the smallest possible set of tests that covers all combinations of a chosen order (e.g., all pairs). This is a well-known problem in mathematics, where the test sets can be represented as *orthogonal arrays*. However, orthogonal arrays have some limitations, for example, as they require all parameters to have the same number of values. Furthermore, in practice not all combinations of values are possible, and there are often constraints between different values. To address these problems, a large number of techniques have been proposed over the years, and a particularly influential approach is the AETG (Automatic Efficient Test Generation) system introduced by Cohen et al. (1996, 1997), which uses a greedy heuristic to generate combinatorial test suites. AETG is a baseline in the literature on combinatorial testing to the present day, with new techniques striving to reduce the number of tests necessary to cover all the required combinations. There are two survey papers in the field that provide further insights: Grindal et al. (2005) provide an overview over 16 different combination strategies, and a more recent survey by Nie and Leung (2011) categorizes all the work in the area up to 2011.

Model-Based Testing

The black box approaches discussed so far require some specification of the inputs and their constraints. The more aspects of the system are specified, the more guidance we have in generating functional tests. In particular, if the behavior is specified in a *formal* way—i.e., if it is written with some notation that is machine-readable and has defined semantics—then test generation can be automated. This idea is what inspired model-based testing: a model of the system is a kind of specification, which models some aspect of its behavior in a simplified, abstract way (e.g., Chow 1978). The underlying assumption is that it is easier to specify the functional behavior in a model, where the implementation details of the system are abstracted away. Given such a model, we can then automatically derive tests. Even better, given such a model, we can compare whether the response of the system under test to the test inputs matches the behavior described in the model. In other words, if we have a test model, we can automate test generation and the test oracle.

As the test model is usually more abstract than the implementation, actions in the model may not directly map to actions in the system; instead, the abstract actions need to be refined to real inputs on the system. Similarly, the concrete response of the system needs to be abstracted again to compare it to the abstract model response. Thus, the main components of a model-based testing system are (1) a model of the system, (2) an algorithm to derive tests from the model, (3) a test driver that refines model inputs to concrete system inputs, and (4) a test oracle that compares the concrete system response with the abstract model response.

The test driver and oracle by definition have to be specific to the system under test. However, the model formalism and test generation algorithm are generic. There

is a wide spectrum of different formalisms to model program behavior, and each type of formalism has its own techniques to generate tests. A popular and intuitive way to model system behavior is using finite state machines.

A finite state machine is a model that consists of states and transitions between these states. For example, assume the `power` function (Fig. 1) is integrated into a calculator (as shown in Fig. 2). When turning the calculator on, one can enter a number by repeatedly pressing a number of digits; once all digits of the first number have been entered, the user presses the "Pow" button and then enters the digits of the second number. Finally, to calculate the result, the user enters the "Result" button. There are four states in this program: First there is the initial state, where the calculator has been switched on but no buttons have been pressed. Then, while entering the digits for the first number, the program is in the second state. The third state is where the user enters the digits of the second number, and finally the fourth state is the result state. The transitions between each of these states are triggered by user inputs: One gets from initial state to the state of entering digits by pressing a digit button. One gets to entering the second number by pressing the "Pow" button. Finally, the result state is reached by pressing the "Result" button. This defines the states and transitions of our state machine, and these are typically shown in a graphical notation, where circles represent states and arrows between the circles are transitions, as shown in Fig. 11. Mathematically, a basic finite state machine is defined as a tuple (S, s_0, T), where S is a set of states, $s_0 \subset S$ is a set of initial states, and $T = S \times S$ is a transition relation. Many programs may have only one initial state when the program is launched, but it is also common that programs persist some user information and start into different initial states.

To make use of a finite state machine for testing, we need to capture input/output behavior of the underlying program in the model. In most cases, each transition is associated with an input that triggers the transition. In our example, the input events are thus Digit, Pow, and Result. In order to do any sensible testing, we also need the state machine to describe outputs. One way to achieve this is to also associate an output with each transition; resulting state machines are known as Mealy machines. Mathematically, we add an input alphabet I, such that $T = S \times I \rightarrow S$. An alternative is to associate outputs O with states, in which case the state machines

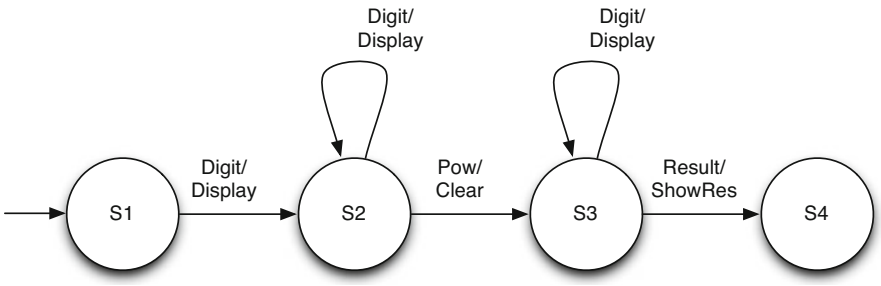

Fig. 11 Finite state machine of a calculator program

are called Moore machines ($O = S \rightarrow O$). Moore machines are usually more common to model hardware systems than software systems, although both are possible. Finally, an alternative is to associate each transition with either an input *or* an output; this is called a labeled transition system. In Fig. 11, each transition is labeled with a pair of values, separated with a "/" symbol; this is a common notation to separate input/output. For example, on pressing the power button (input "Pow"), the system responds by clearing the display (output "Clear").

Given a state machine model with inputs and outputs, the general principle of testing consists of selecting sequences of inputs from the model, applying them to the implementation, and comparing the outputs of the implementation to those of the model. If there are differences between the observed and modeled outputs, then a bug has been found. Note that although this usually is a bug in the implementation, in theory this can also be an error in the model itself. There are different strategies to select sequences of inputs from a model. The most basic strategy, again, is to do so randomly: Starting at the initial state of the model, one picks any outgoing transition from that state; then in the target state, this is repeated. One can either derive complete sequences from the model and then execute them as a whole, which is known as *offline* testing. On the other hand, one can interleave the selection of transitions with execution, for example, done in the TorX tool (Tretmans and Brinksma 2003), an example of *online* testing. Indeed, there are many nuances to model-based testing strategies, some of them obvious and some of them less obvious. The many different dimensions of model-based testing are well captured in the taxonomy of Utting et al. (2012).

In practice, many techniques are based on covering all aspects of a state machine model. For example, Offutt and Abdurazik (1999) generate tests from UML state machines with the objective to satisfy different coverage criteria such as state coverage, transition coverage, or transition-pair coverage. This is also well captured in the seminal book on model-based testing by Utting and Legeard (2010). A particularly popular application of model-based testing currently lies in GUI testing, where models represent states and transitions of an application's user interface (Memon et al. 2001). In particular, several Android-based approaches (Choudhary et al. 2015) make use of some sort of GUI model. It is likely that this approach to model-based testing owes some of its popularity to the fact that the models can be automatically generated using a process called "GUI ripping" (Memon et al. 2003b). In GUI ripping, the user interface of an application is explored automatically, and a model of the application's user interface is created on the fly, during this exploration. This model can then be used to further drive test generation. While the need to manually create a model is removed this way, one loses the advantage of using the model as an automated test oracle. In theory, this means a human test oracle is required; in practice (Choudhary et al. 2015), it means that many model-based GUI testing tools primarily try to find program crashes, which are always undesired.

Conformance Testing

Many different systematic techniques have been proposed in order to detect all mismatches between an implementation and a model; many of these are summarized in detail in the survey of Lee and Yannakakis (1996). While it is accepted that software testing cannot make any guarantees about program correctness or the absence of errors, the availability of a formal model (e.g., a state machine-based model) does allow to draw conclusions about correctness under certain assumptions. In particular, there are different approaches with which to decide whether an implementation *conforms* to the model, i.e., if the behavior of the implementation matches the modeled behavior, and does not have certain types of faults. This is known as *conformance testing*.

There are two main ways in which an implementation can mismatch a state machine model: On one hand, the implementation may make a transition to the correct state but while doing so produce a wrong output; this is called an *output fault* (Fig. 12). On the other hand, the implementation may take a transition to the wrong state—even if the expected output is produced; this is known as a *transfer fault* (Fig. 13). While an output fault would be immediately detected by comparing expected and observed output, a transfer fault may only be detected later on, after further inputs have been sent to the implementation. Besides these two types of

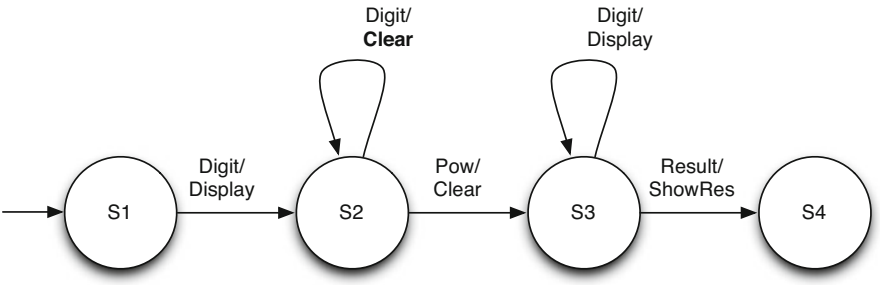

Fig. 12 Output error in the finite state machine of a calculator program

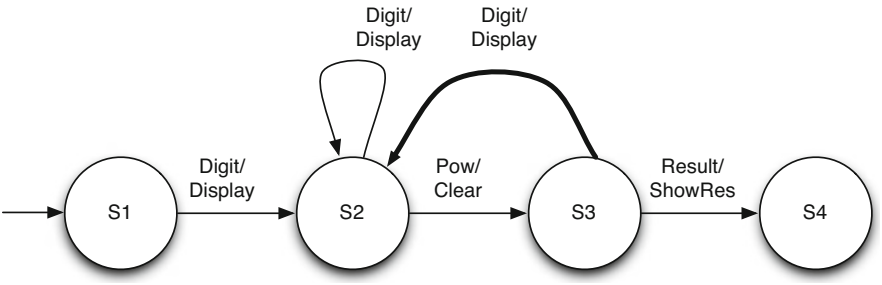

Fig. 13 Transfer error in the finite state machine of a calculator program

faults, an implementation may also miss states entirely or implement extra states that are not part of the specification.

Viewed at a high level, most conformance testing techniques consist of the following steps. For each state S and for each input I for which an outgoing transition from S exists:

1. Reset implementation to initial state.
2. Go to state S.
3. Apply input I.
4. Check output.
5. Verify target state.

This algorithm is simple to implement if one has a direct way to query the state of the implementation and to reset it; it becomes trickier when one has to do either of these things by applying longer sequences of inputs. For example, a unique input/output (UIO) sequence is a sequence of inputs that distinguishes each state from all other states (Sabnani and Dahbura 1988). However, this does not always exist.

In contrast, it is always possible to generate a *set* of sequences that can distinguish any pair of states in a state machine, if one assumes that the state machine is *reduced* (i.e., the number of states is less than or equal to any other equivalent state machine, where two state machines are equivalent if they produce the same sequence of outputs for every possible sequence of inputs). Such a set of sequences is called a *characterizing set*, or W-set, and was derived in the seminal article by Chow (1978). The W-method of testing with final state machines further assumes that the state machine and its implementation are completely specified, i.e., from each state there exists a transition for each possible input. Our original FSM (Fig. 14) satisfies neither of these properties, but it is easy to derive a minimal and complete

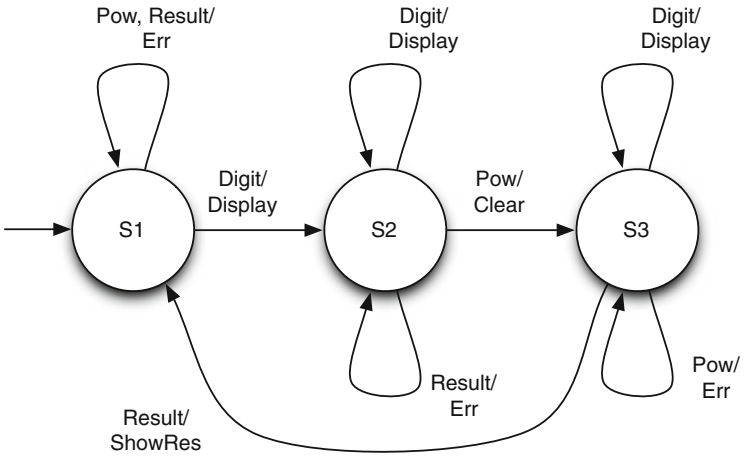

Fig. 14 Complete version of the example FSM: in every state, there is a transition for every input of the alphabet

version by merging equivalent states (S1, S4) and adding transitions for all labels to each state. The result is shown in Fig. 14.

Chow's W-method consists of generating such a W-set as well as a transition cover P, i.e., a set of sequences that reaches all states. In our example, the set W is $W = \{Pow, Result\}$, as these two events lead to a different output for all pairs of states in the minimal FSM. A transition cover P would consist of $\{\epsilon, Pow, Result, Digit \cdot Digit, Digit \cdot Result, Digit \cdot Pow \cdot Digit, Digit \cdot Pow \cdot Pow, Digit \cdot Pow \cdot Result\}$; i.e., there is a sequence for each transition (including ϵ, which is the initializing transition leading to S1). Finally, the overall test set is constructed by using P to reach each state and in each state to apply W to check the conformance of the implementation for that state, which gives us 18 tests.

