Determining Component Reliability Using a Testing Index

John Morris[a], Peng Lam[b], Gareth Lee[a], Kris Parker[a], Gary A Bundell[a]

[a] Centre for Intelligent Information Processing Systems,
Department of Electrical and Electronic Engineering,
The University of Western Australia,
Nedlands WA 6907, Australia
email: [bundell,gareth,morris,kaypy]@ee.uwa.edu.au

[b] School of Engineering,
Murdoch University, Murdoch WA 6150, Australia
email: peng@eng.murdoch.edu.au

Abstract

Component-Based Software Engineering has the potential to provide reliable systems based on tested components quickly and economically, but these systems will only be as reliable as the components from which they are constructed. We propose a 6-point scale which can be used to rate the degree to which a component has been tested. This scale can be used by developers to assess the risk of using a third party component. Since a variety of test strategies are used, it is necessary to correlate testing strategies with our scale. In this paper, we examine the testing strategies specified in British Standard 7925-2 and show how they relate to the reliability levels that we propose. Since well-behaved use of resources is also a key factor in overall system reliability, we propose that an 'R' tag be added to the rated level when resource usage has been verified to be within reasonable bounds.

Keywords: Component Testing, Component-Based Software Engineering, Software Reliability

1 Introduction

Component-Based Software Engineering (CBSE) is an emerging methodology for software development that aims to compose applications with plug and play software components (custom-built or Commercial Off-the-Shelf) in a framework. This paradigm is becoming increasingly important owing to the maturity of several underlying technologies that support building components and developing applications from sets of these components. CBSE assumes a source of reliable components: productivity gains depend not only on time saved in re-use but on the ability to omit unit testing from the system lifecycle and reliability gains depend on the degree to which units (components) may be trusted. If some parts of a system are not under the direct control of system developers (*i.e.* not written by them) then they need to have a means of generating a risk model for the use of the system containing the unknown parts.

Although there is no shortage of software engineering standards - Fenton and Neil report discovering over 250 [Fenton and Neil, 1998] - they do not provide much assistance; they are primarily process standards [Pfleeger et al., 1994, Fenton and Neil, 1998]. For example, Fenton and Neil [Fenton and Neil, 1998] compared BS 4792 - a standard for pushchairs[1] with a defence software safety standard (DEF-STAN 00-55) [Defence, 1997]. BS 4792's 28 requirements were entirely product requirements and 11 of those were external product requirements. In contrast, DEF-STAN 00-55 has 115 requirements - 88 of which are process requirements, the remainder are internal product and resource requirements - *without a single external product requirement.* The British Standard BS 7925-2 [Institution, 1998] prescribes techniques for test case generation, but is still primarily a process standard as it allows a tester to choose one of 13 strategies for developing tests. It even provides an 'other technique' section to allow a tester to use an unspecified technique appropriate to the software being tested and still comply with the standard. Statement coverage appears to be the only testing strategy that is prescribed in any other standard [Defence, 1997, RTCA, 1992, ANSI/IEEE, 1987] but its limitations are well known.

To assess the risk of using a third party component, a software developer needs a reliability index - a value that can be inserted into a risk model for a complete system. The testing levels that we propose in this paper are designed to provide this. Our scale recognizes that testing is expensive and that various criteria (economic/time/*etc.*) will be used to determine what level of testing is justified for any component.

Rapps and Weyuker have classified testing strategies based on the paths taken through a section of code [Rapps and Weyuker, 1985]. Their classification examined paths from *definitions* of variables to *uses*. They prove various subsumption relationships which rank testing strategies in their family of strategies according to their relative strengths. However this work covers only a very narrow band of strategies: we believe that a wider, standard scale - that can be applied to a component tested by any strategy - is needed.

Many other reliability metrics have been proposed which are based on testing, *e.g.* fault discovery rates [Weerahandi and Hausman, 1994]. Most of these metrics are appropriate for large systems where the number of required tests preclude fully testing to any criterion. In this work, we focus on components, which will in general be much smaller units of code to which it is practical to achieve 100% testing with some chosen strategy.

