Software Testing: Perception on Exploration and Ad-libbing

SANJEEV DHAWAN*, KULVINDER S. HANDA*, RAKESH KUMAR**

*Faculty of Computer Engineering, University Institute of Engineering & Technology (U.I.E.T), Kurukshetra University, Kurukshetra (K.U.K)- 136 119, Haryana, INDIA.
**Faculty of Computer Science, Department of Computer Science and Applications (D.C.S.A), Kurukshetra University, Kurukshetra (K.U.K)- 136 119, Haryana, INDIA.
E-mail: rsdhawan@rediffmail.com

Abstract: - Developing software, which is free from faults, remains one of the most challenging and fundamental problems in software engineering. To realizing the precise need of software architectures, researchers have pursued formal methods, mathematically based notations, techniques, and tools for correctly documenting the software. Despite the availability and potential benefits of formal methods, the use of such methods is still far from the routine practices. There are numerous reasons, involving issues such as the complexity of the languages and tools, lack of expertise, and the existing software development culture. Perhaps most importantly, the use of formal methods is widely considered to be expensive, which prohibits their use for most of the critical systems. As a result of these cost issues, practicing software engineers have largely avoided formal methods, instead relying on testing. Despite the costs associated with identifying good candidate test cases, running the tests, and validating the results, developers rely upon testing as their primary method of ensuring software dependability.

Key-Words: - Software architecture, software testing, test generation, test distribution, test analysis, test reduction

1 Introduction

Testing is a crucial part of the software life cycle, and recent trends evidence the importance of this activity along the whole development process. The testing activities have to start at the requirement specification level and have to be propagated down to the code-level, all along the various subsequent refinement steps. Testing involves several demanding tasks: the ability to launch the selected tests (in a controlled host environment, or worse in the tight target environment of an embedded system); deciding whether the test outcome is acceptable or not (which is referred to as the test oracle problem). Therefore, the impacts of failure cost in direct cause (the fault), and the indirect one (root cause analysis). However, the problem that has received the highest attention in the literature is to select an appropriate test case. In brief, how to identify a suite of test cases that is effective in demonstrating that the software behaves as intended, or, otherwise, in evidencing the existing malfunctions. Clearly, a good test suite is in fact the crucial starting point to a successful testing session. In contrast with the conventional practice of handcrafted ad-hoc test cases, or of random input generation, many methods for systematic test selection have been proposed in the past decades. No method is superior to the others, thus several methods should be used in combination throughout the lifecycle, with focus shifting, as development proceeds, on differing aspects of software behavior, and also on differing projections of the system. The term model-based testing refers to test case derivation from a model representing the software behavior. Indeed, testing is always against an expected behavior: the difference being essentially whether such a model is explicit (which is clearly better), or implicit, i.e., in the mind of the testers. In particular, when there exists a specification of the system to be tested in some formal language, this can be used as the reference model both for test-case selection and as a test oracle. This allows for rigorous mathematical analysis, and automated processing. Testing an implementation against its formal specifications is also known as conformance testing, which, looking at the big picture of test strategies belongs to the black box class, because we do not consider the internals of a system, but only its input/output behavior.

After the test cases are derived from the specifications, two major problems remain to be solved: traceability and test execution. Traceability concerns “relating the abstract values of the specification to the concrete values of the implementation”. To be able to execute these tests on the code, we need to refine the test cases into more concrete sequences that have a meaningful interpretation in terms of the actual system I/O interface. Test execution entails forcing the Implementation Under Test (IUT) to execute the specific sequence of events that has been selected. A problem rises with concurrent programs which, starting from the same input, may exercise different sequences of interactions (among several concurrent processes) and produce different results. This problem has already been analyzed in the literature, and deterministic- and non-deterministic-testing approaches have been proposed. In non-deterministic testing [1], the approach is to repeat the launching of a program run under some specified input conditions several times until the desired test sequence is observed (or a maximum number of iterations are reached). In contrast, the deterministic testing approach forces a program to execute a specified test sequence by instrumenting it with synchronization constructs that deterministically reproduce the desired sequence.

