2 Testing and verification processes

This chapter describes the impact of model-based testing on the testing life cycle and the benefits and challenges associated with adopting model-based testing. We start with an overview of some traditional testing processes and show how these differ from the model-based testing process. Then we discuss maturity levels, look at several case studies that demonstrate the pain and the gain of putting model-based testing into practice, and finish by discussing the benefits and limitations of model-based testing.

2.1 What is testing

> Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.

This definition of testing, from the IEEE Software Engineering Body of Knowledge (SWEBOK 2004),[1] describes the top-level goals of testing. It goes on to give more detail:

> Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.

We've emphasized in italics the words that capture the key features of software testing; these are their definitions as they relate to this book.

Dynamic: This means that we execute the program with specific input values to find failures in its behavior. In contrast, static techniques (e.g., inspections, walkthroughs, and static analysis tools) do not require execution of the program. One of the big advantages of (dynamic) testing is that we are executing the actual program either in its real environment or in an environment with simulated interfaces as close as possible to the real environment. So we are not only testing that the design and code are correct, but we are also testing the compiler, the libraries, the operating system and network support, and so on.

Finite: Exhaustive testing is not possible or practical for most real programs. They usually have a large number of allowable inputs to each operation, plus even more invalid or unexpected inputs, and the possible sequences of operations is usually infinite as well. So we must choose a smallish number of tests so that we can run the tests in the available time. For example, if we want to perform nightly regression testing, our tests should take less than 12 hours!

[1] The SWEBOK can be downloaded from http://www.swebok.org or purchased from the IEEE

Selected: Since we have a huge or infinite set of possible tests but can afford to run only a small fraction of them, the key challenge of testing is how to select the tests that are most likely to expose failures in the system. This is where the expertise of a skilled tester is important—he or she must use knowledge about the system to guess which sets of inputs are likely to produce the same behavior (this is called the uniformity assumption) and which are likely to produce different behavior. There are many informal strategies, such as equivalence class and boundary value testing,[2] that can help in deciding which tests are likely to be more effective. Some of these strategies are the basis of the test selection algorithms in the model-based testing tools that we use in later chapters.

Expected: After each test execution, we must decide whether the observed behavior of the system was a failure or not. This is called the oracle problem. The oracle problem is often solved via manual inspection of the test output; but for efficient and repeatable testing, it must be automated. Model-based testing automates the generation of oracles, as well as the choice of test inputs.

Before describing the various kinds of testing, we briefly review some basic terms according to standard IEEE software engineering terminology. A failure is an undesired behavior. Failures are typically observed during the execution of the system being tested. A fault is the cause of the failure. It is an error in the software, usually caused by human error in the specification, design, or coding process. It is the execution of the faults in the software that causes failures. Once we have observed a failure, we can investigate to find the fault that caused it and correct that fault.

So testing is the activity of executing a system in order to detect failures. It is different from, and complementary to, other quality improvement techniques such as static verification, inspections, and reviews. It is also distinct from the debugging and error-correction process that happens after testing has detected a failure.

In fact, there are many kinds of testing. Figure 1.2 shows one way to classify various kinds of testing along three dimensions (adapted from [Tre04]). One axis shows the scale of the system under test (SUT), ranging from small units up to the whole system. Unit testing involves testing a single unit at a time, such as a single procedure or a single class. Component testing tests each component/subsystem separately, and integration testing aims at testing to ensure that several components work together correctly. System testing involves testing the system as a whole. Model-based testing can be applied to any of these levels.

Another axis shows the different characteristics that we may want to test. The most common kind of testing is functional testing (also known as behavioral testing), where we aim to find errors in the functionality of the system—for example, testing that the correct outputs are produced for given inputs. Robustness testing aims at finding errors in the system under invalid conditions, such as unexpected inputs, unavailability of dependent applications, and hardware or network failures. Performance testing tests the

[2]See Lee Copeland's book [Cop04] for a comprehensive overview of the most popular informal test design techniques.