The W-method has since then been refined for different purposes, and of particular relevance is the Wp-method (partial W-method, by Fujiwara et al. (1991), which generally produces smaller test sets. While these approaches aim for completeness, i.e., they find all the conformance errors, the assumptions made by such approaches may in practice not hold. An alternative approach lies in defining conformance over less restricted labeled transition systems (LTS), which are typically nondeterministic and allow for quiescence (quiet state transitions in the system), thus making the notion of conformance less obvious. The seminal work in this area is Tretmans's *ioco* conformance (input/output conformance) relation (Tretmans 1996). An alternative approach lies in *exhaustively* checking models, i.e., *model checking* (Clarke et al. 1999). While model checking is not a testing technique per se (model checking aims to determine if models satisfy or violate given properties), there is a middle ground between testing and model checking. By interpreting a program as a model of its possible executions, it is possible to apply model checking techniques directly to programs (e.g., Holzmann and H Smith 2001; Visser et al. 2003). Visser et al. (2004) formulate test generation as a model checking problem and leverage symbolic execution to generate tests that satisfy or violate program properties specified as input preconditions. Model checkers have also been used as tools to implement model-based testing techniques. For example, given a state-based model, it is possible to use the counterexample facility of standard model checkers (e.g., SPIN (Holzmann 1997) or SMV (McMillan 1993)) in order to select individual test cases (Fraser et al. 2009); in this approach, coverage goals are represented as negated properties, and then a counterexample to such a negated property (*trap property*) represents a test that satisfies the coverage goal.

Metamorphic Testing

One of the advantages of model-based testing over other testing techniques is the full automation it provides: We can generate test inputs from the model, and we can compare the resulting program behavior with the behavior described by the model to automatically decide when a test failed. Some automated techniques do not even require full models, but work by exploring whether the system under test exposes properties that formalize only part of the overall behavior; for example, the QuickCheck tool (Claessen and Hughes 2011) is an example of such a tool, and

is very popular in the Haskell community. When no specifications or models are available, the decision of whether a system response is correct needs to be made by a human. Making this decision, as well as creating precise models or specifications, can be challenging. However, even when it is difficult to predict the specific output of a program for a specific input, it is sometimes possible to predict how the system response will change after a change to that input. This insight is underlying the idea of *metamorphic testing*, initially proposed by Chen et al. (1998).

The prototypical example used in the literature on metamorphic testing is that of an implementation of a sine function. Without an alternative implementation, it is difficult to predict the exact output of the function for a specific number, for example, $sin(17)$, and even when seeing the resulting output value, it is difficult to judge whether the implementation is correct. However, from mathematics we know that $sin(x) = sin(\pi - x)$. The idea of metamorphic testing is to use an existing test, such as $sin(17)$, and to derive a follow-up test $sin(\pi - 17)$, and then without knowing the concrete values of the computation, we can still check if the two outputs are the same. If they are not, then we know that there is an error in the implementation of the sin program.

Considering out example `power` function (Fig. 1), a possible metamorphic property we might come up with is that x to the power y is the same as x to the power of $y - 1$ times x:

$$power(x, y) = x \times power(x, y - 1)$$

Given any test case for $power$, for example $t_1 = power(10, 3)$, we can create a follow-up test $t_2 = power(10, 2)$ and check that $t_1 = 10 \times t_2$. The nice thing is that given such a property, testing becomes fully automated again: We can generate any number of test inputs for $power$, and by generating a follow-up test for each of them, we can check the correctness automatically.

At first it might seem that metamorphic testing is a technique specific for numeric programs; however, metamorphic testing applies to a wide range of application domains. Segura et al. (2016) provide an extensive survey including applications, which covers web services and applications, computer graphics, simulation and modeling, and embedded systems.

3.3.2 White Box Techniques

White box testing in general refers to any testing techniques that consider the source code of the program. Often, these techniques are also known as *structural* testing techniques because they are guided by the structure of the program code. In practice, the world is not black and white—for example, one can use a black box testing technique to select tests and measure their code coverage at the same time. In this chapter, however, we will focus on techniques that consider only the source code. When testing guided by a program's source code, the aim of test generation often reduces to the task of reaching a certain point in a program or following a certain

execution path through the program. This could be, for example, in order to satisfy a coverage objective, such as covering all statements of the program.

Search-Based Testing

Search algorithms are at the core of computer science, but searching for program inputs is not straightforward as the number of inputs for any real program is far too high to explicitly enumerate them all. Consequently, the search for tests needs to be informed by heuristics and needs to use algorithms that can cope with the complex structure and properties of test data. The earliest known publication on search-based testing dates back to 1976: Miller and Spooner (1976) reformulated paths as straight-line programs, where constraints were dynamically solved through numerical maximization techniques. Tests were executed, and these executions were guided toward the required test data using a fitness function, which rewarded inputs that are close to the target test data. Since then, many different heuristics and algorithms have been proposed. As these are generic search algorithms that can be adapted to different problems by supplying a heuristic, they are often referred to as "meta-heuristic" search algorithms.

Search-based testing is a part of a larger movement in software engineering, where meta-heuristic search algorithms are used to solve complex software engineering problems. This field dates back to Miller and Spooner's early work on test data generation but saw a large surge in popularity much later, around the time the term "search-based software engineering" (SBSE) was coined by Harman and Jones (2001). Search-based testing covers everything in SBSE related to testing, which ranges from functional testing (Bühler and Wegener 2008) to test prioritization (Li et al. 2007), test selection (Yoo and Harman 2007), combinatorial interaction testing (Cohen et al. 2003), model-based testing (Derderian et al. 2006), and many others. A good overview of the field and a useful introduction to search-based testing in general are provided by McMinn's survey (McMinn 2004); a more recent snapshot of the literature is provided by Ali et al. (2010). In this section, we are focusing on structural test data generation, which is why it is included as white box technique, even if there are also search-based black box testing approaches.

There are some choices to be made when generating test data for structural testing: First, one needs to select a meta-heuristic search algorithm. Second, one needs to find a suitable representation that encodes test data (i.e., program inputs, sequences of calls, etc.) in a suitable way for that search algorithm. This means that the operators used by the search algorithm (e.g., to explore the neighbors of a given candidate solution in the search space) need to be defined for that representation. Third, one needs to define a fitness function that the search algorithm can use to find the best possible solution. This fitness function is generally independent of the algorithm and the representation, and so we start by looking at this.

The dominant fitness function in use today consists of two parts: The *approach level* was introduced by Wegener et al. (2001) and measures the distance in the control flow between a given test execution and a target statement. For example,

assume we want to generate inputs for our `power` function example (Fig. 1) in order to reach line 8, which is in the while loop. The approach level tells us how close a given execution was to reaching the target (line 8) in terms of the control flow. An initial idea might be to measure the distance between a given execution and the target node in the control flow graph (Fig. 4), and indeed in our example this would be possible. However, in practice not all branches are of relevance for reaching a target branch. For example, assume the then-branch of the second if condition (line 4) would only produce a warning message, but continue execution after that. In this case, whether or not this branch evaluates to true or false is irrelevant for reaching the target line 8. In contrast, in our version of the program, an exception is thrown in line 5, and so it is necessary for the if condition to evaluate to false such that we can reach our target statement. This dependency relation between nodes of the control flow graph is captured in the *control dependence graph*. Loosely stated, a node A is control dependent on another node B if every path in the CFG to A goes through B, but not every path through B also leads to A. The control dependence graph can be generated by determining all dominators and post-dominators (see Young and Pezze (2005) for a gentle introduction to static analysis and dominator/post-dominator calculation). Figure 15 shows the control dependence graph of our

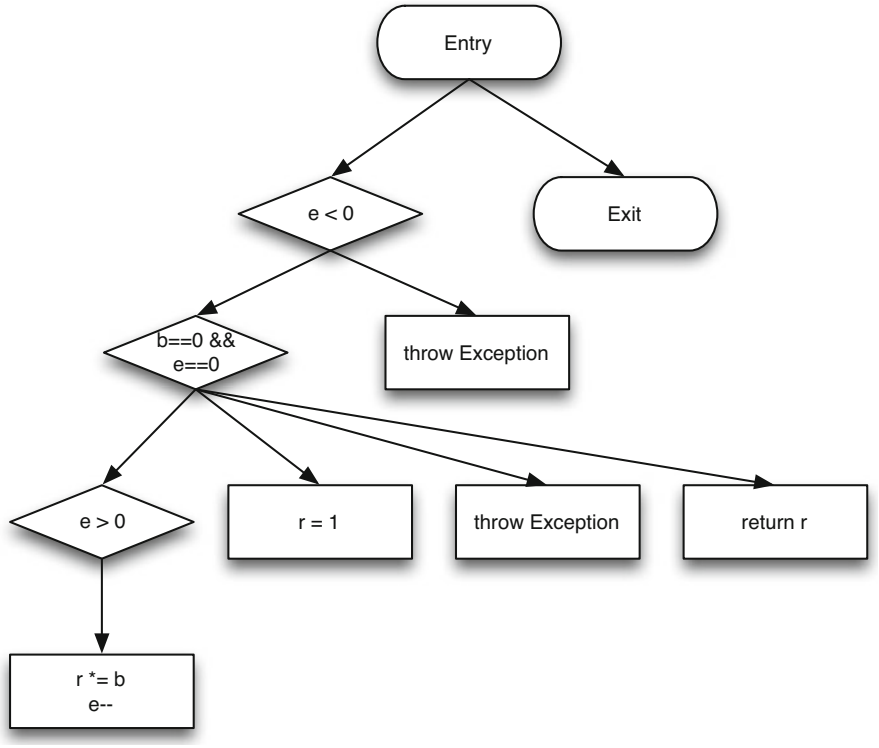

Fig. 15 Control dependence graph of the `power` function

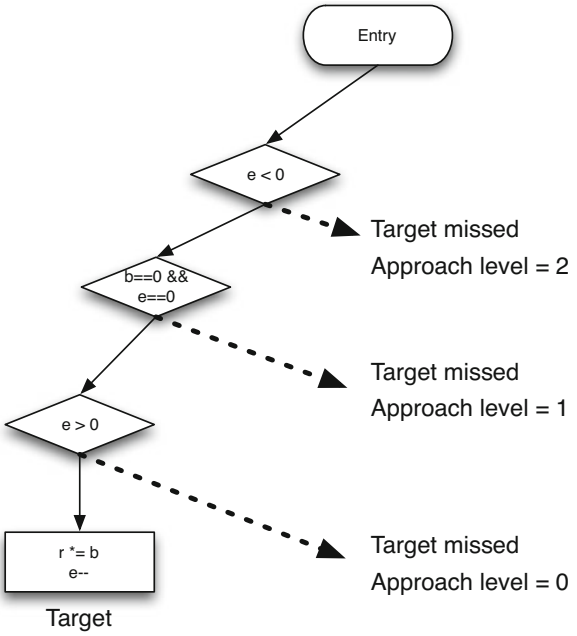

Fig. 16 Approach level illustrated for the target statements in line 8/9. Each control dependency away from the target node increases the approach level by 1

example program: Besides the entry node, our target statement has three control dependencies, as highlighted in Fig. 16. If we, for example, execute inputs $(0, -1)$, then the first if condition would evaluate to true, thus diverging the execution from reaching the target. Looking at the control dependence graph, we see that there are two control dependencies between the target node and this branch, and so this test would have an approach level of 2. If we would try input $(0, 0)$, then we would get one if condition further, thus reducing the approach level to 1. If we reach the loop header of the while loop, then the approach level is 0. Thus, the objective of the search is to minimize the approach level.

The second part of the standard fitness function is the branch distance, which was introduced by Korel (1990). The idea is to estimate for a given branching statement in the source code how far it is from evaluating to true or to false. For example, consider the if statement in line 2 of Fig. 1: `(if(e < 0))`. This condition evaluates to true if `e` is less than 0. The branch distance for making the condition true is per definition 0 if `e` is already less than 0. However, if `e` is greater or equal to 0, then the search needs guidance toward making `e` less than 0. This is achieved by making the branch distance greater, the larger `e` is; it also needs to be greater than 0 if `e` equals 0. Thus, the branch distance is calculated as `e + 1`. Conversely, the distance to making the if condition false is 0 if the condition is already false, and it is abs(`e`) if it is true, thus guiding the search toward making `e` greater or equal to 0. Similar rules are defined for all types of comparisons. For example, Table 3 lists the branch distance values for two of the if conditions of the `power` function. A further question is how to combine branch distances for conditions joined together using logical operators.

Table 3 Example branch distance values for two of the if conditions of the `power` function

Value of `e`	`if(e > 0)` True	`if(e > 0)` False	`if(e != 0)` True	`if(e != 0)` False
−100	0	101	0	100
−10	0	11	0	10
−1	0	2	0	1
0	0	1	1	0
1	1	0	0	1
10	10	0	0	10
100	100	0	0	100

For conjunctions, the branch distance is the sum of constituent branch distances, and for disjunctions it is the minimum branch distance of the constituent branch distances. A detailed overview of the different types of branch predicates and their distance functions is provided by Tracey et al. (1998).

