Verification: Testing

Validation and verification activities fall into two broad categories:

Dynamic: These activities focus on the execution behaviour of the (partially developed) system. Either by simulating or by executing components of the code in suitable test harnesses, the aim is to use the behaviour of the system in its validation or verification. Often simulations are used early in the development process to validate design decisions while later in the process test strategies attempt to detect defects in the real code of the system.

Static: These activities focus on the text of the system and aim to discover defects by subjecting the text to scrutiny. Static validation and verification can range from the process of code reviews and walkthroughs where a group including the author of the code examine it in detail, to the mathematical analysis of the code in an attempt to show that formally expressed properties hold of the system.

The topic of static techniques (including a discussion of verification in the context of Extended ML) is covered in the CS3 module *Functional Programming and Specification*. Here we note that the main methods are various manual program inspections and reviews, static program analysis tools (e.g. the ML typechecker and exhaustiveness checker) which check that code has desirable properties, and a variety of mathematical techniques, usually going under the name "formal methods". SEOC1, and this note, concentrates on dynamic methods and testing in particular as a means of validating and verifying systems.

An Overview of Testing

The aim of testing is to reveal *defects* in the system. A defect can relate either to validation or verification concerns. The limitations of testing are well known: testing can only reveal the *presence* of defects, not their *absence*. The absence of defects can only be established by static means. When a test has been performed, it is not always easy to determine if the outcome is correct or not, perhaps because the correct answer is not explicit in the specification or because the failure is subtle: this is called the *oracle problem*.

Categories of Testing

Testing can be grouped into three broad categories depending on the entity and characteristics being tested:

Acceptance: This level of testing is entirely concerned with validating the design — therefore acceptance testing should not be deferred to the time when the final system is delivered. Prototypes, simulations etc. can be subject to acceptance testing early in the development process. Then the outcome of early acceptance testing is a requirement that the delivered system conform to the prototype or simulation. Such requirements require verification testing as the design proceeds.

Integration: These are tests at the module, sub-system and system level. As we move to higher levels of integration the purpose of testing moves from predominantly verification to validation as we approach the full system. If we think about this process taking place in the ML modules system then as we integrate the system we will generate richer and richer structures, but we will also know the signature we intend each structure to match — thus the integration tests can be linked to signatures which will be generated during design. Tests can be written as functors which take a structure with appropriate signature as argument.

Unit: This type of testing is oriented to discovering defects in the individual procedures and functions making up the system. Because it involves small components deeply embedded within the system, this is mainly a verification concern. Good practice suggests that unit tests should be constructed by someone other than the particular unit's author.

Test Planning

The process of planning system testing proceeds as the design is developed:

Acceptance test plan: this should be generated as the requirements are determined. It concentrates on ensuring that the acceptance test covers all system requirements. The conceptual design of the system will also play a rôle in deciding on the acceptance tests.

System integration test plan: this is generated from the technical design specification and a high-level characterisation of the system design.

Module and unit tests: these are developed during detailed design.

Sub-system integration test plan: this utilises some low-level aspects of the system design and the detailed implementation design.

The outcome of such planning is the set of tests that will be applied to the system as it is developed and integrated.

Test Strategies

There are two broad types of test which we can apply to simple sequential systems (concurrent systems require more testing relating to synchronisation). These are:

Behavioural testing: focuses on testing the functionality that the system is intended to provide, according to both its requirements and its specification. Testing can proceed with or without knowledge of the actual code. There are numerous approaches to behavioural testing and this note presents an overview of major techniques in the section 'Revealing Defects.'

Stress testing: all implementations of systems have resource limits which complicate the behaviour of the system and may render it inoperative if they are exceeded and the programmer takes no account of the possibility of such failure. Typical limitations are: storage, compute time, communication capacity, database retrieval time. Stress testing should be designed to drive the system to the point where these limits are reached. Such tests often reveal the omission of features in the implementation designed to deal with such limitations; they also often turn up potential defects which could result in catastrophic failure (e.g. large scale loss of data or service).

Testing Tools

A wide range of testing tools are available. The most common are some form of *profiling* the execution of a program on test input — see the Solaris manual page on **tcov**. Such tools give an idea of when statements have been executed in the course of testing. Along with structural testing, they can provide evidence that all possible paths in the code have been tested.

For some languages (e.g. Ada) the process of test path generation is semi-automated and one can generate the data required to do path testing from the source code of the program. For other popular languages, including Java, there are numerous commercially produced tools. Other useful tools include file comparators (e.g. **vdiff** in Solaris) to check long output files for differences, simulators to provide testing environments and debugging environments (e.g. **dbx** or **ups** in Solaris) which provide some means to locate defects.