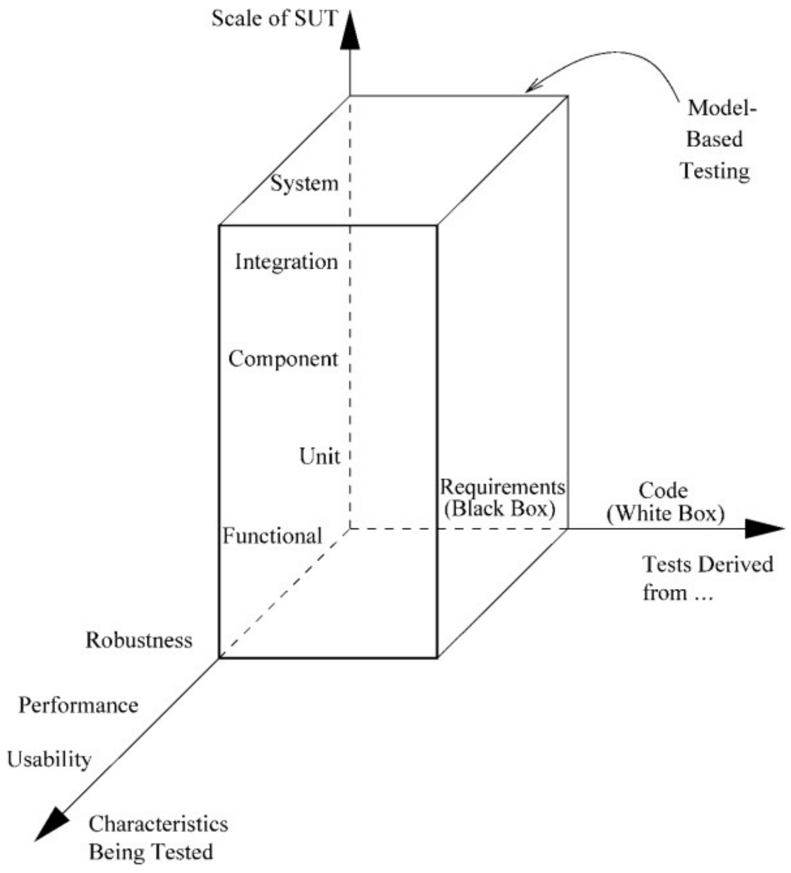

Figure 2.1: Different kinds of testing. Source: From Tretmans [Tre04]. Used with permission.

throughput of the system under heavy load. Usability testing focuses on finding user interface problems, which may make the soft- ware difficult to use or may cause users to misinterpret the output. The main use of model-based testing is to generate functional tests, but it can also be used for some kinds of robustness testing such as testing the system with invalid inputs. It is not yet widely used for performance testing, but this is an area under development. The third axis shows the kind of information we use to design the tests. Black-box testing means that we treat the SUT as a "black box," so we do not use information about its internal structure. Instead, the tests are designed from the system requirements, which describe the expected behavior of that black box. On the other hand, white-box testing uses the implementation code as the basis for designing tests. For example, we might design a set of tests to ensure statement coverage of a procedure, meaning that each statement will be executed by at least one of the tests. Much has been written about the pros and cons of black-box and white- box testing. Hoframewever, the most common practice is to use black-box testing techniques to design functional and robustness tests. Some testers then use white-box coverage metrics to check which parts of the implementation have not been tested well so that extra tests can be designed for those cases. Model- based testing is a form of black-box testing because tests are generated from a model, which is derived from the requirements documentation. The next section describes model-based testing in more detail.

2.2 Classic testing processes (program-based)

When doing functional testing, there are three key issues:

This section describes several classic testing processes that are widely used in industry. We start by describing a simple manual testing process, and then we progress through several testing processes that use automated test execution. We finish with a table that shows how each of these testing processes addresses the three stated testing issues, and what issues remain unsolved. Figure 2.1 shows the notations that we use in the process diagrams in this chapter. Informal Document means a document in natural language, such as English, while Formal Documents are written in some precise notation (such as models and test scripts) that can be parsed or executed by tools.

2.2.1 A Manual Testing Process

The left side of Figure 2.2 shows a completely manual testing process. This was the earliest style of testing, but it is still widely used.