Note that in this paper, we adopt a very broad definition for the term 'component': it encompasses everything in the spectrum from 'pure' dataflow functions to components with all the capabilities associated with the term [Bundell et al., 2000a]. As long

Copyright ©2001, Australian Computer Society, Inc. This paper appeared at the Twenty-Fifth Australasian Computer Science Conference (ACSC2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

[1] Previously known as 'perambulators' but more recently as 'baby strollers' in some countries that claim to use English as the primary language.

as there is a clearly defined interface to a component (something that can be quite difficult in light of polymorphism and exception handling), a test specification may be assembled for it and a meaningful reliability level from our table may be assigned to it. We also adopt the term 'method' here with the meaning normally ascribed to it when describing object-oriented systems. However, our discussions and rankings apply equally well to any procedure, function or subroutine in a procedural language. In object-oriented systems, we make no distinctions betweeen methods, constructors and destructors and use the term method to encompass all.

1.1 Outline

In section 2, we introduce our testing level table and explain how a software system would acquire a 'rating' at the various levels in the table. Section 3 reviews BS 7925-2 - focussing on the test strategies permitted by it. Section 4 attempts to correlate all the techniques described in BS7925 and show how they are related to each other and to the proposed reliability levels. Section 5 discusses some of the issues associated with using our levels.

2 Reliability Ranking of Components and Testing Levels

The degree to which a component has been tested is a good measure of the risk which a user may be taking when using that component. We define 'reliability' as the degree of proof that an implementation adheres to its specification [2]. This generates a metric for the risk associated with using a component in a system. Thus we have ranked testing strategies and assigned a numerical level to them. Table 1 sets out the testing strategies and the rank assigned to each. To keep the table simple and easy to use, seven basic levels were assigned values from 0 to 6 - with 0 representing no testing at all and 6 representing code that was derived by formal transformation from the specification.

In practice, this means that tested components are assigned a rating from 1 to 5 - levels below 1 are of no interest and levels above 5 are only achievable with intensive effort that would rarely be applied for software that was not life critical. To accommodate variations in testing procedures, partial compliance and insertion of new strategies, levels of verification are expressed as real numbers.

Level 6 represents a fundamentally different approach to the software production process: requirements expressed in some formal notation are transformed by provably correct steps into the final code. Thus it does not formally involve testing at all and differs from our other ratings which involve progressively more stringent testing. However there is a need to recognise this type of component - some system requirements will demand components built to this standard - and therefore a level is assigned for it.

We also recognized that it is necessary to verify that a component is well-behaved with respect to resources that it uses. This requirement is orthogonal to tests which concern themselves with correctness of output only; it can be applied at any level of testing. We propose attaching an 'R' (**R**esource) to the numerical rating when resource bounds have been checked for each test used to achieve the numerical rating.

[2] Note that we assume that the specification is corrrect, *i.e.* this rating scheme does not attempt to say anything about *validation* - matching the specification with any user's requirements.

Many different methods are used to develop test data sets: for example, BS7925-2 lists 12 distinct strategies [Institution, 1998] and adds two more - random testing and a category for all other published testing techniques. Table 1 is primarily based on equivalence class analysis: in order to use it effectively other test generation techniques must be correlated with equivalence class based techniques. In section 4, we relate all the techniques listed in BS7925-2 to the levels in our table. By describing each of those techniques in terms of the equivalence classes effectively 'covered' by them, an assessor can readily determine a level from our table that applies to a component tested by each technique.

2.1 Testing Levels

With the exception of Level 6, for a component to pass at any level, it must pass at all lower levels also. Level 6 (formal derivation from requirements) is excluded from this constraint because it is not a testing strategy *per se* and it is possible to argue that formal derivation makes testing - which is at best a demonstration that a program is likely to be correct - redundant. A pass at lower levels is particularly relevant at Level 2, where it might be assumed that one test suffices for legal inputs: the requirement to pass at Level 1 will ensure a set of critical tests are included. It also clears up some difficulties between Levels 3 and 4 and makes the rules for assigning ratings clear.