2 Software Architecture and Testing

Software architecture (SA) represents the most promising approach to tackle the problem of scaling up in software engineering, because, through suitable abstractions, it provides the way to make large applications manageable. Nowadays, SA descriptions are commonly integrated into the software development process; SA production and management are, in general, quite expensive tasks. Therefore the effort is worthwhile if the SA artifacts are extensively used for multiple purposes. Typical use of SA is as a high-level design blueprint of the system to be used during system development and later on for maintenance and reuse. In particular, the importance of the role of SA in testing and analysis is evident. SA formal dynamic descriptions are used for many different kinds of analysis. We are here interested in SA primarily as a means for driving the testing of large, complex systems. Our concern is on exploiting the information described at the SA level to drive the testing of the implementation. How formal SA descriptions (and the obtained models) can be used for testing purposes [2]. In other words, we assume the SA description is correct and we are investigating approaches to specification-based integration and system testing [3], whereby the reference model used to generate the test cases is the SA description [4][2]. Figure shown below provides a useful hierarchical decomposition of different testing techniques and their relationship to different classes of test adequacy criteria.

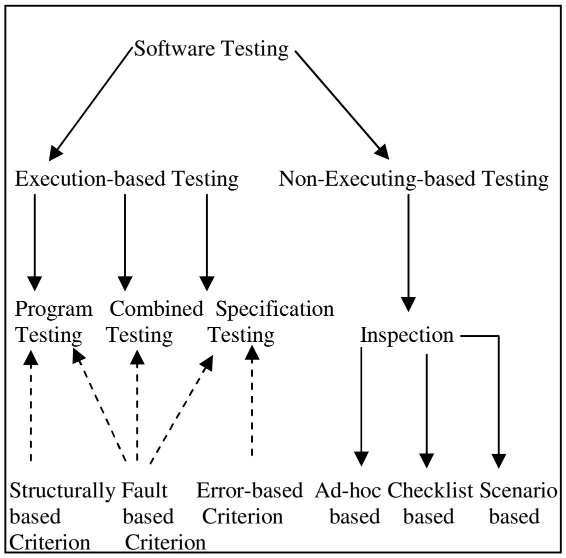

Fig. 1. Shows the hierarchical decomposition of different testing techniques

2.1 Fault Detection

This is something that is quite beyond most current testing methods. The claim “the system/component is fault-free” is quite beyond current testing methods. In practice all we can usually say is that we have uncovered a number of faults over a period of testing effort and the graph of the number of faults against the period or amount of testing, measured suitably, indicates that the growth rate is reducing. Figure 2 shows the relationship between fault detection and test time. The trouble is we do not know that no further faults are in the system at any particular time. Also, in general we cannot assert that the only faults remaining are located in a specific module or component. A general formula for this curve is not known, if one existed it would probably depend on the type of system, on the type of test methods and perhaps on the people doing and managing the testing as well as wider issues relating to the management of the design project, the attitudes of the clients, the implementation vehicle, the design methods and so on.

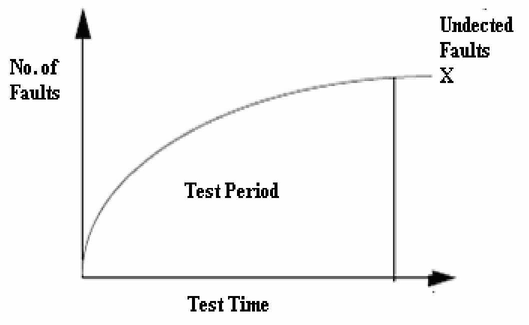

Fig. 2. Shows the relationship between fault detection and test time

High quality empirical results obtained over a very long period would be needed to make proper use of this approach even then it is less than ideal.

2.2 Effectiveness of Test

The problem of measuring the effectiveness of an individual testing project usually depends on estimating the coverage of the tests [5]. For example, if the tests are structurally based, using program charts, say, then popular methods include establishing that every path has been exercised or every decision node has been visited. This does not tell us anything about fault detection; it merely measures effort rather than reward! The current accepted definition of fault coverage is misleading since it is not the case that a precise measure can be placed on the number of faults that remain after the test process has been applied. In most cases the definition of fault coverage is based on assumptions of the type described above, that we are questioning. In much of the literature the estimates of the fault coverage are obtained by running experiments with simple examples involving implementations that have faults seeded in them and counting the numbers of known faults detected by the methods. These empirical results are of curiosity value only. Miller & Paul provide a theoretical method for establishing fault coverage of a test strategy; however, they assume that the implementation machine has the same number of states as the specification machine. Before we look, briefly, at what progress there has been in addressing some of the theoretical issues of testing we will consider a popular method for the analysis and comparison of the effectiveness of different testing methods.