The test plan gives a high-level overview of the testing objectives, such as which aspects of the SUT should be tested, what kinds of test strategies should be used, how often testing should be performed, and how much testing will be done. The test design is done manually, based on the informal requirements documents. The output of the design stage is a human-readable document that describes the desired test cases. The description of test cases can be quite concise and high-level; many of the low-level details about interacting with the system under test can be left to the common sense of the test execution person, who is called a manual tester. However, the manual designing of the

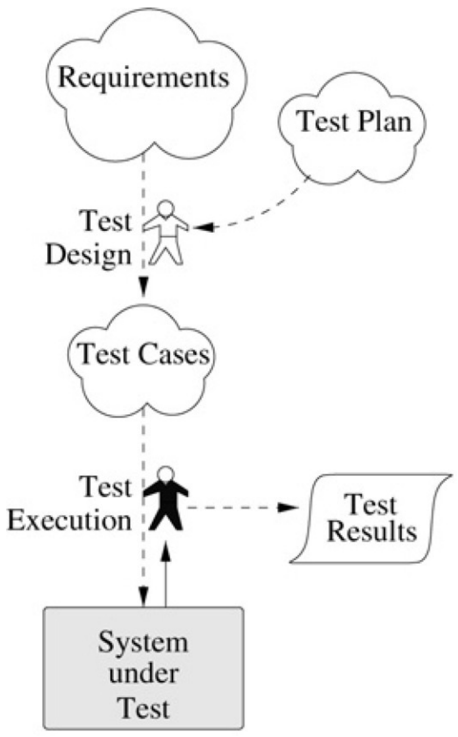

Figure 2.2: A manual testing process

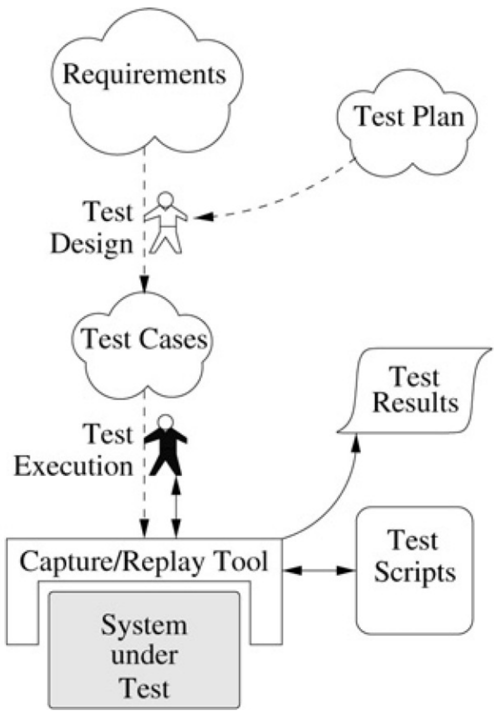

Figure 2.3: A Capture and replay process

tests is time-consuming and does not ensure systematic coverage of the SUT functionality. The test execution is also done manually. For each test case, the manual tester follows the steps of that test case, interacts directly with the SUT, compares the SUT output with the expected output, and records the test verdict. In embedded applications, where it is often not possible to interact directly with the SUT (it may be just a black box with some wires coming out), a test execution environment may be used to allow the tester to enter inputs and observe outputs. However, the execution of each test case is still performed manually. Note that the required skills of the test designer and manual tester are rather different. The test designer needs to have expert knowledge about the SUT, plus some skill with test design strategies. The manual tester has a much more menial task, which requires some knowledge of how to interact with the SUT but mostly involves simply following the steps of the test case and recording results. This manual test execution process is repeated each time a new release of the SUT needs to be tested. This quickly becomes a very boring and time- consuming task. Since there is no automation of the test execution, the cost of testing each SUT release is constant and large. In fact, the cost of repeating the manual test execution is so high that, to keep testing costs within budget, it is often necessary to cut corners by reducing the number of tests that are executed after each evolution of the code. This can result in software being delivered with incomplete testing, which introduces a significant risk regarding product maturity, stability, and robustness. The next few testing processes propose various ways of automating the test execution, to reduce this cost and permit more comprehensive testing.

2.2.2 A Capture/Replay Testing Process

Capture/replay testing attempts to reduce the cost of test re-execution by capturing the interactions with the SUT during one test execution session and then replaying those interactions during later test execution sessions. In this process, test cases are still manually designed. The right side of Figure 2.2 shows how this is different from manual test execution. The interaction with the SUT is managed by a testing tool that we call the capture/replay tool. This records all the inputs sent to the system under test and the outputs that result (e.g., procedure return results, screen snapshots, data files, etc.). Then when a new release of the SUT must be tested, the capture/replay tool can attempt to rerun all the recorded tests and report which ones fail. To rerun each recorded test, the tool sends the recorded inputs to the SUT and then compares the new outputs with the recorded outputs from the original test execution. The main problem with capture/replay testing is that it is very fragile. For example, a change to the layout of a window in the SUT (such as changing from a combo box to a radio button) or a small change to the interface of one procedure in an API can cause a large number of the recorded tests to fail. These have to be tested manually again and recorded for future sessions. This inability to adapt to small changes in the SUT creates a huge maintenance problem with the recorded tests and often leads to the capture/replay method being abandoned after several SUT releases. This problem comes from a lack of abstraction in the recorded tests. That is, it is the low-level details of the actual SUT input and output values that