The combination of approach level and branch distance was proposed by Wegener et al. (2001): the resulting fitness function consists of the approach level plus the branch distance for the control-dependent branch at which the execution diverged from reaching the target statement; both values are minimized. To ensure that the branch distance does not dominate the approach level, it is normalized in the range [0, 1], whereas the approach level is an integer number. For example, if we target line 8 in our example again and consider test input $(0, -10)$, then this will diverge at the first if condition, which means approach level 2. It is this branch in which we measure the branch distance, which is $abs(-10-0) = 10$. To normalize this value, we use, for example, the function provided by Arcuri (2013), which is simply $x/(x+1)$. Consequently, our overall fitness value is $2+10/(10+1) = 2.909$. Increasing the value of e to -9 changes the fitness value to $2+9/(9+1) = 2.9$ and so on. This standard fitness function—or indeed any other fitness function (Harman and Clark (2004) argue that many standard metrics used by software engineers can serve as fitness functions)—can be applied in almost any meta-heuristic search algorithm, once the representation of test data has been suitably encoded for the algorithm.

As a simple example, assume we want to apply hill climbing to the example `power` function (Fig. 1), in order to reach line 5 which throws an exception as both inputs are 0 (which, of course, would be trivially easy for a human tester). The representation is simply a tuple (b, e) for the two input parameters, and we need to define a neighborhood, for which we can use the Moore neighborhood (i.e., $(b-1, e-1)$, $(b-1, e)$, $(b, e-1)$, ...). Figure 17 depicts the resulting search space: There is a global optimum at (0, 0); for values of $e < 0$ the search is guided by the branch distance for making e greater or equal to 0. For values of e greater or equal than 0, the approach level is one less, and the search is guided by the combination of the branch distances of e and b.

Hill climbing starts by selecting random values for b and e and calculating the fitness value of that input pair. Calculating the fitness value in this case means

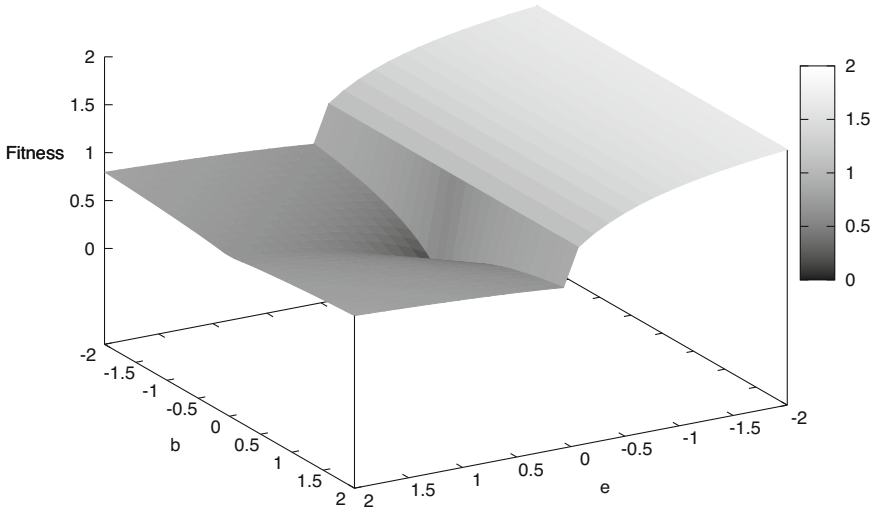

Fig. 17 Search landscape for reaching line 5 in the `power` function (assuming floating point numbers for illustrative purposes), based on approach level and branch distance. The global optimum (minimum) is at (0, 0)

executing the `power` function with these inputs. Instrumentation is required in order to calculate the approach level and branch distances from which to calculate the overall fitness value. Then, the fitness value of the neighbors of that point are calculated, and the neighbor with the best fitness value is chosen as next position. There, the algorithm again looks at all neighbors and so keeps moving until an optimal solution has been found.

Korel (1990) describes a variant of hill-climbing search known as the alternating variable method: For a program with a list of input parameters, the search is applied for each parameter individually in turn. To increase efficiency, the search on a parameter does not consist of the basic hill climbing of a neighborhood as just described; instead, "probe" movements are first applied to determine in which direction the search should head. For example, our `power` function has two parameters (b, e). AVM would start with a random assignment of values to these parameters, for example, $(-42, 51)$. First, AVM would apply probe movements of $b + 1$ and $b - 1$ on the first parameter, b. That is, it would run $(-41, 51)$ and $(-43, 51)$. If neither of the two movements leads to a fitness improvement, then the next parameter in the list is considered (e). If one of the probes on b leads to an improvement, then the search is applied in the direction of the probe using an accelerated hill climber, where the step size is increased with a factor of 2 with each step. Thus, if $(-41, 51)$ improved fitness, the next step will be a movement by $+2$ $(-39, 51)$, then $+4$ $(-35, 51)$, and so on, until the fitness does no longer improve. If this happens, the algorithm starts with probes on the same parameter again. If no more improvements can be achieved, it moves on to the next parameter. If

neither parameter can be improved, then AVM is restarted with new random values. The chaining approach (Ferguson and Korel 1996) improves this by considering sequences of events rather than individual inputs.

The use of genetic algorithms for test data generation was initially proposed by Xanthakis et al. (1992), and later used by many other authors. Genetic algorithms are population-based, which means that rather than optimizing a single individual, there is a whole population of candidate solutions that undergo modifications simulating those in natural evolution. For example, the fitter an individual, the higher the probability of being selected for recombination with another one, etc. Early work such as that by Xanthakis et al. (1992) and Jones et al. (1996) encoded test data as binary sequences, as is common with genetic algorithms. In the case of our `power` function with two inputs, if we assume that each of the input parameters is represented as an 8 bit number, then a chromosome would be a sequence of 16 bits. For example, the input (10, 3) would be encoded as `00001010 00000011`. Based on the fitness function as described above, a population of such individuals would be evolved by first selecting parent individuals based on their fitness; the fitter an individual, the more likely it is selected for reproduction. For example, tournament selection consists of selecting a set (e.g., 5) of individuals randomly and then choosing the fittest of that set. Two chosen parent individuals are used to produce offspring with the search operators of *mutation* and *crossover*, which are applied with a certain probability. For example, consider two individuals (10, 3) and (24, 9). Let us assume we select our crossover point right in the middle between, then two offspring would be produced by combining the first half of the first test with the second half of the second test, and vice versa:

```
00001010 00000011 → 00001010 00001001
00011000 00001001 → 00011000 00000011
```

Mutation would consist of flipping individual bits in a chromosome, for example:

```
00001010 00000011 → 00001010 00100011
```

Later work using genetic algorithms, for example, by Michael et al. (2001), Pargas et al. (1999), or Wegener et al. (2001), uses real value encoding. In this case, a chromosome would not be a sequence of binary numbers but simply a vector of real numbers (e.g., (10, 3)). More recently, Tonella (2004) demonstrated the use of genetic algorithms to generate unit tests for object-oriented classes, where each test is represented as a sequence of calls that construct and modify objects, which has spawned significant interest in automated unit test generation. The benefit of search-based testing clearly shows in this type of work: The underlying fitness function to generate a test that covers a specific goal is still the same as described above, and a genetic algorithm is also still used; the only difference lies in the changed representation, which is now sequences of constructor and method calls on objects, together with appropriate mutation and crossover operators. Fraser and Arcuri (2013) have extended this work to show generating sets of tests leads to higher

code coverage than generating individual tests, and the resulting EvoSuite[6] tool is recently seeing high popularity. Again a genetic algorithm is used; the representation here is no longer individual sequences of calls but sets of sequences of calls, and the fitness function is an adaptation of the branch distance. It is even possible to apply exactly the same search algorithm and fitness function with a test representation of user interactions, as done by Gross et al. (2012)—a clear demonstration of the flexibility of search-based testing.

Symbolic Execution

Originally proposed by King (1976) and neglected for a long time period due to its inherent scalability limitations, symbolic execution has gained more relevance in the last decade, thanks to the increased availability of computational power and advanced decision procedures. White box test data generation stands out as one of the most studied applications of symbolic execution. Among the most relevant early work, DeMillo and Offutt (1991) combined symbolic execution and mutation analysis to derive fault-revealing test data with the Mothra testing system, and Gotlieb et al. (1998) developed a more efficient constraint-based test generation approach with early pruning of unfeasible paths. More recently, many new techniques and tools have been developed which rely on symbolic execution for test generation, some of them with industrial and commercial impact, as surveyed by Cadar et al. (2011).

Symbolic execution consists in executing a program under test using symbolic variables instead of concrete values as input arguments. At any point along the execution, a symbolic state consists of symbolic values/expressions for variables, a path condition, and a program counter. The path condition represents the constraints that must hold for the execution to follow the current path. When a conditional statement is reached (e.g., if or loop conditions), the symbolic execution forks to explore the two possible outcomes of the conditional, each with a different updated path condition. Every time a new constraint is added to it, the satisfiability of the path condition is checked—normally using off-the-shelf constraint solvers. If the new path condition is satisfiable, the symbolic execution continues along that path, otherwise the path is ignored, since it has been proved unsatisfiable.

Figure 18 shows how symbolic execution works on our running example. The root node represents the starting point of the executions, with the input arguments being bound to unconstraint variables and an empty path condition. When reaching the first if statement, the symbolic execution evaluates the if condition to both *true* and *false*. In the first case, the constraint $E < 0$ is added to the path condition, terminating the program with an exception due to a negative exponent passed as input (terminating nodes are denoted with gray boxes). The second case corresponds to a valid exponent passed as argument (zero or positive). The second if condition

[6]EvoSuite—automatic test suite generation for Java. http://evosuite.org. Accessed March 2017.

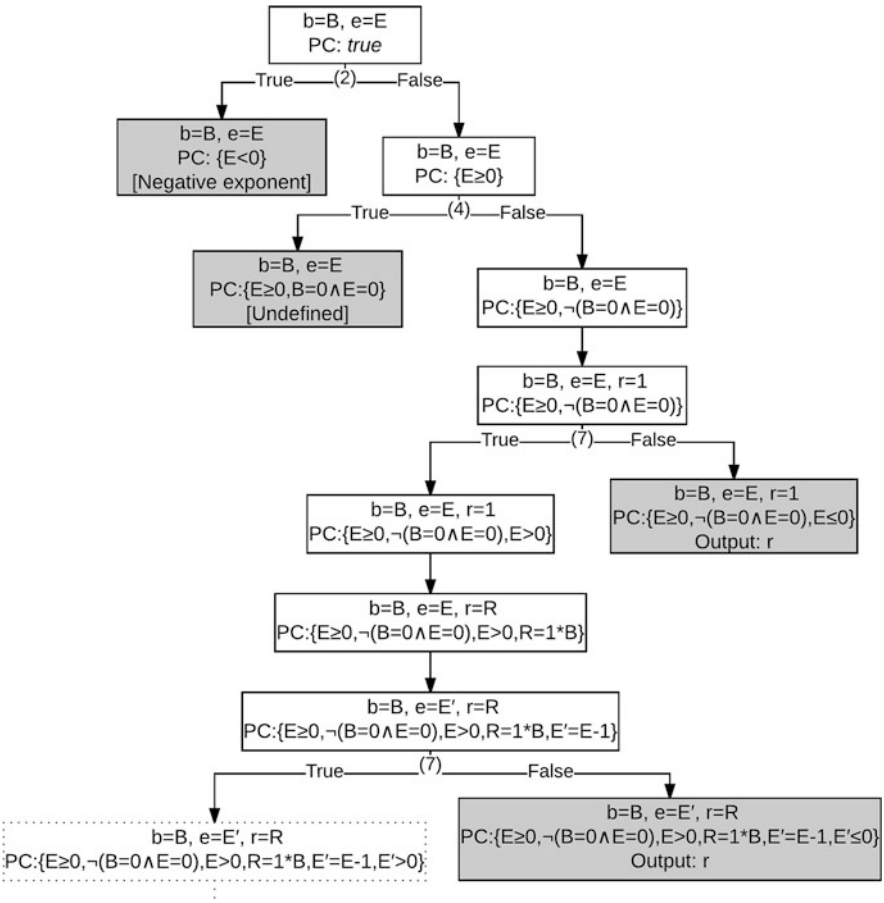

Fig. 18 Symbolic execution of the `power` example

is handled similarly to the first one; notice how the path conditions are updated with complementary constraints. Loop conditions are evaluated as normal conditionals. However, when the loop condition depends on an input argument, as in our example, a bound on the exploration depth must be imposed to ensure termination. Figure 18 covers the cases where the loop is not entered at all (the exponent is zero) and the case where the loop is entered and its body is executed only once (exponent is one) and leaves paths with greater exponent values unexplored (dotted leaf node in the tree).

Dynamic Symbolic Execution

The frequently huge search space and vast complexity of the constraints to be handled in symbolic execution hamper the scalability and practicality of testing approaches based on symbolic execution. Moreover, due to its systematic nature, symbolic execution is oblivious to the feasibility of an execution path at runtime; hence it is prone to produce false positives. To alleviate these limitations, Godefroid et al. (2005) and Sen et al. (2005) proposed dynamic symbolic execution as a combination of *conc*rete and symb*olic* execution (hence also dubbed *concolic* execution).

Dynamic symbolic execution keeps track of a concrete state alongside a symbolic state. It starts by executing the system under test with randomly generated concrete input values. The path condition of this concrete execution is collected, and its constraints are systematically negated in order to explore the vicinity of the concrete execution. This alternative has been shown to be more scalable than traditional symbolic execution alone because it allows a deeper exploration of the system and can cope with otherwise problematic features like complex nonlinear constraints or calls to libraries whose code is unavailable for symbolic execution.