2.3 Test Method Effectiveness

A number of authors have sought to compare the effectiveness of, for example, random testing methods with formally based functional testing [1]. Here the method was to take a small system and to insert known faults into it, then to apply the two techniques to establish which was most successful at detecting these faults. Further analysis could be done on the type of faults each method was good or poor at detecting. Faults were classified for this purpose in a number of categories. Results from this type of survey can be useful in establishing the relative strengths and weakness of different, specific methods of test set generation. However, the situation is essentially artificial and it is not clear what can be said in general. The method is unable to prove, for example, that either approach is better at detecting naturally occurring or unseeded faults (the seeded ones may not be typical of real faults), or to identify conditions under which a method detects all faults. A number of attempts at developing a theory of testing or to analyze the testing situation have been made. We will briefly consider few of them as follows:

2.3.1 An Algebraic Approach to Computational Modeling

The process of software design, including within that activity all phases of requirements capture, specification, design, prototyping, analysis, implementation, validation, verification and maintenance is one that is oriented, or should be, around the construction of computational solutions to specific problems. When we are constructing a software system (this also applies to hardware) we are attempting to construct something that will, when operating, carry out some computable function. Consequently it is worth considering what this means. Essentially, computable functions have been identified as the functions computed by Turing machines. The method will not be applicable to implementations that behave like a Turing machine that does not halt. In other words we will not try to deal with those systems that regress into an infinite loop from which no output emanates, for our purposes these systems will be deemed to be unacceptable anyway. A way to establish that a system is not of this form is to identify a period of time, which is the maximum that the system can run for without producing any detectable output. We will also assume that the specification of the system is also of this form, namely a Turing machine that halts under its intended operating conditions. Real-time systems are covered by this definition since we require that the specified system does have detectable behavior under all conditions. This is a kind of design for test condition that we will see more of later. We then have two algebraic objects, the Turing machine representing the specification of the desired system and the Turing machine representing the complete implementation. A testing method would then try to ascertain if these two machines computed the same function.

This is a basic strategy that we will develop, however, not in the context of a Turing machine, which is too low level and unwieldy, but in the context of a more useful, elegant and equivalent model. In so doing we will quote some important theoretical results that justify what we are doing. It is important to stress that the method of finite state machine testing proposed by Chow, and developed by a number of other authors is based on a similar sort of philosophy, the difference being that they have to make very strong assumptions about the nature of the implementation machine. However, their work did act as an important inspiration for our own platform testing for the finite state machines.

2.3.2 Model-based Testing

Simply put, a model of software is a depiction of its behavior. Behavior can be described in terms of the input sequences accepted by the system, the actions, conditions, and output logic, or the flow of data through the application's modules and routines. In order for a model to be useful for groups of testers and for multiple testing tasks, it needs to be taken out of the mind of those who understand what the software is supposed to accomplish and written down in an easily understandable form. It is also generally preferable that a model be as formal as it is practical. With these properties, the model becomes a shareable, reusable, precise description of the system under test. There are numerous such models, and each describes different aspects of software behavior. For example, control flow; data flow, and program dependency graphs express how the implementation behaves by representing its source code structure. Decision tables and state machines [6], on the other hand, are used to describe external so-called black box behavior. When we speak of MBT, the testing community today tends to think in terms of such black box models.

2.3.3 Differential Testing

Differential testing addresses a specific problem—the cost of evaluating test results. Every test yields some result. If a single test is fed to several comparable programs (for example, several C compilers), and one program gives a different result, a bug may have been exposed. For usable software, very few generated tests will result in differences. Because it is feasible to generate millions of tests, even a few differences can result in a substantial stream of detected bugs. The trade-off is to use many computer cycles instead of human effort to design and evaluate tests. Particle physicists use the same paradigm: they examine millions of mostly boring events to find a few high-interest particle interactions. Several issues must be addressed to make differential testing effective [4]. The first issue concerns the quality of the test. Any random string fed to a C compiler yields some result—most likely a diagnostic. Feeding random strings to the compiler soon becomes unproductive, however, because these tests provide only shallow coverage of the compiler logic. Developers must devise tests that drive deep into the tested compiler. The second issue relates to false positives.
The results of two tested programs may differ and yet still be correct, depending on the requirements. Similarly, even for required diagnostics, the form of the diagnostic is unspecified and therefore difficult to compare across systems. The third issue deals with the amount of noise in the generated test case. Given a successful random test, there is likely to be a much shorter test that exposes the same bug. The developer who is seeking to fix the bug strongly prefers to use the shorter test. The fourth issue concerns comparing programs that must run on different platforms. Differential testing is easily adapted to distributed testing.