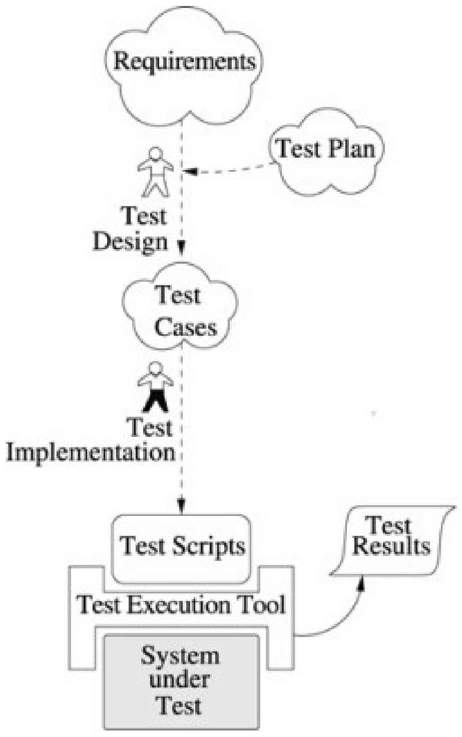

Figure 2.4: A script-based testing process-Testing processes with automated test execution: manually developed test scripts (left)

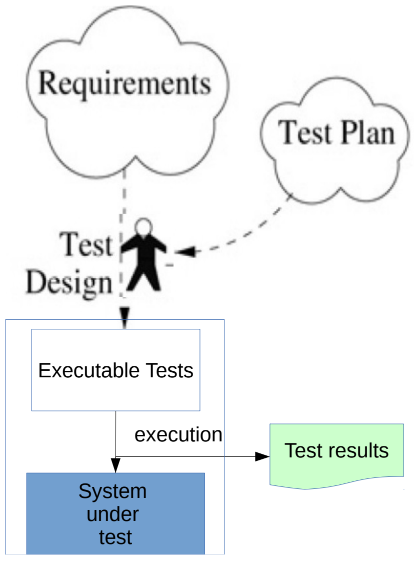

Figure 2.5: Tests as programs

are recorded rather than a higher-level, more abstract, view. The verification of the correctness of the outputs is also usually too low-level and based on comparing screen snapshots or strings rather than checking just the relevant higher-level features of the output. The key issue of automating the test execution is only partially solved by the capture/replay approach. This is due to the extreme sensitivity of this approach to any changes in the SUT. Notice also that this approach is usually used only to automate the testing of the graphical user interface (GUI) of the SUT, which is only one part of the desired functional test suite.

2.2.3 A Script-Based Testing Process

The left side of Figure 2.3 shows a testing process that uses test scripts to automate the execution of tests. A test script is an executable script that runs one or more test cases. This usually involves initializing the SUT, putting the SUT in the required context, creating the test input values, passing those inputs to the SUT, recording the SUT response, comparing that response with the expected outputs, and assigning a pass/fail verdict to each test. The test scripts may be written in some standard programming or scripting language or in a special testing lan- guage such as TTCN-3 (Testing and Test Control Notation) [WDT + 05]. So writing test scripts is now a programming task, which requires different skills from test design or test execution. Test scripts must control and observe the SUT using some API. This is a strong design constraint on the application—it must offer the required points of control and observation for testing purposes. These are important testabil- ity criteria, which are standard quality criteria in most software engineering processes. These points of control and observation are then invoked in the test scripts both to stimulate the SUT and to observe if the actual states or responses are the expected ones. The script-based testing approach solves the test execution problem by automating it. Each time that we want to rerun the tests for regression testing, this can be done for free by just running the test scripts again. However, this increases the test maintenance problem because the test scripts must evolve not only when some requirements change, but also whenever some implementation details change (e.g., when some parameters change in the API used to stimulate the SUT). Since the total size of test scripts can be nearly as big as the application under test and the details of one interface to the SUT are usually spread out over many tests, maintenance of the test scripts becomes very costly. Abstraction is the key to reducing the maintenance costs of test scripts, but the level of abstraction depends on the individual skill of the test script designer.