To illustrate a case in which intertwining concrete and symbolic execution overcomes an inherent limitation of traditional symbolic execution, let us modify slightly our running example by adding a conditional statement guarding an error state (Fig. 19). Let us assume that the execution of `mist(b, e)` relies on either a nonlinear constraint over `e` or a system call. Due to the current limitations in constraint solving or to the impossibility of symbolically executing native code, symbolic execution would be unable to enter the `then` branch of the conditional; hence it would fail to produce a test revealing the error. In contrast, dynamic symbolic execution would use the concrete values of `b` and `a` to invoke `mist` and execute the error-revealing branch.

Multiple tools have been proposed which implement this dynamic symbolic execution approach. Among the most successful ones, DART (Godefroid et al. 2005) has been shown to detect several programming errors (crashes, assertion violations and nontermination) related to arithmetic constraints in C. CUTE (Sen et al.

```
1  int power(int b, int e){
2     if (e < 0)
3        throw new Exception("Negative exponent");
4     if ((b == 0) && (e == 0))
5        throw new Exception("Undefined");
6     if (b != mist(b, e))
7        throw new Exception("Error");
8     int r = 1;
9     while (e > 0){
10       r = r * b;
11       e = e - 1;
12    }
13    return r;
14 }
```

Fig. 19 Modified running example

2005) also supports more complex C features (e.g., pointers and data structures), KLEE (Cadar et al. 2008) can significantly improve coverage of developer-written tests, and SAGE (Godefroid et al. 2008b) is used daily to test Microsoft software products such as Windows and Office.

3.3.3 Concurrency Testing Techniques

While many test generation approaches target sequential programs and standard structural coverage criteria, there is also a need to generate tests for concurrent programs. Arguably, the need for automation is even higher in this case, since manually testing concurrency aspects is particularly tedious. A very successful approach to testing concurrency aspects of programs lies in using existing tests and then manipulating the thread interleavings while executing the test. For example, Contest (Edelstein et al. 2002) repeatedly executes tests and adds random delays at synchronization points. The CHESS (Musuvathi et al. 2008) tool systematically explores thread schedules with a stateless model checking approach, and can reliably find concurrency issues. Generating tests dedicated to explore concurrency aspects is a more recent trend; the Ballerina (Nistor et al. 2012) tool uses random test generation to explore interleavings, and ConSuite (Steenbuck and Fraser 2013) uses search-based test generation to generate test suites targeting concurrency coverage.

3.4 Executing Tests

In any industrial development scenario, as programs grow in size, so do the test suites accompanying them. Software companies nowadays rely on automated frameworks for managing their build processes and executing their test suites (e.g., Jenkins[7] and Maven Surefire[8]). Moreover, continuous integration frameworks enable development teams to ensure functionality and avoid regression defects in their codebases by running existing tests every time a change has been made. Efficient test execution soon becomes a challenge. How to control the size of a test suite? How to ensure that changes to the codebase do not break existing functionality? How to decide which tests to run after each change? How to extend existing test suites? This practice is known as *regression testing*. In this section, we describe the most outstanding optimization techniques related to the execution of test suites to detect bugs as software evolves. As in previous sections, an overview is presented in Fig. 20, and we can also here point to a comprehensive survey on the topic, this time by Yoo and Harman (2012), for more detailed information.

[7] Jenkins— open-source automation server. https://jenkins.io. Accessed March 2017.

[8] Surefire—A test framework project. http://maven.apache.org/surefire/. Accessed March 2017.

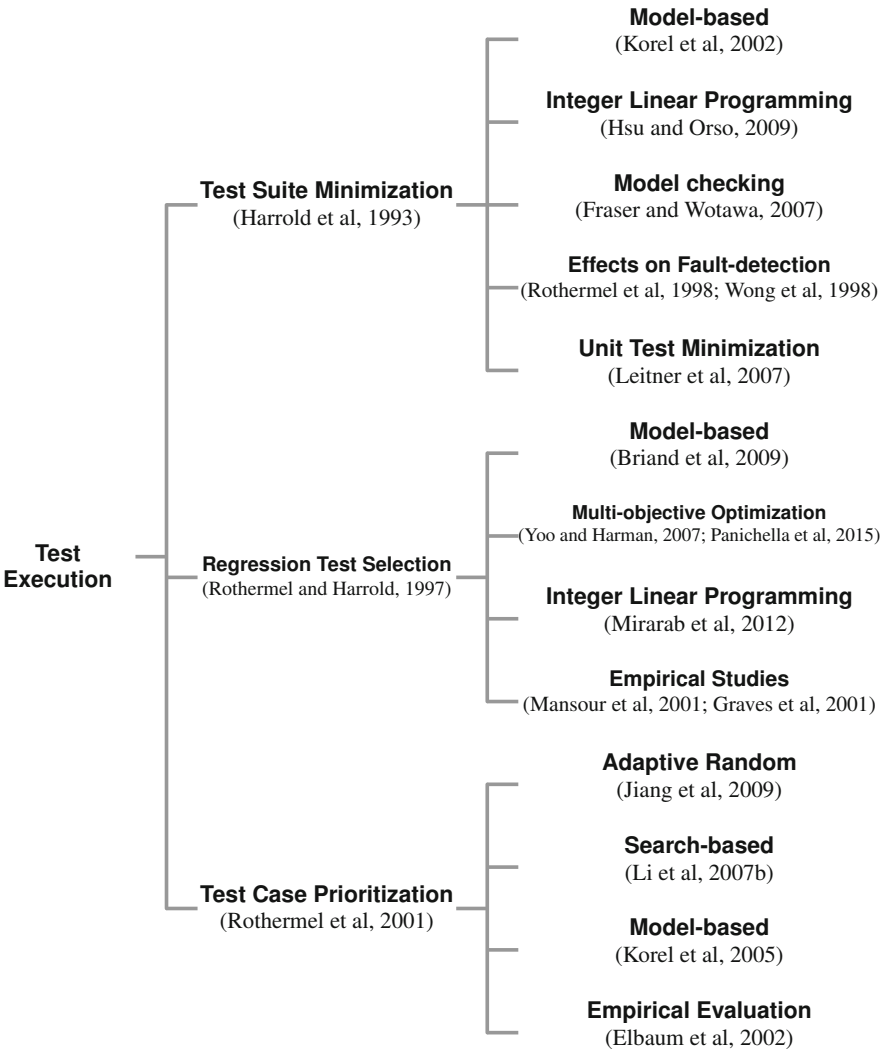

Fig. 20 Executing tests: genealogy tree

3.4.1 Automated Test Execution

While there are many obvious benefits to automating test execution (e.g., reusability, reduced regression testing time), there is the question how this automation is achieved. A widespread technique with a long tradition is the family of capture and replay tools. Besides many commercial tools, an example of an open-source capture and replay framework is Selenium (see Sect. 2). In these tools, the tester interacts with the program under test, and while doing so the interactions are recorded. Later, these recorded interactions can be replayed, and an automated

system performs the same actions again. An important benefit of this approach is that the tester needs to have no programming knowledge but simply needs to be able to use the program under test. However, capture and replay tests tend to be *fragile* (Hammoudi et al. 2016): if the position of a button is changed, or if the locator of an element on a web page is changed, then all tests interacting with that element fail and need to be repaired or regenerated.

A more flexible approach thus lies in making the tests editable, which can be done in different ways. In keyword-driven testing (Faught 2004), testers create automated tests in a table format by using keywords specific for the application under test, for which developers provide implementations. Many testing frameworks provide custom domain-specific scripting languages; for example, tools like Capybara[9] allow testers to write automated functional tests that interact with the program under test using a choice of back ends. However, it is also very common to use regular scripting languages (e.g., Python) to automate tests.

In 1998, Kent Beck introduced the SUnit framework for automated tests for the Smalltalk language (Beck 1999). Similar frameworks have since then been produced for almost any programming language, with Java's JUnit, originally implemented by Kent Beck and Erich Gamma, may be being the most well-known example. The current version of JUnit is JUnit 4, and we have already seen some examples of JUnit tests in this chapter. For example, the following test calls the `power` function with parameters 10 and 10 and then uses a JUnit assertion (`assertEquals`) to check if the actual result matches the expected result, 100:

```
1 @Test
2 public void testPowerOf10() {
3   int result = power(10, 10);
4   assertEquals(100, result);
5 }
```

While the name *xUnit* suggests that these frameworks target unit testing, they are not only useful for unit testing. It is similarly possible to write automated integration tests, and by using testing frameworks such as Selenium, it is also possible to write automated system tests using JUnit.

3.4.2 Test Suite Minimization

As software evolves, managing the size of existing test suites becomes a necessary task: existing tests can become obsolete (i.e., they test behavior that is no longer implemented) or redundant (i.e., they do not contribute to increasing the overall effectiveness of the test suite). Maintaining test suites is seldom the priority of any development team, and hence it is often neglected. Developers normally add new tests when a change is made or new behavior is implemented. However, developers seldom check whether existing tests already cover the modified code or whether

[9]Capybara—acceptance test framework for web applications. http://teamcapybara.github.io/capybara/. Accessed March 2017.

tests have become unnecessary after a change. Harrold et al. (1993) developed a methodology to control the size of existing test suites by automatically detecting and removing obsolete and redundant test cases.

Given a system under test, a set of tests, and a set of test requirements (e.g., line coverage), the test suite minimization problem consists in constructing the smallest possible test suite such that the effectiveness of this new test suite—the number of fulfilled test requirements—is the same as the effectiveness of the original suite. Because this is an NP-complete problem, Harrold et al. (1993) developed a heuristic to find a representative solution and demonstrated that the size of a test suite can be significantly reduced even for small programs.

The algorithm proposed by Harrold et al. (1993) uses knowledge about the relation between tests and testing requirements to minimize a test suite: the set of requirements each test meets must be known. Let us describe their algorithm through an example. Table 4 shows a list of seven test cases for the `power` function. We assume that our test requirements are based on statement coverage; to make the test requirements easy to interpret, we list them with their line numbers. The table lists for each test which of the test requirements (statements/lines) it covers. If a test t_i meets requirement r_j, an x is shown at cell (i, j). The algorithm starts with an empty test suite T. First, requirements that are met by one single test are considered, and those tests are added to the resulting suite. In the example, the statements on lines 3 and 5 are each met by only one test (t_1 and t_2, respectively); hence $T = T \cup \{t_1, t_2\}$ and lines 2, 3, and 5 are marked as covered. Then, the process iteratively looks at test requirements not met yet which are met by two tests, then those met by three tests, and so on. In our example, there are no requirements covered by two tests, but the lines 8 and 9 are covered by three tests (t_4, t_5, and t_7). If there are several candidates to choose from, ties are resolved by considering the test requirements with the next cardinality covered by the candidate tests. However, since our tests cover exactly the same, this tie can only be broken by randomly choosing one, for example, t_4, making $T = \{t_1, t_2, t_4\}$. Since all test requirements are now met by T, minimization is complete and T is the final minimized test suite.

Following the seminal work by Harrold et al. (1993), the effects of test suite minimization on the fault detection of the minimized test suites was studied by Rothermel et al. (1998) and Wong et al. (1998). More recent approaches

Table 4 Test cases and testing requirements

Test	Input	Testing requirement (line)								
		2	3	4	5	6	7	8	9	11
t_1	(0, −10)	x	x							
t_2	(0, 0)	x		x	x					
t_3	(10, 0)	x		x		x	x			x
t_4	(1, 1)	x		x		x	x	x	x	x
t_5	(2, 2)	x		x		x	x	x	x	x
t_6	(−10, 0)	x		x		x	x			x
t_7	(−2, 2)	x		x		x	x	x	x	x

have explored alternative approaches beyond the relation of tests with coverage requirements. Korel et al. (2002) used a dependence analysis on finite state machine models to guide test suite reduction. Hsu and Orso (2009) developed a framework to encode multiple criteria (e.g., coverage, cost, fault detection) into an integer linear programming (ILP) problem and used modern ILP solvers to derive optimal solutions. Fraser and Wotawa (2007) minimized test suites by identifying and removing redundancy among automatically generated test cases. A different flavor of test minimization was explored by Leitner et al. (2007), who used program slicing and delta debugging to minimize the sequence of failure-inducing method calls in the test code (as opposed to removing tests from the test suite).

3.4.3 Regression Test Selection

While test suite minimization allows to reduce the number of tests to be executed, removing redundant tests altogether entails a danger in the long run. A test might be removed solely on the basis that another test that meets the same testing requirement already exists. However, it is not hard to imagine that the removed test may have been useful to reveal future bugs. Selecting the best tests for regression testing comes as an alternative to test suite minimization: it is not a problem to keep large test suites in the codebase, as long as the set of tests to be run whenever a change has been made is carefully decided.

Regression test selection comprises two subproblems: identification of affected code and selection of tests to run. Integer programming, symbolic execution, dataflow analysis, and path analysis are examples of general techniques that have been successfully applied to the regression test selection problem. Rothermel and Harrold (1997), for instance, formulated a safe and efficient technique based on the traversal of the control flow graphs (CFGs) of the original and modified versions of the system which has served as seminal work in this area. The main idea is to identify the parts of the program affected by a change by performing a parallel traversal of both CFGs. At each step in the traversal, the current nodes from both graphs are compared for lexicographical equivalence. If any difference is found, the tests traversing a path from the entry node to such modification node are collected and included in the selection set (these tests are referred to as *modification-traversing* tests).