3 Need of Generating Tests

The difficulty of generating tests from a model depends on the nature of the model. Models that are useful for testing usually possess properties that make test generation effortless and, frequently, automatable. For some models, all that is required is to go through combinations of conditions described in the model, requiring simple knowledge of combinatorics. In the case of finite state machines, it is as simple as implementing an algorithm that randomly traverses the state transition diagram. The sequences of arc labels along the generated paths are, by definition, tests. For example, in the state transition diagram below, the sequence of inputs "a, b, d, e, f, i, j, k" qualifies as a test of the represented system.

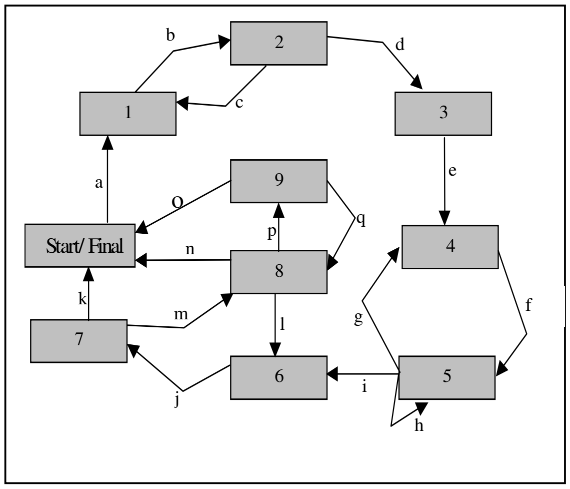

Fig. (3) shows the state transition diagram

There are a variety of constraints on what constitutes a path to meet the criteria for tests. Examples include having the path start and end in the starting state, restricting the number of loops or cycles in a path, and restricting the states that a path can visit. While writing the automation code, adherence to good engineering practices is required. Scripts are bound to interact with each other and evolve as the software evolves. Scripts can be used for as long as the software is being tested, so it worth while investing some time in writing good, efficient ones. With model-based testing, the number of simulation routines is in the order of the number of inputs, so they are generally not too time-consuming to write [7].

4 Test Distribution

Each tested or comparison program must be executed where it is supported. This may mean different hardware, operating system, and even physical location. There are numerous ways to utilize a network to distribute tests and then gather the results. One particularly simple way is to use continuously running watcher programs. Each watcher program periodically examines a common file system for the existence of some particular files upon which the program can act. If no files exist, the watcher program sleeps for a while and tries again. On most operating systems, watcher programs can be implemented as command scripts. There is a test master and a number of test beds. The test master generates the test cases, assigns them to the test beds, and later analyzes the results. Each test bed runs its assigned tests. The test master and test beds share a file space, perhaps via a network. For each test bed there is a test input directory and a test output directory. A watcher program called the test driver waits until all the (possibly remote) test input directories are empty. The test driver then writes its latest generated test case into each of the test input directories and returns to its watch-sleep cycle. For each test bed there is a test watcher program that waits until there is a file in its test input directory. When a test watcher finds a file to test, the test watcher runs the new test, puts the results in its test output directory, and returns to the watch-sleep cycle.

Another watcher program called the test analyzer waits until all the test output directories contain results. Then the results, both input and output, are collected for analysis, and all the files are deleted from every test input and output directory, thus enabling another cycle to begin. Using the file system for synchronization is adequate for computations on the scale of a compile-and-execute sequence. Because of the many sleep periods, this distribution system runs efficiently but not fast. If throughput becomes a problem, the test system designer can provide more sophisticated remote execution. The distribution solution as described is neither robust against crashes and loops nor easy to start. It is possible to elaborate the watcher programs to respond to a reasonable number of additional requirements.