2.2.4 A Program-Based Testing Process

Write programs that test programs:

- the test are designed as before,
- the tester writes the tests as programs possibily in the same language as the program itself
- the execution and the verdict are done automaticlly by running the tests
- very easy to rerun test cases
- require more time to write the tests (and maintain)

2.2.5 Solved and Remaining Problems

In Table 2.1 we summarize the testing problems that are solved by each of the mentioned approaches, as well as the problems that remain unsolved. As we go down the table, the testing processes become more sophisticated and provide better solutions for minimizing the cost of executing and reexecuting tests. However, all these processes still rely on manual design of the test cases and manual tracking of the relationship between the requirements and the tests. In addition, because they all rely on manual design of the tests, none of them guarantees very systematic and repeatable coverage of the SUT behavior. The model-based testing process aims to solve the following three remaining problems that the other testing processes do not fully address:

Solved and Remaining Problems

Testing Process	Solved Problems	Remaining Problems
Manual Testing	Functional testing	Imprecise coverage of SUT functionality No capabilities for regression testing Very costly process (every test execution is done manually) No effective measurement of test coverage
Capture/ Replay	Makes it possible to automatically reexecute captured test cases	Imprecise coverage of SUT functionality Weak capabilities for regression testing (very sensitive to GUI changes) Costly process (each change implies recapturing test cases manually)
Script- Based Testing	Makes it possible to automatically execute and reexecute test scripts	Imprecise coverage of SUT functionality Complex scripts are difficult to write and maintain Requirements traceability is developed manually (costly process)
Program-based Testing	No extra language is required	It may require additional effort during maintenance

Table 2.1: Comparison of Testing Approaches

- Automation of the design of functional test cases (including generation of the expected results) to reduce the design cost and to produce test suites with systematic coverage of the model
- Reduction of the maintenance costs of the test suite
- Automatic generation of the traceability matrix from requirements to test cases

The next section presents the model-based testing process and describes each step of the process in more detail.

2.3 The model-based testing process

Model-based testing automates the detailed design of the test cases and the generation of the traceability matrix. More precisely, instead of manually writing hundreds of test cases (sequences of operations), the test designer writes an abstract model of the system under test, and then the model-based testing tool generates a set of test cases from that model. The overall test design time is reduced, and an added advantage is that one can generate a variety of test suites from the same model simply by using different test selection criteria. The model-based testing process can be divided into the following five main steps, as shown in Figure 2.4.

1. Model the SUT and/or its environment.
2. Generate abstract tests from the model.
3. Concretize the abstract tests to make them executable.

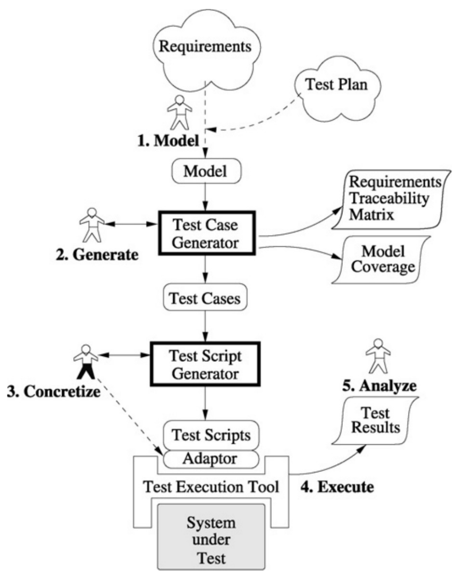

Figure 2.6: The model-based testing process (testing tools are in the boxes with very bold lines).

4. Execute the tests on the SUT and assign verdicts.
5. Analyze the test results.

Of course, steps 4 and 5 are a normal part of any testing process, even manual testing. Step 3 is similar to the “adaptor” phase of keyword-based testing, where the meaning of each keyword is defined. The first two steps distinguish model-based testing from other kinds of testing. In online model-based testing tools, steps 2 through 4 are usually merged into one step, whereas in offline model-based testing, they are usually separate. But it is still useful to explain the steps separately to ensure a clear understanding of the model- based testing process. We will now give a more detailed description of each step of the process and mention some of the practical issues about using model-based testing in large projects.