To illustrate this algorithm, consider the CFG in Fig. 21. Consider the existing test cases as well, whose traces are shown in Table 5 as sequences of edges between two consecutive nodes in the CFG of the original program (Fig. 4). Observe how the modified CFG differs from the original one: the statement in line 9 (which controls the loop termination) has been mistakenly removed, and the statement in line 5 has been changed (assume the specification changed and a different exception is now thrown). The algorithm starts by initializing the set of selected tests to empty and comparing both entry nodes. The algorithm proceeds by recursively traversing both CFGs at the same time, without finding any difference after traversing the path (Entry,L2), (L2,L4), (L4,L6), (L6,L7), (L7,L8). However, when comparing L8

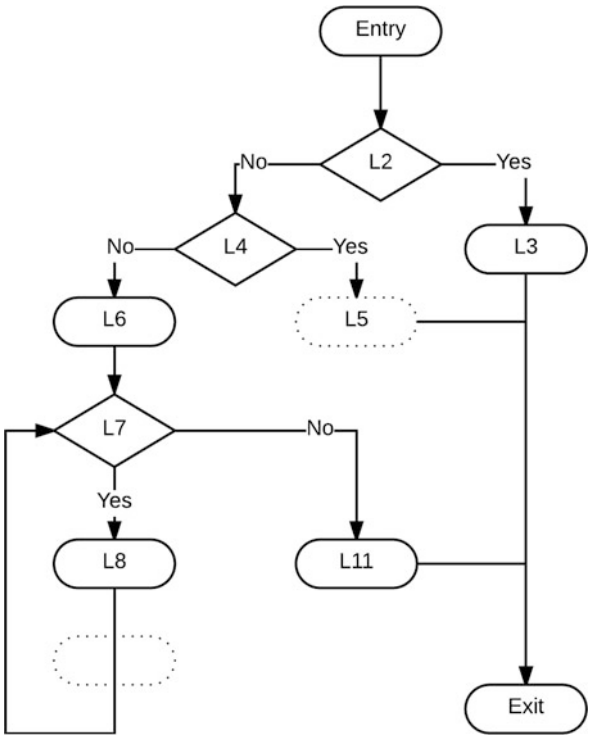

Fig. 21 Control flow graph of the modified `power` function

between the two CFGs, a difference is observed in the successor node of L8 (L9 in the original version, L7 in the modified version). As a result, tests traversing the edge (L8, L9), i.e., $\{t_4\}$, are added to the selection set. Similarly, when exploring the true branch of node labeled L4, a difference is observed in the successor L5, which further triggers the inclusion of a test that traverses the edge (L4,L5), namely, t_2. The resulting selected test suite is then $\{t_2, t_4\}$.

The above described algorithm works at the intraprocedural level, i.e., within the frontiers of a single procedure, function, or method. In the same seminal work, Rothermel and Harrold (1997) also extended the algorithm for interprocedural testing, where subsystem and system tests tend to be larger than tests for single procedures. This extended test selection algorithm performs CFG traversals similarly to the intraprocedural algorithm but additionally keeps track of *modification-traversing* information across procedure calls. A key idea for this to work is that the traversal of the CFG of each method in the system is conditioned to the nonexistence of a previous difference between the two CFGs under comparison.

Mansour et al. (2001) and Graves et al. (2001) explored the properties of different test selection techniques and their relative costs and benefits for regression testing. Alternative methods to inform test selection have been studied in recent years. Yoo and Harman (2007) and later Panichella et al. (2015) formulated regression

Table 5 Test trace for regression test selection

Test	Test trace
t_1	(Entry, L2), (L2, L3), (L3, Exit)
t_2	(Entry, L2), (L2, L4), (L4,L5), (L5, Exit)
t_3	(Entry, L2). (L2, L4), (L4, L6), (L6, L7), (L7, L11), (L11, Exit)
t_4	(Entry, L2), (L2, L4), (L4, L6), (L6, L7), (L7, L8), (L8, L9), (L9, L7), (L7,L11), (L11, Exit)

test selection as a multi-objective search optimization problem. Analogously to the work of Hsu and Orso (2009) for test suite minimization, Mirarab et al. (2012) formulated test selection as a multi-criteria integer linear programming problem. Model-based approaches have also been explored for test selection: Briand et al. (2009), for instance, presented an methodology to guide regression test selection by the changes made in UML design models.

3.4.4 Test Case Prioritization

Complementary to test suite minimization and selection, test case prioritization aims at finding the best order in which a set of tests must be run in an attempt to meet the test requirements as early as possible. Rothermel et al. (2001) investigated several test case prioritization techniques with the aim to increase the fault-revealing ability of a test suite early in the testing as soon as possible. However, because fault-detection rates are normally not available until all tests have run, other metrics must be used as surrogates to drive test prioritization. In their work, Rothermel et al. (2001) formulated test requirements as the total code coverage achieved by the set of tests (branch and statement coverage), their coverage of previously not covered code (again, using branch and statement coverage), or their ability to reveal faults in the code (using mutation testing). In the case of structural coverage surrogates, a greedy algorithm is used to prioritize tests that achieve higher coverage. In the case of fault-revealing ability, the mutation score of a test, calculated as the ratio of mutants executed and killed by the test over the total number of non-equivalent mutants, determines the priority of each test.

A common way to illustrate prioritization techniques is to show how the order of execution influences the rate of detection for a given set of faults. Let's assume we generate a set of faults (mutants) using the Major[10] mutation tool (Just et al. 2011) for our `power` function example, which gives us a set of 31 mutants. Consider the test suite shown in Table 4, which kills 24 of these mutants. Depending on the order in which the tests are executed, these mutants will be killed sooner or later. Figure 22 illustrates the rate at which the mutants are found. For example, if we execute the tests in a random order, for example, t_5, t_3, t_4, t_6, t_7, t_2, and t_1, then the first test kills nine mutants, the second one three additional mutants, and so on. If we prioritize

[10]The Major mutation framework. http://mutation-testing.org/. Accessed March 2017.

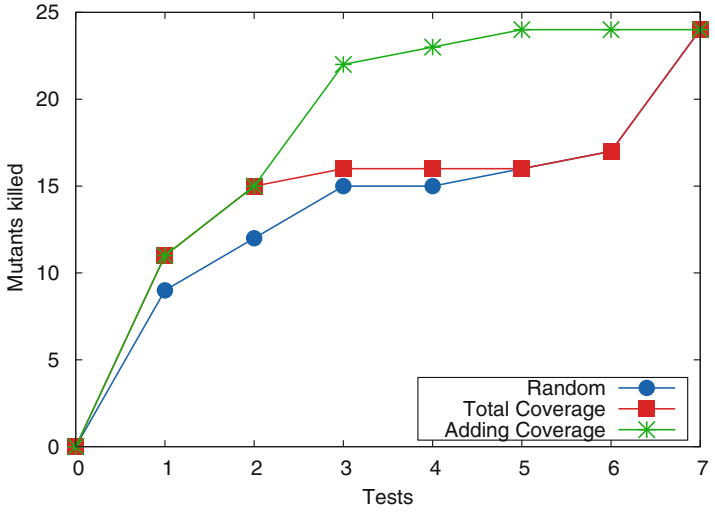

Fig. 22 Mutants found with test cases ordered using different prioritization techniques

based on the total coverage achieved by each tests, then the tests would be executed in the order t_4, t_5, t_7, t_3, t_6, t_2, t_1, resulting in 11 mutants killed by the first test, then another 4, and so on. If we prioritize tests by the additional coverage they provide over previous tests, then we would execute them, for example, in order t_4, t_2, t_1, t_5, t_7, t_3, t_6, such that by the time of the third test already 22 mutants are killed and by the fifth test all mutants are killed.

The effect of the test case order can be quantified as the weighted average of the percentage of faults detected, known as the APFD metric, introduced by Rothermel et al. (2001). In essence, the APFD metric calculates the area under the curves shown in Fig. 22. The average percentage of faults detected by the random order in Fig. 22 is 0.64, whereas total coverage prioritization increases this to 0.68, and the additive coverage prioritization increases this further to 0.85, confirming that this is the best order in which to execute the tests.

Similarly to test suite minimization and regression test selection, research on test case prioritization has evolved to empirically validate the benefits of prioritizing tests (e.g., Elbaum et al. 2002) and to explore alternative ways to inform the prioritization. Korel et al. (2005) presented several methods to prioritize tests based on information of changes to state-based models of a system. Li et al. (2007) showed that search-based algorithms could be applied for test prioritization, and Jiang et al. (2009) showed that adaptive random testing was as effective as and more efficient than existing coverage-based prioritization techniques.

3.4.5 Testing Embedded Software

Embedded systems define a wide range of different systems where the software interacts with the real world. An embedded system typically controls some specific hardware, receives input signals through sensors, and responds with outputs sent through actors; these outputs then somehow manipulate the environment. Embedded software is often targeted for specific hardware with limited resources. The restricted hardware and the interactions with the environment through sensors and actuators pose specific challenges for testing these systems: Tests need to somehow provide an environment that creates the necessary sensory input data, and updates the environment based on the system's actions. To achieve this, testing of embedded systems is usually performed at different levels (Broekman and Notenboom 2003). Since embedded systems are often developed using model-driven approaches (Liggesmeyer and Trapp 2009), the first level of testing is known as *model-in-the-loop*. At this level, the model of the program under test is embedded in a model of the environment and then tested. At the *software-in-the-loop*, the compiled program is tested in the simulated environment but still on the development hardware. At the *processor-in-the-loop*, the software is executed on the target hardware but still interacts with the simulated environment. Finally, at the *hardware-in-the-loop* level, the actual system with real sensors and actuators is tested on a test-bench with a simulated hardware environment.

A specific aspect of testing embedded systems is that timing in these systems is often critical. That is, these *real-time* systems must often guarantee a response to an input signal within specified time constraints. There are various approaches to testing the violation of timing constraints (Clarke and Lee 1995); in particular, the use of timed automata (Nielsen and Skou 2001) is an approach that is popular in the research domain to formalize timing constraints in systems.

The use of formal specifications and conformance testing (Krichen and Tripakis 2009) is relatively common in the domain of real-time embedded systems, possibly because these systems are often safety critical. That is, errors in such programs can cause physical harm to users, such that common international standards (e.g., IEC61508 Commission et al. 1999) require rigorous software testing methods. For example, the DO-178B standard (Johnson et al. 1998), which is used by the US Federal Aviation Administration for safety critical software in aircrafts, requires that software tests need to be adequate according to the MC/DC criterion (Sect. 3.1.2).

3.5 Current Trends in Testing Research

This chapter has so far focused on seminal work in testing, but not all of these aspects of testing are active areas of research. In order to provide an intuition about current trends, we now take a closer look at research activities in the last 10 years. We collected the titles of all papers published at mainstream software testing research conferences (IEEE International Conference on Software Testing,

Fig. 23 Software testing research published between 2007 and 2017 at the main software engineering conferences

Verification, and Validation, ACM International Symposium on Software Testing and Analysis), and all papers related to testing published at mainstream software engineering conferences (IEEE/ACM International Conference on Software Engineering, SIGSOFT Symposium on the Foundations of Software Engineering, IEEE/ACM Conference on Automated Software Engineering). Figure 23 summarizes the paper titles in a "word cloud," in which several trends become apparent.

3.5.1 Model-Based Testing

The most frequent term in Fig. 23 is "automata," and this is related to generally frequent terms related to model-based testing. A commonly made assumption by researchers is that testing techniques have some sort of formal specification or model as a starting point. The reason this is popular in research is clear: automation requires some form of executable representation of the intended behavior. Given a formal specification, for example, in terms of an automaton, it is possible to derive sequences of inputs that fully cover the behavior, it is possible to predict what their expected response is, and it is typically possible to cover the intended behavior in fewer tests based on an abstract specification compared to the concrete implementation. A particularly attractive aspect of automata-based testing is the prospect of guarantees that can be made by testing (Lee and Yannakakis 1996; Tretmans 1996). For example, under certain assumptions it is possible to prove properties of the system under test. There is also a wealth of different techniques and tools originating in formal methods that can be exploited to support testing tasks. For example, although the aim of model checking tools is to prove properties, their ability to provide counterexamples for violated properties has been used as a means to generate tests. As a basic example, each coverage goal can be represented

as a property that states the goal cannot be achieved, and then the counterexample to such a property is, essentially, a test that does cover the goal (Fraser et al. 2009).

3.5.2 Automated Test Generation

A trend clearly reflected in Fig. 23 is research on automated test generation. The ongoing and ever more severe problems in practice caused by software reinforce the need for thorough software testing, but since testing is a laborious and error-prone activity, researchers seek to automate as much of the testing process as possible. While some aspects, like test execution, are commonly automated in practice, the task of deriving new tests usually is not automated. Therefore, this is a central problem researchers are aiming to address. There is an overlap between this topic and model-based testing; however, more recently techniques to generate tests without explicit specifications (e.g., based on source code, or inferred models) are becoming more popular.

Research on automated test generation is influenced by the different application areas and types of programs. Test generation is most common for system and unit testing, but less common for integration testing. Web and mobile applications (Di Lucca and Fasolino 2006; Choudhary et al. 2015) are new application areas that have seen a surge of new publications in recent years.