5 Test Analysis

The test analyzer can compare the output in various ways. The goal is to discover likely bugs in the compiler under test. The initial step is to distinguish the test results by failure category, using corresponding directories to hold the results. If the compiler under test crashes, the test analyzer writes the test data to the crash directory. If the compiler under test enters an endless loop, the test analyzer writes t he test data to the loop directory. If one of the comparison compilers crashes or enters an endless loop, the test analyzer discards the test, since reporting the bugs of a comparison compiler is not a testing objective. If some, but not all, of the test case executions terminate abnormally, the test case is written to the ABEND directory. If all the test cases run to completion but the output differs, the case is written to the test diff directory. Otherwise, the test case is discarded.

6 Test Reduction

A tester must examine each filed test case to determine if it exposes a fault in the compiler under test. The first step is to reduce the test to the shortest version that qualifies for examination. A watcher called the crash analyzer examines the crash directory for files and moves found files to a working directory. The crash analyzer then applies a shortening transformation to the source of the test case and reruns the test. If the compiler under test still crashes, the original test case is replaced by the shortened test case. Otherwise, the change is backed out output, are collected for analysis, and all the files are deleted from every test input and output directory, thus enabling another cycle to begin. Using the file system for synchronization is adequate for computations on the scale of a compile-and-execute sequence. Because of the many sleep periods, this distribution system runs efficiently but not fast. If throughput becomes a problem, the test system designer can provide more sophisticated remote execution. The distribution solution as described is neither robust against crashes and loops nor easy to start. It is possible to elaborate the watcher programs to respond to a reasonable number of additional requirements.

7 Generator of Test Data

Testing the functional requirements of the software, i.e. the relationship between input and output, to check non-functional requirements like temporal constraints and a test adequacy criterion. There exist many of them. For example, in the statement coverage we require all the statements in the program to be executed. On the other hand, branch coverage requires taking all the branches in the conditional statements. The same test adequacy criterion is taken in condition-decision coverage. To fulfill this criterion all conditions must be true and false at least once after executing all the set of test data on it. A condition is an expression that is evaluated during the program execution to a Boolean value (true or false) with no other nested conditions. All the comparison expressions are conditions. On the contrary, a decision is a Boolean expression whose value affects the control flow. It is important to note that full condition-decision coverage implies full branch coverage but not vice versa. That is, if we find a set of test inputs that makes true and false all the program conditions at least once we can ensure that all the decisions will take values true and false and, in consequence, that all branches will be taken; but taking all branches does not ensure that all conditions take the two Boolean values.

8 Conclusions and Predictions

Testing is an important technique for the improvement and measurement of a software system's quality. Any approach to testing software faces essential and accidental difficulties. While software testing is not an elixir that can guarantee the production of high quality applications. However, the theoretical and empirical investigations have shown that the rigorous, consistent, and intelligent application of testing techniques can improve software quality. Software testing normally involves the stages of test case specification, test case generation, test execution, test adequacy evaluation, and regression testing. Each of these stages in our model of the software testing process plays an important role in the production of programs that meet their intended specification. The body of theoretical and practical knowledge about software testing continues to grow as research expands the applicability of existing techniques and proposes new testing techniques for an ever-widening range of programming languages and application domains [8].

References:

[1] Kaner, Cem, Jack Falk, and Hung Nguyen, Testing Computer Software, 2nd Edition, John Wiley & Sons, 1999.

[2] Bertolino, A., Corradini, F., Inverardi, P., Muccini, H.: Deriving Test Plans from Architectural Descriptions. In ACM Proc. Int. Conf. on Software Engineering (ICSE2000), pp. 220-229, June 2000.

[3] W.J. Gutjahr, "Importance Sampling of Test Cases in Markovian Software Usage Models," Probability in the Engineering and Information Sciences, v. 11, 1997, pp.19-36.

[4] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2004), Boston, 11–14 July 2004. IEEE.

[5] Bach, James, General Functionality and Stability Test Procedure, http://www.testingcraft.com/.

[6] Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, A Test Sequence Selection Method for Statecharts. The Journal of Software Testing, Verification & Reliability, 10(4): 203-227, December 2000.

[7] David Coppit and Jennifer M. Haddox-Schatz. On the use of specification-based assertions as test oracles. Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop, Maryland, 6–7 April 2005. IEEE.

[8] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 00), York, UK, October 2000.