The first step of model-based testing is to write an abstract model of the system that we want to test. We call it an abstract model because it should be much smaller and simpler than the SUT itself. It should focus on just the key aspects that we want to test and should omit many of the details of the SUT. Later chapters give detailed guidelines for how to perform this modeling step using various modeling notations. While writing the model, we may also annotate it with requirements identifiers to clearly document the relationship between the informal requirements and the formal model.

After writing the model, it is advisable to use tools to check that the model is consistent and has the desired behavior. Most modeling notations provide some automated verification tools (such as typecheckers and static analysis tools), as well as some interactive tools (such as animators) that allow us to explore the behavior of the model and check that it is what we expect. The use of an animator is highly recommended for novice modelers, but experienced modelers may prefer to save time by omitting animation—any errors that remain in the model will be noticed during later steps, and the animator can be used then to pinpoint the problem in the model.

The second step of model-based testing is to generate abstract tests from the model. We must choose some test selection criteria, to say which tests we want to generate from the model, because there are usually an infinite number of possible tests. For example, we might interact with the test generation tool to focus on a particular part of the model or to choose a particular model coverage criterion, such as all-transitions, or we might write some test case specifications in some simple pattern language to specify the kinds of test we want generated.

The main output of this step is a set of abstract tests, which are sequences of operations from the model. Since the model uses a simplified view of the SUT, these abstract tests lack some of the detail needed by the SUT and are not directly executable. Most model-based testing tools also produce a requirements traceability matrix or various other coverage reports as additional outputs of this step. The requirements traceability matrix traces the link between functional requirements and generated test cases. This is generally a many-to-many relation: a requirement can be covered by several test cases, and a single test case may exercise several requirements. The coverage reports give us some indications of how well the generated test set exercises all the behaviors of the model. Note that these reports are talking about coverage of the model, not coverage of the SUT—we have not even executed the tests on the SUT yet! For example, a coverage report may give us some coverage statistics for the operations or transitions of the model, tell us what percentage of boolean decisions in the model have been tested with true and false values, or give the coverage results for many other kinds of coverage measure (see Chapter 4). We can use such coverage reports simply for statistical feedback about the quality of the generated test set, or we can use them to identify parts of the model that may not be well tested and investigate why this has happened. For example, if a particular path through the model has no tests generated for it, we could try changing some of the test generation parameters and repeat the test generation step. This is where it can be useful to use an animation tool to investigate the behavior of the path in the model and decide whether the lack of tests is due to an error in the model, a normal feature of the model, or inadequate test generation. In the latter case, if we want to improve the cover- age of our test set, we could add an abstract test for this path by hand or give the test generation tool some explicit hints about how to find the desired test.

The third step of model-based testing is to transform the abstract tests into executable concrete tests. This may be done by a transformation tool, which uses various templates and mappings to translate each abstract test case into an executable test script. Or it may be done by writing some adaptor code that wraps around the SUT and implements each abstract operation in terms of the lower-level SUT facilities. Either way, the goal of this step is to bridge the gap between the abstract tests and the concrete SUT by adding in the low-level SUT details that were not mentioned in the abstract model. One advantage of this two-layer approach (abstract tests and concrete test scripts) is that the abstract tests can be quite independent of the language used to write tests and of the test environment. By changing just the adaptor code or the translation templates, we can reuse the same set of tests in different test execution environments.

The fourth step is to execute the concrete tests on the system under test. With online model-based testing, the tests will be executed as they are produced, so the model-based testing tool will manage the test execution process and record the results. With offline model-based testing, we have just generated a set of concrete test scripts in some existing language, so we can continue to use our existing test execution tools and practices. For example, we might use Mercury TestDirector[3] to manage the tests, execute them regularly, and record the results. The fifth step is to analyze the results of the test executions and take corrective action. For each test that reports a failure, we must determine the fault that caused that failure. Again, this is similar to the traditional test analysis process. As usual, when a test fails, we may find that it is due to a fault in the SUT or we may find that it is due to a fault in the test case itself. Since we are using model-based testing, a fault in the test case must be due to a fault in the adaptor code or in the model (and perhaps also the requirements documents)[4]. So this is another place where we get feedback about the correctness of the model. In our experience, the first execution of the test set usually sees a high percentage of the tests fail—typically because of some minor errors in the adaptor code. Once these are fixed, the remaining failures are more interesting and require deeper analysis to find the fault. Perhaps roughly half of these failures will result from faults in the SUT and the other half from faults in the model and the requirements. However, this ratio can vary widely, de- pending upon the experience of the testers, the kind of project, the rate of change in the requirements and the model, and the rate of change in the SUT. To finish this section, let us step back and take a more philosophical view of model-based testing. It is always the case that test design is based on some kind of model of expected behavior, but with manual test design, this model is usually just an informal mental model. By making the model explicit, in a notation that can be used by model-based testing tools, we are able to generate tests automat- ically (which decreases the cost of testing), generate an arbitrary number of tests, as well as obtain more systematic coverage of the model. These changes can increase both the quality and quantity of our test suite.