Very often, when exploring a new testing domain, the first approach is to use random test generation (e.g., Pacheco et al. 2007), and this tends to work surprisingly well. Since the original coining of the term "search-based software engineering" (Harman and Jones 2001), this has now become a mainstream approach to test generation, and is seeing many applications. A further recent trend in test generation is the use of symbolic methods: While for a long time they were neglected because the power of constraint solvers was deemed insufficient for test generation, the vast improvements made on SAT solvers in recent years have caused a proliferation of test generation approaches based on symbolic techniques, such as dynamic symbolic execution (Godefroid et al. 2005; Sen et al. 2005).

Often, in a newly explored application area of automated test generation, the initial objective is to find program crashes, because a crashing program can always be assumed to be a problem, without knowing any further details about the intended behavior. However, once test generation techniques mature, the *test oracle problem* (Barr et al. 2015) becomes more prominent: Given automatically generated tests, how do we decide which of these tests reveal bugs?

3.5.3 Test Analysis

Terms related to test analysis are common in Fig. 23. On one hand, code coverage is a generally accepted default objective of test generation. On the other hand, the suitability of code coverage as a measurement of fault-detection effectiveness has been the focus of discussions more recently (Inozemtseva and Holmes 2014). This

is why, in the research community, mutation testing has risen in popularity (Jia and Harman 2009). While mutation testing still suffers from scalability issues, there has been substantial progress on making mutation testing applicable in practice in recent years.

3.6 Current Trends in Software Testing Practice

3.6.1 Developers Are Testers

Over the last 10 years, agile software development has transformed the role of quality assurance. Where previously quality assurance was usually done separately, this is now integrated into the software development process. Developers now *own* software quality and are therefore in charge of testing their code early in the process. The largest software companies are leading this testing revolution by redefining their development and testing engineering roles. Developers are using unit testing to drive the development of new features (test-driven development (Beck 2003)). While pure-testing roles still exist, their work has evolved from scripted testing to exploratory testing (Kaner 2006), with emphasis on understanding and verifying requirements, interfacing with business stakeholders, and discovering problems at the system or application level.

Test-driven development (TDD), introduced in the late 1990s, is a central approach in this large-scale testing paradigm shift. In TDD, developers apply short development cycles consisting of three phases. First, they write unit tests for unimplemented functionality. Then they write the minimal amount of code necessary to make these tests run correctly (i.e., pass). Finally, they refactor and optimize their code, preserving quality. The benefits of using TDD are evident by its extensive usage in practice but also have been confirmed via empirical research (Shull et al. 2010). Acceptance test-driven development (ATDD) (Beck 2003) and behavior-driven development (BDD) (North 2006) are extensions of TDD that promote the application of agile principles into the development cycle (e.g., permanent communication with stakeholders).

As a consequence of this paradigm shift, the required skills for software engineers have changed, too. Today, testing skills are important for software developers in order to write automated tests and adopt and use testing frameworks during development. Testers, on the other hand, are required to have a more creative mindset, since their job goes beyond the mechanical reproduction of scripted tests.

3.6.2 Test Automation

The necessity for programming skills in testing is generally related to the trend toward test automation. Whereas in the past automated tests required expensive commercial tools, the availability of high-quality open-source testing frameworks

means that automation is now much more feasible and common. Often, the task of software testing involves not only writing and executing specific tests for a program, but also engineering elaborates automated testing solutions (Whittaker et al. 2012).

Continuous integration (CI) and automated testing are now standard for most software projects. CI frameworks, for example, Jenkins, TeamCity, Travis CI, and GitLab CI, provide comprehensive sets of features that allow development teams to keep track of build and test executions and overall software quality. At a finer-grained level, popular automation tools include Mockito[11] (mocking framework) and Hamcrest[12] (assertion framework) which are useful to improve the quality and expressiveness of executable tests. Automation tools also exist which are tailored for specific domains. For example, SeleniumHQ[13] automates browser interaction, JMeter[14] allows testing web performance, WireMock[15] provides mocking of http interactions, and Monkey[16] automates the stress testing of apps for the Android mobile operating system.

The general need for more automation in software testing is further reinforced by a recent proliferation of companies trying to make a business out of test automation. For example, the Capterra index of software solutions lists 120 test automation products at the time of this writing.[17] Many of these automation systems are still quite basic in what they automate; for example, many tools support record and replay or scripting of automated tests. It is to be expected that the degree of automation will continue increasing over time, including automation of aspects like test generation.

3.6.3 Trending Application Domains

The ongoing shift toward web and mobile applications in software engineering naturally has effects on testing. Ten out of 20 search topics reported as trending most over the last 10 years as reported by Google Trends[18] are related to web applications and JavaScript testing frameworks. The increasing use of mobile apps enforces stronger testing requirements, since the connected nature of mobile apps and their access to sensitive data means that security concerns are more pressing. The changed

[11] Mockito—tasty mocking framework for unit tests in Java. http://mockito.org. Accessed September 2017.

[12] Hamcrest—matchers that can be combined to create flexible expressions of intent. http://hamcrest.org. Accessed September 2017.

[13] SeleniumHQ—browser automation. http://www.seleniumhq.org. Accessed September 2017.

[14] Apache JMeter. http://jmeter.apache.org. Accessed September 2017.

[15] WireMock. http://wiremock.org. Accessed September 2017.

[16] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html. Accessed September 2017.

[17] Best Automated Testing Software—2017 Reviews of the Most Popular Systems http://www.capterra.com/automated-testing-software. Accessed September 2017.

[18] Google Trends. https://trends.google.co.uk. Accessed March 2017.

ecosystem of apps sold quickly via app stores also implies that insufficient testing can be the demise of an app if there is a more robust alternative. On the other hand, the concept of app stores makes it possible to quickly deploy fixes and updates, often making the end-user the tester. A recent trend to improve testing of apps lies in using crowds of testers in a crowd testing scenario. This idea was popularized by companies like uTest (now Applause[19]) and is now a flourishing business, with many other crowd testing companies appearing.

4 Future Challenges

4.1 Test Analysis

In this chapter, we have discussed various ways to determine and quantify the adequacy of a test set. Some techniques, such as basic code coverage criteria like statement or branch coverage, are commonly used in practice. Such source code metrics, however, can be misleading: Code structure is an unreliable proxy of underlying program behavior, and may not truly measure when a test set is good. As an example, code coverage on simple getter/setter methods contributes to a code coverage measurement, but is unlikely to help finding bugs. A further problem with code coverage is that it does not measure how well the exercised code is checked. Indeed there are empirical studies suggesting that code coverage is not strongly correlated with the effectiveness of a test suite (Inozemtseva and Holmes 2014).

In this chapter, we discussed mutation analysis as an alternative: Mutation analysis explicitly checks how well the tests verify expected behavior by seeding artificial faults. However, mutation testing has one crucial disadvantage—it is inherently slow. Even though more mutation tools have started to appear, these rarely make mutation analysis on large code bases feasible. While code coverage is clearly less computationally expensive, the instrumentation it requires still adds a computational overhead, and even that can prove problematic on very large code bases where tests are frequently executed, for example, as part of continuous integration.

Thus, there are at least two challenges that need to be addressed: We need better adequacy metrics, and we need to make them scale better to large systems. One avenue of research targeting the former problem goes back to an idea initially proposed by Weyuker (1983): If we can use machine learning to infer from a given set of tests a model that is an accurate representation of the program, then that set of tests is adequate. Recent work has shown that this is both feasible and provides a better proxy of program behavior than pure code coverage (Fraser and Walkinshaw 2015). To address the problem of scalability, a promising avenue of research lies

[19] Applause: Real-World Testing Solutions. https://applause.com/testing/. Accessed March 2017.

in dynamically adding and removing instrumentation, as exemplified by Tikir and Hollingsworth (2002).

4.2 Automated Testing

Although testing has found a much more prominent spot in the daily life of software engineers with the uprise of test-driven development and many surrounding tools, such as the xUnit family of testing frameworks, testing a program well remains an art. A long-standing dream of software engineering researchers is therefore to fully automate software testing, and to relieve the developers and testers of much of the legwork needed to ensure an acceptable level of software quality. We have discussed various approaches to automatically generating test data in this chapter, but despite these, the dream of full automation remains a dream.

On one hand, while test generation techniques have made tremendous advances over the last decades, fully testing complex systems is still a challenge. Sometimes this is because program inputs are simply too complex to handle (e.g., programs depending on complex environment states or complex input files), require domain knowledge (e.g., testing a web app often requires knowing username and password), or are simply larger than what test generation tools can realistically handle. This leaves many different open challenges on improving test generation techniques such that they become able to cope with any type of system.

A second challenge is that tests generated for a specific aim, such as reaching a particular point in the code, may be completely different to what a human would produce. As a simple example, generating a text input that satisfies some conditions may look like an arbitrary sequence of characters, rather than actual words. If such a test reveals a bug, then developers may have a much harder time understanding the test and debugging the problem. Maintaining such a test as part of a larger code base will be similarly difficult. Tests making unrealistic assumptions may also be irrelevant—for example, it is often possible to trigger NullPointerExceptions in code by passing in null arguments, and even though these exceptions technically represent bugs, developers might often not care about them, as the same scenario may simply be impossible given the context of a realistic program execution. Finally, test generation typically only covers the part of test *data* generation, meaning that a human is required to find a test oracle matching the test data—which again is likely more difficult with unrealistic or unreadable tests.

4.3 Test Oracles

In order to find faults, tests require test oracles, i.e., some means to check whether the behavior observed during test execution matches the expected behavior. For example, a test oracle for a simple function would consist of comparing the return

value of a function to an expected value. Recall the example test input of (0, 0) for our example power function in Sect. 2: For this input, we don't expect a return value, but an exception to occur; this expectation similarly represents a test oracle. For more complex types of tests and programs, what constitutes a test oracle similarly becomes a more complex question (Memon et al. 2003a).

The oracle problem is maybe the largest challenge on the path to full test automation: Sect. 3.2 described different techniques to generate tests automatically, but in most cases, this refers only to generating test *data* or test *inputs*. The only exception is if tests are derived from formal specifications or test models, as the specification or model can also serve as an automated oracle. For example, given a state machine model, we can check whether the system under test produces the same outputs as the model and whether the system reaches the same states as the model. The main challenge with this approach lies in the required mapping between abstract states and outputs to concrete observations.

Having a formal specification that serves as test oracle is the ideal case, but it requires the human effort to generate an accurate specification in the first place, and this needs to be maintained and kept up to date with the code base. Where such a specification is not available, there is a range of different ways to address the test oracle problem, and these are comprehensively surveyed by Barr et al. (2015). However, the test oracle problem remains one of the central problems in software testing, and there is no ideal solution yet.

At the far end of the spectrum of test oracles is the common assumption that a human developer or tester will provide that oracle for an automatically generated test input. However, this can be challenging as discussed above, and recent experiments (Fraser et al. 2015) suggest that simply dropping generated test inputs on developers and hoping that they improve their testing with that are overly optimistic—clearly, automated tools need to do more to support testers. One direction of work that aims at providing tool support to developers and testers who have to create test oracles lies in producing suggested oracles. For example, Fraser and Zeller (2012) have shown that mutation analysis, as discussed earlier in this chapter, is a useful vehicle to evaluate which candidate oracles are best suited for given test data. However, it is clear that more work is required in order to provide suitable oracle suggestions in more general settings.

An alternative source of oracles is provided by runtime checking, and this is very well suited for combination with automated test generation. For example, any runtime environment provides a *null oracle* of whether the program crashed or not. Beyond this, program assertions, originally introduced to support formal verification of programs (Clarke and Rosenblum 2006), can be checked at runtime, during test execution. A classical program assertion is a Boolean expression included in the source code and typically raises an exception if the asserted condition is violated, but more elaborate variants have been made popular as part of the idea of design by contract (Meyer 1992), where object-oriented classes are provided together with pre-/post-conditions and invariants; violations of such contract by automatically generated tests can serve to find faults (Ciupa et al. 2007). Again, a main challenge is that there is the human effort of creating these specifications in the first place.

The challenge of the oracle problem goes beyond the problem of finding a specific expected output for a given input to a program: For many categories of programs, it is more difficult to find oracles. For example, for scientific programs the exact output is often simply not known ahead of time, and for randomized and stochastic programs, the output may change even when executed repeatedly with the same input. This is a problem that was identified, once more, by Weyuker (1982). Often, metamorphic testing (Chen et al. 1998) can be applied to some degree in such situations, but the test oracle problem definitely has plenty of remaining challenges in store.

4.4 Flaky Tests

When an automated test fails during execution, then that usually either means that it has detected a fault in the program, or that the test is out of date and needs to be updated to reflect some changed behavior of the program under test. However, one issue that has arisen together with the increased automation is that, sometimes, tests do not fail deterministically: A test that has intermittent failures is known as a *flaky* test. Recent studies have confirmed this is a substantial problem in practice (e.g., at Microsoft (Herzig and Nagappan 2015), Google (Memon et al. 2017), and on TravisCI (Labuschagne et al. 2017)).

Flaky tests can have multiple reasons (Luo et al. 2014). For example, multi-threaded code is inherently non-deterministic since the thread schedule can vary between test executions. It is a common (bad) practice in unit testing to use `wait` instructions to ensure that some multi-threaded computation has completed. However, there are no guarantees, and it can happen that such a test sometimes fails on a continuous testing platform if the load is high. Other sources of non-determinism such as the use of Java reflection or iterations over Java `HashSet` or `HashMap` data structures can similarly cause tests to sometimes pass and sometimes fail. Any type of dependency on the environment can make a test flaky; for example, a test accessing a web service might sometimes fail under a high network load. Dependencies between tests are a further common source of flaky tests (Zhang et al. 2014). For example, if static (global) variables are read and written by different tests, then the order in which these tests are executed has an influence on the result.