Debugging

Once a defect has been revealed, the process of *debugging* attempts to:

locate the defect in the design or implementation. This is far from a trivial problem — it may be possible to locate the defect in different parts of the design. For example, the defect may relate to validation if a real requirement is not met and the implementor has correctly implemented an erroneous requirement, or it may relate to verification if the requirement is correct but the implementor has failed to correctly implement it.

re-design that part of the system to eliminate the defect. This may require major reconsideration of the overall design and, in practice, can often result in the introduction of new defects which were not present in the original design. The re-design process is crucial to the effectiveness of testing and, if it is not undertaken with full consideration of the component's intended behaviour, it will often lead to the introduction of serious defects.

re-implement the revised design.

re-test the component with *all tests for that component*. This is an important part of debugging: the component in which the fault has been eliminated must be completely tested in order that defects which might have been introduced can be revealed. Of course this is not a foolproof approach because the new defects may not be revealed by old tests — but at least we know that the new component passes more tests than the old one.

Testing: Revealing Defects

Defect testing is aimed at revealing the presence of errors in a system. Of course, although testing may give us a strong feeling that the code is correct, it cannot confirm the absence of errors because we can only have explored a small fraction of the whole space of potential inputs to the system. There are techniques which allow us to test whole classes of inputs e.g. formal proof of correctness or symbolic evaluation. However, most of the time we must settle for evaluation on specific inputs and attempt to justify the generalisation of a small number of test cases to support a belief that the system is defect free.

In designing tests to uncover defects the following broad guidance should be followed:

- Testing should be aimed at capabilities rather than components. The decision to test some unit in a particular should be motivated by its interest in providing some capability to the user rather than it being of some interest to the implementation. For instance, testing should be organised so as to cover all *use-cases* for the system.
- If the system is being revised or maintained, concentrate on the existing functionality (which the users expect to remain unchanged) since maintaining its performance is more important than ensuring flawless performance of the new components.
- Concentrate mainly on typical data rather than extremes and boundary conditions — in use these boundary conditions almost never occur whereas performance on typical data should be as good as possible.

The rest of this note covers three popular techniques used in defect testing. The first two, black-box and glass-box testing, can be used at any stage in testing but black-box approaches are probably more suitable for system tests while glass-box is appropriate for unit testing. The third approach is interface testing which is commonly used in integration testing to check the interaction between different modules. Finally, we also consider an approach to determining the efficiency of testing: residual defect estimation.

Functional (or "Black Box") Testing

Here the test cases are derived from the specification of the system or module which is to be tested. The theoretical basis for black-box testing is known as *equivalence partitioning.* Because any system is defined finitely by some collection of programs the claim is that the space of possible inputs can be partitioned[1] into a finite number of *equivalence partitions.* Two members of the same equivalence partition result in essentially the "same" output. The notion of "same" output is also defined by equivalence partitions. The output space of the system is also partitioned into some finite collection of equivalence partitions. Two inputs in the same input equivalence partition should result in outputs which are in the same output equivalence partition. A specification describes how input partitions relate to output partitions. For example, suppose we are given a structure X, which matches the signature:

$$
\begin{array}{lll}
\mathit{splitSig} & = & \textbf{sig} \\
 & & \quad \begin{array}{ll} \textbf{type} & \mathit{object} \\ \textbf{val} & \mathit{split}\text{: } (\mathit{object} \rightarrow \mathit{bool}) \rightarrow \mathit{object\ list} \rightarrow (\mathit{object\ list} * \mathit{object\ list}) \end{array} \\
 & & \textbf{end}
\end{array}
$$

and we know that $\mathit{split}\, p\, l$ should have as result a pair (l', l'') where l' contains elements, a, of l such that $p\,a$ is false, and l'' those for which p is true. Then how do we go about testing X to check that it is reasonably free of functional defects? The conventional approach to black-box testing requires the identification of *equivalence partitions* on the space of possible inputs. An equivalence partition is a grouping of inputs which all generate the same (or at least similar) outputs. For our small example the partitions might be:

1. any p, $l = []$
2. $p\,a = false$ for every a, any l
3. $p\,a = true$ for every a, any l
4. l, p such that there are a, a' in l for which $p\,a = \mathit{false}$ and $p\,a' = \mathit{true}$

These four partitions of the inputs to *split* appear to cover the significantly different outcomes of a call of the function.

A black box test set should consist of at least two examples from each input partition. Such a test set will demonstrate that items in the same input equivalence partition end up in the same output partition and that for each input partition the correct output partition is derived. More tests will give further confidence but there is a law of diminishing returns. Additional successful tests provide some extra confidence — but to obtain a statistically significant result, one would require thousands of tests.

The deficiency of the approach is that most reasonably efficient implementations work with much finer partitions than are discernible from the specification. It may also be the case that the specification is inadequate to determine the correspondence between input and output equivalence partitions.