2.3.1 Is testing a verification or validation activity?

2.4 Formal verification

Throughout this book we use the term verification in the broad sense of checking whether a program or system is consistent with some form of specification. The broad sense of verification includes, for example, inspection techniques and program testing against informally stated specifications. The term **formal verification** is used in the scientific

[3]2 A trademark of Mercury Interactive Corporation; see http:// www.mercury.com.

[4]3 We should also mention the possibility of an error in the model-based testing tools themselves. Of course, this is unlikely, since they are surely well tested!

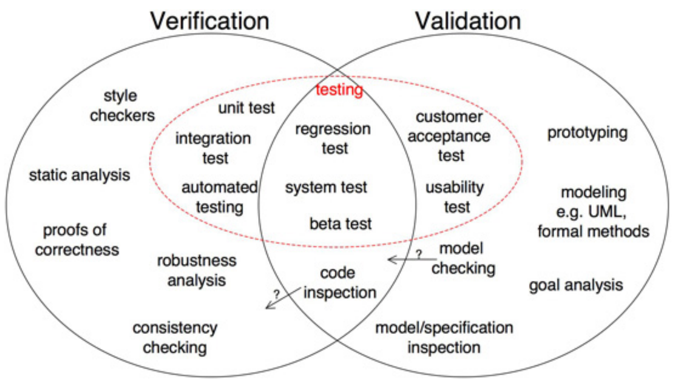

Figure 2.7: Validation and verification techniques

literature in a much narrower sense to denote techniques that construct a mathematical proof of consistency between some formal representation of a program or design and a formal specification.

In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

Formal verification can be helpful in proving the correctness of systems such as: cryptographic protocols, combinational circuits, digital circuits with internal memory, and software expressed as source code.

The verification of these systems is done by providing a formal proof on an abstract mathematical model of the system, the correspondence between the mathematical model and the nature of the system being otherwise known by construction. Examples of mathematical objects often used to model systems are: finite state machines, labelled transition systems, Petri nets, timed automata, hybrid automata, process algebra, formal semantics of programming languages such as operational semantics, denotational semantics, axiomatic semantics and Hoare logic.

Formal specification and verification can help reduce or eliminate bugs, aid in code development, maintenance and extension, and facilitate interoperability and code reuse. While formal specification is widespread in industry, formal verification is most often applied in safety-critical situations (airplanes, cars, medical equipment, nuclear power plants). Researchers continue to try to develop systems to automate verification, and to develop programming methodologies in which proofs of correctness are produced along with programs

2.4.1 Program verification

Current practice is to gather evidence for program correctness by testing – both black-box testing (in which tests are designed independent of the code) and white-box testing (in which tests are designed based on the code). This is analogous to checking that a propositional formula is a theorem by trying a few valuations, or to checking that a predicate formula is a theorem by constructing a few models and interpretations. Exhaustive testing is difficult even for small programs, and impossible in the case where a program can consume an unbounded amount of data.

The process of formal verification starts with the formal description of a specification for a program in some symbolic logic, following that with a proof (in some proof system) that the program meets the formal specification. If the proof system is sound, then this implies that the program meets its specification for all inputs. The question of whether the formal specification conforms to the informal notion of what the program should do is not a technical question, but a social and organizational matter.

Program verification

- Current practice is to gather evidence for program correctness by testing
- However, exhaustive testing is difficult even for small programs
- Testing cannot prove that a program is correct.
- Program verification can prove that a program is correct
  1. starts with the formal description of a specification for a program (It may be implicit, e.g. a null pointer is never dereferenced)
  2. a proof (in some proof system) that the program meets the formal specification.