Common solutions to such problems are to use mock objects to replace dependencies or to make sure that all tests are in clean environments (Bell and Kaiser 2014). Recent advances also resulted in techniques to identify state polluting tests (e.g. Gyori et al. 2015; Gambi et al. 2018). However, the problem of flaky tests remains an ongoing challenge.

4.5 Legacy Tests

The focus of this chapter was mainly on how to derive more and better tests, and how to evaluate these tests. With increased use of advanced testing techniques and automation of these tests (e.g., using xUnit frameworks), a new problem in testing is emerging: Sometimes we now may have too many tests. To some degree this is anticipated and alleviated by regression testing techniques discussed in this chapter. Most of these techniques aim at reducing the execution time caused by a large set of tests. This, however, is not the only problem: Automated tests need to be maintained like any other code in a software project. The person who wrote a test may have long left the project when a test fails, making the question of whether the test is wrong or the program under test is wrong a difficult one to answer. Indeed, every test failure may also be a problem in the test code. Thus, not every test provides value. There is some initial work in the direction of this challenge; in particular, Herzig et al. (2015) skip tests that are likely not providing value, but ultimately they are still kept and executed at some point. This leaves the question: when do we delete tests? A related problem that may also arise from large test sets, but maybe one that is easier to answer than the first one, is which of a set of failing tests to inspect first. This is related to the larger problem of prioritizing bugs—not every bug is crucial, and so not every failing test may have a high priority to fix. How can we decide which tests are more important than others?

4.6 Nonfunctional Testing

Testing research in general, but also in this chapter, puts a strong emphasis on functional correctness of the program under test. Almost all techniques we discussed in this chapter are functional testing techniques. This, however, is only one facet in the spectrum of software quality. There are many nonfunctional properties that a program under test may need to satisfy, such as availability, energy consumption, performance security, reliability, maintainability, safety, testability, or several others. Considering all these properties, it might be somewhat surprising that research focuses mostly on functional properties. This might be because functionality is often easier to specify clearly, whereas specifying nonfunctional requirements is often a somewhat more fuzzy problem.

Consequently, automated approaches are most commonly applied to nonfunctional properties that are easy to measure, such as performance and energy consumption. Performance is particularly of concern for web applications as well as embedded real-time systems and, for example, Grechanik et al. (2012) used a feedback loop where generated tests are used to identify potential performance bottlenecks, which are then tested more. Other problems related to performance are the behavior under heavy load (load testing), for which test generation has also been demonstrated as a suitable technique by, for example, Avritzer and Weyuker (1995)

and Zhang et al. (2011). The recent surge of mobile devices and applications has fostered research on measuring and optimizing the energy consumption of these applications. Testing and analysis have been suggested as possible means to achieve this (e.g., Hao et al. 2013). Besides these nonfunctional properties, automation is sparse in most areas. Search-based testing would be particularly well suited to target nonfunctional testing, in particular as anything quantifiable can already support a basic search-driven approach (Harman and Clark 2004). However, Afzal et al. (2009) only identified 35 articles on search-based testing of nonfunctional properties in their survey, showing that there is significant potential for future work.

In particular, one specific area that is already seeing increased interest, but will surely see much more attention in the future, is the topic of testing for security—after all, most security leaks and exploits are based on underlying software bugs. A popular approach to security testing currently is fuzzing (see Sect. 3.3), but there are many dimensions of security testing, as nicely laid out by Potter and McGraw (2004).

4.7 Testing Domain-Specific Software

While readers of books like this one will presumably have a general interest in software engineering, for example, as part of higher education, software is not written only by professional software engineers. There is an increasing number of people who are not software engineers but write computer programs to help them in their primary job. This is known as end-user development (Lieberman et al. 2006), and the prime example of this are scientists, physicists, and engineers. Spreadsheet macros provided a further surge to end-user programming activities, and today languages such as Python and PHP let anyone, from doctors, accountants, teachers, to marketing assistants, write programs. Without software engineering education, these end-user programmers usually have little knowledge about systematic testing and the techniques described in this chapter. Nevertheless, software quality should be of prime concern even for an accountant writing a program that works on the credit history of a customer. Conveying knowledge about testing to these people, and providing them with the tools and techniques they need to do testing, is a major challenge of ever increasing importance.

4.8 The Academia-Industry Gap

A recurring topic at academic testing conferences these days is the perceived gap between software testing research and software testing practice. Researchers develop advanced coverage and mutation analysis techniques, and practitioners measure only statement coverage if they measure coverage at all. Researchers develop automated specification-driven testing techniques, while practitioners often

prefer to write code without formal specifications. Researchers develop automated test *generation* techniques, while in practice automated test *execution* is sometimes already considered advanced testing. While this chapter is not the place to summarize the ongoing debate, the academia-industry gap does create some challenges worth mentioning in this section on future challenges in testing: Researchers need to get informed about the testing problems that industry faces; practitioners need to receive word about the advanced testing techniques available to help them solve their problems. How these challenges can be overcome is difficult to say, but one opportunity is given by development of more testing tools that practitioners can experiment with, and data-sets that researchers can investigate.

5 Conclusions

If software is not appropriately tested, then bad things can happen. Depending on the type of software, these bad things range from mild annoyance to people being killed. To avoid this, software needs to be tested. In this chapter, we discussed techniques to create tests systematically and automatically, how to evaluate the quality of these test sets, and how to execute tests in the most efficient way. Knowledge of these aspects is crucial for a good tester, but there are other aspects that a good tester must also face which we did not cover in this chapter.

Who Does the Testing?
Some years ago, the answer to this question was clear: Testing principle number 2 in Myer's classical book (Myers 1979) states that a programmer should avoid testing their own program. The reasoning behind this principle is intuitive: A person who made a mistake while writing a program may make the same mistake again during testing, and programmers may also be "gentle" when testing their program, rather than trying to break it. Thus, testing would be performed by dedicated quality assurance teams or even outsourced to testing companies. Today, the rise of test-driven development (Beck 2003) and the heavy focus of agile development methodologies on testing means that a lot of testing effort now lies with the developers. Developers write unit tests even before they write code; developers write integration tests that are frequently executed during continuous integration. Software is even tested by the *users*—for example, Google's Gmail application was in "beta" status for years, meaning that everyone using it was essentially a tester. Continuous delivery (Humble and Farley 2010) opens up new possibilities for testing; for example, "canary releases" are used to get small subgroups of users try and test new features before they are released to the wider public. Thus, the question of who does the testing is difficult to answer today.

When Is Testing Performed?
Classical software engineering education would teach us that the earlier a bug is discovered, the cheaper it is to detect; thus testing ideally starts even before coding starts; the V-model comes to mind. The same arguments as outlined above also

mean that it is no longer clear when testing is done; this depends on who does the testing. The argument of the increasing costs when bugs are found in later phases of the software project is also blurred by the frequent releases used in today's agile software world. We have all become used to updating all the apps on our devices frequently as new bugs are fixed after release. For non-safety critical software, we as users are essentially permanent beta testers. It is still true that the earlier a bug is fixed the better; for example, people may stop using software and switch to a different app if they hit too many bugs. However, even more so than finding bugs as early as possible is to find the *important* bugs as early as possible.

What Other Tasks Are Involved in a Testing Process?
Developer testing is sometimes ad hoc; you write the tests you need to help you write a new feature or debug a broken feature. Systematic testing is usually less ad hoc, in particular if performed by dedicated testers or quality assurance teams. In this case, there are often strict processes that testers have to follow, starting with a test planning phase and producing elaborate test reports as the part of the process. This particularly holds when process models like Capability Maturity Model Integration (CMMI) are applied. Classical software engineering literature covers this extensively.

References

Acree, A.T.: On mutation. PhD thesis, Georgia Institute of Technology, Atlanta, GA (1980)
Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Inf. Softw. Technol. **51**(6), 957–976 (2009)
Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. **36**(6), 742–762 (2010)
Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
Ammann, P., Offutt, J., Huang, H.: Coverage criteria for logical expressions. In: IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 99–107 (2003)
Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 402–411 (2005)
Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Softw. Test. Verification Reliab. **23**(2), 119–147 (2013)
Arcuri, A., Briand, L.: Adaptive random testing: an illusion of effectiveness? In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 265–275 (2011)
Avritzer, A., Weyuker, E.: The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. **21**(9), 705–716 (1995)
Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
Beck, K.: Kent Beck's Guide to Better Smalltalk: A Sorted Collection, vol. 14. Cambridge University Press, Cambridge (1999)
Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional, Upper Saddle River (2003)
Bell, J., Kaiser, G.: Unit test virtualization with VMVM. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 550–561. ACM, New York (2014)

Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM Syst. J. **22**(3), 229–245 (1983)

Briand, L.C., Labiche, Y., He, S.: Automating regression test selection based on UML designs. Inf. Softw. Technol. **51**(1), 16–30 (2009)

Broekman, B., Notenboom, E.: Testing Embedded Software. Pearson Education, Boston (2003)

Bron, A., Farchi, E., Magid, Y., Nir, Y., Ur, S.: Applications of synchronization coverage. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 206–212. ACM, New York (2005)

Bühler, O., Wegener, J.: Evolutionary functional testing. Comput. Oper. Res. **35**(10), 3144–3160 (2008)

Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Symposium on Operating Systems Design and Implementation (USENIX), pp. 209–224 (2008)

Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 1066–1071 (2011)

Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Department of Computer Science, Hong Kong University of Science and Technology, Technical Report HKUST-CS98-01 (1998)

Chen, T.Y., Lau, M.F., Yu, Y.T.: Mumcut: a fault-based strategy for testing boolean specifications. In: Sixth Asia Pacific Software Engineering Conference (APSEC), pp. 606–613. IEEE, New York (1999)

Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: Advances in Computer Science, pp. 320–329 (2004)

Chilenski, J.J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report, DTIC Document (2001)

Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **9**(5), 193–200 (1994)

Choudhary, S.R., Gorla, A., Orso, A.: Automated test input generation for android: are we there yet?(e). In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 429–440. IEEE, New York (2015)

Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178 (1978)

Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Experimental assessment of random testing for object-oriented software. In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 84–94 (2007)

Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM Sigplan Not. **46**(4), 53–64 (2011)

Clarke, D., Lee, I.: Testing real-time constraints in a process algebraic setting. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 51–60. ACM, New York (1995)

Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Softw. Eng. Notes **31**(3), 25–37 (2006)

Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Softw. **13**(5), 83 (1996)

Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. **23**(7), 437–444 (1997)

Cohen, M., Gibbons, P., Mugridge, W., Colbourn, C.: Constructing test suites for interaction testing. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 38–48 (2003)

Commission, I.E., et al.: IEC 61508: functional safety of electrical. Electronic/Programmable Electronic Safety-Related Systems (1999)

DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Trans. Softw. Eng. **17**(9), 900–910 (1991)

DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**(4), 34–41 (1978)
Derderian, K., Hierons, R.M., Harman, M., Guo, Q.: Automated unique input output sequence generation for conformance testing of FSMs. Comput. J. **49**(3), 331–344 (2006)
Di Lucca, G.A., Fasolino, A.R.: Testing web-based applications: the state of the art and future trends. Inf. Softw. Technol. **48**(12), 1172–1186 (2006)
Dowson, M.: The Ariane 5 software failure. ACM SIGSOFT Softw. Eng. Notes **22**(2), 84 (1997)
Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Trans. Softw. Eng. **SE-10**(4), 438–444 (1984)
Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded java program test generation. IBM Syst. J. **41**(1), 111–125 (2002)
Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)
Faught, D.R.: Keyword-driven testing. Sticky minds. https://www.stickyminds.com/article/keyword-driven-testing [online]. Retrieved 28.10.2018
Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. **5**(1), 63–86 (1996)
Forrester, J.E., Miller, B.P.: An empirical study of the robustness of windows nt applications using random testing. In: Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4, WSS'00, p. 6. USENIX Association, Berkeley (2000)
Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. ACM Comput. Surv. **8**(3), 305–330 (1976)
Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Softw. Eng. **39**(2), 276–291 (2013)
Fraser, G., Walkinshaw, N.: Assessing and generating test sets in terms of behavioural adequacy. Softw. Test. Verification Reliab. **25**(8), 749–780 (2015)
Fraser, G., Wotawa, F.: Redundancy Based Test-Suite Reduction, pp. 291–305. Springer, Heidelberg (2007)
Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Trans. Softw. Eng. **28**(2), 278–292 (2012)
Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Softw. Test. Verification Reliab. **19**(3), 215–261 (2009)
Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated unit test generation really help software testers? a controlled empirical study. ACM Trans. Softw. Eng. Methodol. **24**(4), 23 (2015)
Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. **17**(6), 591–603 (1991)
Gambi, A., Bell, J., Zeller, A.: Practical test dependency detection. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Västerås, Sweden, pp. 1–11 (2018)
Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 146–162. Springer, Berlin (1999)
Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. SIGPLAN Not. **40**(6), 213–223 (2005)
Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. ACM Sigplan Not. **43**(6), 206–215 (2008a)
Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium (NDSS), pp. 1–16 (2008b)
Goodenough, J.B., Gerhart, S.L.: Toward a theory of test data selection. IEEE Trans. Softw. Eng. **SE-1**(2), 156–173 (1975)
Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. ACM SIGSOFT Softw. Eng. Notes **23**(2), 53–62 (1998)
Graves, T.L., Harrold, M.J., Kim, J.M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. ACM Trans. Softw. Eng. Methodol. **10**(2), 184–208 (2001)