[1] If S is a set, then $S_1, \ldots, S_n$ is a *partition* of S iff $S_i \cap S_j = \emptyset$ for all $i \neq j$ and $S_1 \cup \ldots \cup S_n = S$.

Functional Testing Strategies

We need to test the emerging system as we build it rather than waiting until it is finished. The problem is that the system is incomplete but still we would like to test what has been done so far. There is a choice of strategy here depending on the design methodology and the character of the system:

Top-down testing: this involves testing components when the modules they depend on may as yet be incomplete. In terms of ML modules we would carry out testing of functors before the structures they should be applied to are complete. This would require the writing of dummy structures having the appropriate signatures so we can apply the functor an obtain a testable functor. The dummy structures ("*stubs*") can adopt a variety of strategies varying from just failing when they are called to querying the tester for appropriate values.

Bottom-up testing: in this case we test modules when their operational environment is not yet fully designed. This requires the construction of "*drivers*" which exercise the code. In ML terms, we must write functors which take the structures to be tested as arguments and provide some approximation to the real operational environment. In general drivers try to simulate the operational environment in which the software will be used. In this case issues like timing and resource use may be important in the design of the driver.

A serious disadvantage of bottom-up testing is that the top-level components are usually the most important to the functionality of the system but they are the last to be tested. This means that the discovery of major faults tends to be postponed.

Structural (or "Glass Box") Testing

Structural testing is also called *glass box* testing because in this process the tester is permitted to look at the code rather than just its specification. In this form of testing the emphasis is on ensuring that all the code in a function has been exercised, and, if possible, all possible paths through the function have been followed. The tester uses the code of the function to generate test cases to force the code under test through a particular sequence of steps and may use tools like test coverage monitors to check the level of coverage. For simple imperative languages the approach to this form of testing is to construct the flow graph for the program and then generate a set of *independent* paths through the flowgraph in which each block of straight line code appears at least once. A path p is independent from a set S of paths if it contains an edge not included in any other path in S.

This form of testing complements and refines black-box testing since seeing the structure of the code allows the tester to refine the equivalence partitions on the input. However, black-box testing should be undertaken first since it allows the tester to be independent of the implementation choices made in the module.

In structural testing the text of the program is used to give a measure of the adequacy of the test set. This has two consequences: the test set is closely matched to the implementation rather than to the specification; and there is no need to refer to the specification to generate the test set (although it is necessary to look at the specification to see if the results are correct).

A variety of adequacy criteria have been proposed in the literature. The most commonly used is the "all paths" criterion where a set of tests is judged to be adequate if each path in the program text is explored by at least one test. In the case of functional languages this means that the set of tests exercises each case in the top level function and that for each case tested then all the possible cases in the functions called by the top level function are exercised.

In general it is impossible to check whether a particular condition can ever be true. This means it is impossible to guarantee complete coverage in structural testing because some piece of code may only be accessible in circumstances which cannot be determined from the code of the program. However, in practice the situation is often much easier and structural testing can yield good results.

Two broad classes of tools are used in support of structural testing. These are:

Data Flow Analysis: These provide a static analysis of the possible paths in a program and in some cases can generate conditions on the data values which cause a given path to be followed (in simple circumstances it is possible to calculate values which cause a path to be followed). The techniques used by these tools are well developed and mostly derive from tools used in optimising compilers.

Test Coverage: These are monitoring tools which provide a trace of the paths executed in a program when it is exercised on a particular set of tests. These tools provide a useful empirical measure of the adequacy of a particular test set and may suggest how to improve the test set. A typical example is `tcov`, a C test coverage tool — see the Solaris manual page for details.

Checking coverage can be automated because structural testing requires no knowledge of what a program is supposed to do (except when deciding if the result is correct), only its structure. One reasonable strategy is to first do black box testing, then check coverage of those tests, then add more tests to complete the coverage.

Interface Testing

During integration testing we often want to see the pattern of interaction between two modules. Depending on the type of system we are dealing with the protocol governing the interaction between two modules may be quite different. For example, it could be message passing, some kind of protocol stack or, as it is in the case of simple ML modules, function call. The aim in interface testing is to exercise all the possible patterns of interaction across a particular interface. In general this can be quite difficult. In some languages it is possible to specify the interaction across interfaces.

Residual Defect Estimation

This is a technique which is commonly used to explore the efficacy of the test suite for a system. The system is seeded with a number of "common" errors and the test suite is then run on that version of the system. The number of detected defects is then used to provide evidence of the efficacy of the tests and hence provide evidence of how many errors remain in the real system. The basis of this method is rather dubious but it does provide some indication of the "efficiency" of a test suite in detecting errors.

and finally...

This note is mostly empirical in nature, suggesting some approaches to testing and how we can test modular programs. Testing remains an area in which it seems difficult to take a general approach: the specifics of a particular program and its proposed environment are critical in determining the test programme.