2.4.2 Runtime verification

Runtime verification is a computing system analysis and execution approach based on extracting information from a running system and using it to detect and possibly react to observed behaviors satisfying or violating certain properties. Some very particular properties, such as datarace and deadlock freedom, are typically desired to be satisfied by all systems and may be best implemented algorithmically. Other properties can be more conveniently captured as formal specifications. Runtime verification specifications are typically expressed in trace predicate formalisms, such as finite state machines, regular expressions, context-free patterns, linear temporal logics, etc., or extensions of these. This allows for a less adhoc approach than normal testing. However, any mechanism for monitoring an executing system is considered runtime verification, including verifying against test oracles and reference implementations. When formal requirements specifications are provided, monitors are synthesized from them and infused within the system by means of instrumentation. Runtime verification can be used for many purposes, such as security or safety policy monitoring, debugging, testing, verification, validation, profiling, fault protection, behavior modification (e.g., recovery), etc. Runtime verification avoids the complexity of traditional formal verification techniques, such as model checking and theorem proving, by analyzing only one or a few execution traces and by working directly with the actual system, thus scaling up relatively well and giving more confidence in the results of the analysis (because it avoids the tedious and error-prone step of formally modelling the system), at the expense of less coverage. Moreover, through its reflective capabilities runtime verification can be made an integral part of the target system, monitoring and guiding its execution during deployment.

2.4.3 Formal model verification

Formal verification is the process of checking whether a design satisfies some requirements (properties). In order to formally verify a design, it must first be converted into a simpler "verifiable" format. The design is specified as a set of interacting systems; each has a finite number of configurations, called states. States and transition between states constitute FSMs. The entire system is an FSM, which can be obtained by composing the FSMs associated with each component. Hence the first step in verification consists of obtaining a complete FSM description of the system. Given a present state (or current configuration), the next state (or successive configuration) of an FSM can be written as a function of its present state and inputs (transition function or transition relation).

We note that this entire framework is one of discrete functions. Discrete functions can be represented conveniently by BDDs (binary decision diagram; a data structure that represents boolean (2-valued) functions) and its extension MDDs (multi-valued decision diagram; a data structure that represents finite valued discrete functions). We use BDDs and MDDs to represent all quantities required in this discrete space (more specifically the transition functions, the inputs, the outputs and the states of the FSMs). For BDDs and MDDs to be efficient representations of discrete functions, a good ordering of input variables (actual inputs, outputs, state) of the functions must be computed. In general, BDDs operate on sets of points rather than individual points; this is called symbolic manipulation.

Example

- A light is initially off. If the user presses a button becomes on if it is off and viceversa.
- Model: the FSM
- Properties
  - if the user never presses the button, the light stays off.
  - whenever the light is off and the user presses the button it becomes on
  - whenever the light is on and the user presses the button it becomes off
- differenze con il testing

2.4.4 Model checking

The most popular method for automatic formal verification is model checking.

[WIKIPEDIA] In computer science, model checking aka property checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. Typically, one has hardware or software systems in mind, whereas the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking is a technique for automatically verifying correctness properties of finite-state systems.

In order to solve such a problem algorithmically, both the model of the system and the specification are formulated in some precise mathematical language: To this end, it is formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is verifying whether a given formula in the propositional logic is satisfied by a given structure.

Property checking is used for verification instead of equivalence checking when two descriptions are not functionally equivalent. Particularly, during refinement, the specification is complemented with the details that are unnecessary in the higher level specification. Yet, there is no need to verify the newly introduced properties against the original specification. It is not even possible. Therefore, the strict bi-directional equivalence check is relaxed to one-way property checking. The implementation or design is regarded a model of the circuit whereas the specifications are properties that the model must satisfy.[1]

An important class of model checking methods have been developed for checking models of hardware and software designs where the specification is given by a temporal logic formula. Pioneering work in the model checking of temporal logic formulae was done by E. M. Clarke and E. A. Emerson[2][3][4] and by J. P. Queille and J. Sifakis.[5] Clarke, Emerson, and Sifakis shared the 2007 Turing Award for their work on model checking.[6][7]

Model checking is most often applied to hardware designs. For software, because of undecidability (see computability theory) the approach cannot be fully algorithmic; typically it may fail to prove or disprove a given property.

The structure is usually given as a source code description in an industrial hardware description language or a special-purpose language. Such a program corresponds to a finite state machine (FSM), i.e., a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node, typically stating which memory elements are one. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.

Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula p, and a structure M with initial state s, decide if $M, s \models p$. If M is finite, as it is in hardware, model checking reduces to a graph search.