Grechanik, M., Fu, C., Xie, Q.: Automatically finding performance problems with feedback-directed learning software testing. In: Proceedings of the 34th International Conference on Software Engineering, pp. 156–166. IEEE Press, New York (2012)
Grindal, M., Offutt, J., Andler, S.: Combination testing strategies: a survey. Softw. Test. Verification Reliab. **15**(3), 167–199 (2005)
Gross, F., Fraser, G., Zeller, A.: Search-based system testing: high coverage, no false alarms. In: ACM International Symposium on Software Testing and Analysis (2012)
Gutjahr, W.J.: Partition testing vs. random testing: the influence of uncertainty. IEEE Trans. Softw. Eng. **25**(5), 661–674 (1999)
Gyori, A., Shi, A., Hariri, F., Marinov, D.: Reliable testing: detecting state-polluting tests to prevent test dependency. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA), pp. 223–233. ACM, New York (2015)
Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, pp. 970–978. Wiley, New York (1994)
Hamlet, D., Taylor, R.: Partition testing does not inspire confidence. IEEE Trans. Softw. Eng. **16**(12), 1402–1411 (1990)
Hammoudi, M., Rothermel, G., Tonella, P.: Why do record/replay tests of web applications break? In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 180–190. IEEE, New York (2016)
Hanford, K.V.: Automatic generation of test cases. IBM Syst. J. **9**(4), 242–257 (1970)
Hao, S., Li, D., Halfond, W.G., Govindan, R.: Estimating mobile application energy consumption using program analysis. In: 35th International Conference on Software Engineering, pp. 92–101. IEEE, New York (2013)
Harman, M., Clark, J.: Metrics are fitness functions too. In: Proceedings. 10th International Symposium on, Software Metrics, pp. 58–69. IEEE, New York (2004)
Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)
Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. ACM SIGSOFT Softw. Eng. Notes **19**(5), 154–163 (1994)
Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. **2**(3), 270–285 (1993)
Herzig, K., Nagappan, N.: Empirically detecting false test alarms using association rules. In: ICSE SEIP (2015)
Herzig, K., Greiler, M., Czerwonka, J., Murphy, B.: The art of testing less without sacrificing quality. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 483–493. IEEE Press, New York (2015)
Holzmann, G.J.: The model checker spin. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
Holzmann, G.J., H Smith, M.: Software model checking: extracting verification models from source code. Softw. Test. Verification Reliab. **11**(2), 65–79 (2001)
Howden, W.E.: Reliability of the path analysis testing strategy. IEEE Trans. Softw. Eng. **SE-2**(3), 208–215 (1976)
Hsu, H., Orso, A.: MINTS: a general framework and tool for supporting test-suite minimization. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 419–429 (2009)
Huang, J.C.: An approach to program testing. ACM Comput. Surv. **7**(3), 113–128 (1975)
Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Pearson Education, Boston (2010)
Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 435–445. ACM, New York (2014)
Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Technical Report TR-09-06, CREST Centre, King's College London, London (2009)
Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)

Jiang, B., Zhang, Z., Chan, W., Tse, T.: Adaptive random test case prioritization. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 233–244 (2009)
Johnson, L.A., et al.: Do-178b, software considerations in airborne systems and equipment certification. Crosstalk, October 1998
Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Softw. Eng. J. **11**(5), 299–306 (1996)
Just, R., Schweiggert, F., Kapfhammer, G.: Major: an efficient and extensible tool for mutation analysis in a Java compiler. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 612–615 (2011)
Just, R., Jalali, D., Inozemtseva, L., Ernst, M., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 654–665 (2014)
Kane, S., Liberman, E., DiViesti, T., Click, F.: Toyota Sudden Unintended Acceleration. Safety Research & Strategies, Rehoboth (2010)
Kaner, C.: Exploratory testing. In: Quality Assurance Institute Worldwide Annual Software Testing Conference (2006)
King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. **16**, 870–879 (1990)
Korel, B., Tahat, L.H., Vaysburg, B.: Model based regression test reduction using dependence analysis. In: IEEE International Conference on Software Maintenance (ICSM), pp. 214–223 (2002)
Korel, B., Tahat, L.H., Harman, M.: Test prioritization using system models. In: IEEE International Conference on Software Maintenance (ICSM), pp. 559–568 (2005)
Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods Syst. Des. **34**(3), 238–304 (2009)
Kuhn, D., Wallace, D., Gallo, A. Jr.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. **30**(6), 418–421 (2004)
Labuschagne, A., Inozemtseva, L., Holmes, R.: Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, pp. 821–830. ACM, New York (2017)
Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines-a survey. Proc. IEEE **84**(8), 1090–1123 (1996)
Leitner, A., Oriol, M., Zeller, A., Ciupa, I., Meyer, B.: Efficient unit test case minimization. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 417–420 (2007)
Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. Computer **26**(7), 18–41 (1993)
Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. **33**(4), 225–237 (2007)
Lieberman, H., Paternò, F., Klann, M., Wulf, V.: End-user development: an emerging paradigm. In: End User Development, pp. 1–8. Springer, Berlin (2006)
Liggesmeyer, P., Trapp. M.: Trends in embedded software engineering. IEEE Softw. **26**(3), 19–25 (2009)
Lu, S., Jiang, W., Zhou, Y.: A study of interleaving coverage criteria. In: The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, pp. 533–536. ACM, New York (2007)
Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. ACM Sigplan Not. **43**(3), 329–339 (2008)
Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 643–653. ACM, New York (2014)

Mansour, N., Bahsoon, R., Baradhi, G.: Empirical comparison of regression test selection algorithms. J. Syst. Softw. **57**(1), 79–90 (2001)
McMillan, K.L.: Symbolic model checking. In: Symbolic Model Checking, pp. 25–60. Springer, Berlin (1993)
McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verification Reliab. **14**(2), 105–156 (2004)
Memon, A.M., Soffa, M.L., Pollack, M.E.: Coverage criteria for gui testing. ACM SIGSOFT Softw. Eng. Notes **26**(5), 256–267 (2001)
Memon, A., Banerjee, I., Nagarajan, A.: What test oracle should i use for effective gui testing? In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings, pp. 164–173. IEEE, New York (2003a)
Memon, A.M., Banerjee, I., Nagarajan, A.: Gui ripping: reverse engineering of graphical user interfaces for testing. In: WCRE, vol. 3, p. 260 (2003b)
Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J.: Taming Google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 233–242 (2017)
Meyer, B.: Applying'design by contract'. Computer **25**(10), 40–51 (1992)
Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Trans. Softw. Eng. **27**(12), 1085–1110 (2001)
Miller, J.C., Maloney, C.J.: Systematic mistake analysis of digital computer programs. Commun. ACM **6**(2), 58–63 (1963)
Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Trans. Softw. Eng. **2**(3), 223–226 (1976)
Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Commun. ACM **33**(12), 32–44 (1990)
Mirarab, S., Akhlaghi, S., Tahvildari, L.: Size-constrained regression test case selection using multicriteria optimization. IEEE Trans. Softw. Eng. **38**(4), 936–956 (2012)
Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 267–280. USENIX Association, Berkeley (2008)
Myers, G.: The Art of Software Testing. Wiley, New York (1979)
Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), 1–29 (2011). Article 11
Nielsen, B., Skou, A.: Automated test generation from timed automata. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 343–357. Springer, Berlin (2001)
Nistor, A., Luo, Q., Pradel, M., Gross, T.R., Marinov, D.: BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 727–737 (2012)
North, D.: Behavior modification: the evolution of behavior-driven development. Better Softw. **8**(3) (2006). https://www.stickyminds.com/better-software-magazine-volume-issue/2006-03
Ntafos, S.C.: A comparison of some structural testing strategies. IEEE Trans. Softw. Eng. **14**(6), 868 (1988)
Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: International Conference on the Unified Modeling Language: Beyond the Standard (UML), pp. 416–429. Springer, Berlin (1999)
Offutt, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. In: Wong, W.E. (ed.) Mutation Testing for the New Century, pp. 34–44. Kluwer Academic Publishers, Boston (2001)
Offutt, A.J, Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 100–107 (1993)
Osterweil, L.J., Fosdick, L.D.: Dave–a validation error detection and documentation system for fortran programs. Softw. Pract. Exp. **6**(4), 473–486 (1976)

Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating functional tests. Commun. ACM **31**(6), 676–686 (1988)
Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems and Application (OOPSLA Companion), pp. 815–816. ACM, New York (2007)
Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 75–84 (2007)
Panichella, A., Oliveto, R., Penta, M.D., Lucia, A.D.: Improving multi-objective test case selection by injecting diversity in genetic algorithms. IEEE Trans. Softw. Eng. **41**(4), 358–383 (2015)
Pargas, R.P., Harrold, M.J., Peck, R.: Test-data generation using genetic algorithms. Softw. Test. Verification Reliab. **9**(4), 263–282 (1999)
Potter, B., McGraw, G.: Software security testing. IEEE Secur. Priv. **2**(5), 81–85 (2004)
Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Softw. Eng. (4), 367–375 (1985). https://doi.org/10.1109/TSE.1985.232226
Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. **6**(2), 173–210 (1997)
Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: IEEE International Conference on Software Maintenance (ICSM), pp. 34–43 (1998)
Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27**(10), 929–948 (2001)
Sabnani, K., Dahbura, A.: A protocol test generation procedure. Comput Netw. ISDN Syst. **15**(4), 285–297 (1988)
Segura, S., Fraser, G., Sanchez, A., Ruiz-Cortes, A.: A survey on metamorphic testing. IEEE Trans. Softw. Eng. (TSE) **42**(9), 805–824 (2016)
Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ACM Symposium on the Foundations of Software Engineering, pp. 263–272. ACM, New York (2005)
Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M., Erdogmus, H.: What do we know about test-driven development? IEEE Softw. **27**(6), 16–19 (2010). https://doi.org/10.1109/MS.2010.152
Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: ACM SIGPLAN Notices, vol. 35, pp. 1–13. ACM, New York (1999)
Steenbuck, S., Fraser, G.: Generating unit tests for concurrent classes. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 144–153. IEEE, New York (2013)
Stocks, P., Carrington, D.: A framework for specification-based testing. IEEE Trans. Softw. Eng. **22**(11), 777–793 (1996)
Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, Boston (2007)
Tassey, G.: The economic impacts of inadequate infrastructure for software testing, final report. National Institute of Standards and Technology (2002)
Tikir, M.M., Hollingsworth, J.K.: Efficient instrumentation for code coverage testing. ACM SIGSOFT Softw. Eng. Notes **27**(4), 86–96 (2002)
Tonella, P.: Evolutionary testing of classes. In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 119–128 (2004)
Tracey, N., Clark, J., Mander, K., McDermid, J.A.: An automated framework for structural test-data generation. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 285–288 (1998)
Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence (1996)
Tretmans, J., Brinksma, E.: TorX: automated model-based testing. In: First European Conference on Model-Driven Software Engineering, pp. 31–43 (2003)
Untch, R.H., Offutt, A.J., Harrold, M.J.: Mutation analysis using mutant schemata. ACM SIGSOFT Softw. Eng. Notes **18**(3), 139–148 (1993)

Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2010)
Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**(5), 297–312 (2012)
Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. ACM SIGSOFT Softw. Eng. Notes **29**(4), 97–107 (2004)
Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Inf. Softw. Technol. **43**(14), 841–854 (2001)
Weyuker, E.J.: On testing non-testable programs. Comput. J. **25**(4), 465–470 (1982)
Weyuker, E.J.: Assessing test data adequacy through program inference. ACM Trans. Program. Lang. Syst. **5**(4), 641–655 (1983)
Weyuker, E., Goradia, T., Singh, A.: Automatically generating test data from a boolean specification. IEEE Trans. Softw. Eng. **20**(5), 353 (1994)
White, L.J., Cohen, E.I.: A domain strategy for computer program testing. IEEE Trans. Softw. Eng. **6**(3), 247–257 (1980)
Whittaker, J.A., Arbon, J., Carollo, J.: How Google Tests Software. Addison-Wesley, Upper Saddle River (2012)
Wong, W.E., Horgan, J.R., London, S., Mathur, A.P.: Effect of test set minimization on fault detection effectiveness. Softw. Pract. Exper. **28**(4), 347–369 (1998)
Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., Karapoulios, K.: Application of genetic algorithms to software testing. In: Proceedings of the 5th International Conference on Software Engineering and Applications, pp. 625–636 (1992)
Yang, C.S.D., Souter, A.L., Pollock, L.L.: All-du-path coverage for parallel programs. ACM SIGSOFT Softw. Eng. Notes **23**(2), 153–162 (1998)
Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 140–150. ACM, New York (2007)
Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012)
Young, M., Pezze, M.: Software Testing and Analysis: Process, Principles and Techniques. Wiley, New York (2005)
Yuan, X., Cohen, M.B., Memon, A.M.: GUI interaction testing: incorporating event context. IEEE Trans. Softw. Eng. **37**(4), 559–574 (2011)
Zhang, P., Elbaum, S., Dwyer, M.B.: Automatic generation of load tests. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 43–52. IEEE Computer Society, Washington (2011)
Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M.D., Notkin, D.: Empirically revisiting the test independence assumption. In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 385–396. ACM, New York (2014)
Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (1997)