Level 1 is the lowest level at which a component would be regarded as tested at all. It relies upon an ability to identify 'critical' functions from a specification. At Level 2, a tester must examine each input and identify legal, illegal and null values for that input. A test case with legal inputs will have been included at level 1, so this level adds tests that ensure that illegal values of inputs are trapped in testing. Assertion mechanisms - provided for C, C++ and Eiffel - are the simplest way to achieve this. The capability is easily provided in languages which do not provide an inbuilt mechanism: a simple function taking a boolean parameter along with some mechanism for compiling it out suffices. More elaborate versions providing logging, stack tracing and other useful output are available, for example, Lee and Waters' Assert class for Java [Lee and Waters, 1999]. Without an assertion mechanism, a component cannot reach Level 2: this will ensure that component designers embed into code self-checking mechanisms to ensure that faults in the component's environment cannot generate illegal inputs which would produce undefined behaviour if not trapped. This level also implies that a specification exists which follows the 'contract' notion of separation of responsibility of users and developers [Meyer, 1997] and clearly defines the responsibilities of users with respect to legal inputs. It is likely that testers trying to reach Level 2 will identify many omissions in the specification - unspecified behaviour for 'unlikely' or 'unreasonable' inputs. All such omissions will need to be rectified before a component can pass at Level 2.

It is possible to write a specification in which behaviour is defined for all possible input values. For example, in Java, one could require that an `IllegalArgumentException` is thrown for all unrealistic values of an input. If a rigorous examination of the specification shows this to be the case, then no further testing is needed to achieve Level 2, but all such exceptions would need to be raised at Level 2.5. Note that we distinguish here between assertions which test obligations of a component user to supply correct arguments and exceptions required by the specification. Assertions can be removed from pro-

Level	Description
0	Totally untested
0.1	Passes test set which exercises randomly chosen methods.
1	Passes test set which exercises critical capabilities. Critical capabilities should be identified in the specification as vital to the operation of a system in which this component would be embedded.
2	Passes test set based on analysis of ranges of input parameters and enumeration of possible exceptions. This set will include 1. at least one value chosen from the legal range of every input parameter, 2. at least one value chosen from the illegal range of every input parameter *and*
2.2	Nulls: For each input for which a null value can be identified, a test case includes that null value.
2.5	Exceptions: Each exception in the specification is raised by at least one test case.
2.7	Memory Leaks: A testing tool has been used to demonstrate the absence of memory leaks.
3	A formal equivalence class analysis has been conducted for every input and output parameter. For inputs, equivalence classes include classes of illegal values defined in the specification and for which method behaviour is undefined. For outputs, equivalence classes include exceptions and other error outputs or states which are not necessarily reflected in values of output parameters. Test cases include 1. at least one representative of each equivalence class (both input and output, legal and illegal values)
	Below Level 4, testing is 'black box', i.e. based on a precise specification only. Level 4 and above implies that implementation details are checked also.
4	Test cases have been chosen using the criteria at Level 3 and an automated technique has been used to ensure coverage of every program statement.

4.5	Coverage includes 'missing branches'. A statement of the form:
	`if ( cond ) statement;`
	has a 'missing branch': it is 'executed' when `cond` is false. Test cases should make `cond` both true and false. Similarly, a switch statement with an empty default part has a missing branch, *e.g.*
	`switch( a ) {`
	`case x: statement1;`
	`case y: statement2;`
	`}`
	Inputs should be chosen such that `a` evaluates to `x`, `y` and a value which is not `x` or `y`.

5	Coverage has been achieved at Level 4 and every definition-use path in the code has been executed at least once in the course of testing.
5.5	Every identifiable path through the code has been executed. See note. (This level considered impractical for all but the simplest pieces of code.)
6	Code derived by provably correct formal transformation from specification

Table 1: Reliability Ranking and Testing Levels

duction code if it has been shown that illegal values are never generated by the system in which a component is embedded whereas statements which generate exceptions mentioned in the specification must remain in production code.

Level 2.2 requires that null inputs be identified and behaviour verified. We include this level explicitly because specifications may often fail to prescribe behaviour and because they generally represent boundary values for which code is often omitted. Nulls will need to be identified for each input type - most are obvious: 0 for integers, 0.0 for floating point numbers, null references for objects, *etc.* Some types have more than one 'null' value, *e.g.* strings will have null pointers and 0 length strings. Some types - particularly enumerated types - will not have nulls and will not generate tests for this level. A test set in which each input is independently set to each of the possible nulls should be included. Thus with n inputs, Level 2.2 will typically generated n additional tests.

At Level 2.5 we have required checking all errors that are specified. This vital area - commonly left until the last minute - is a common cause of insomnia for software engineers. It generates a relatively small number of additional tests - one for each exception explicitly mentioned in the specification, but it is vital for robust code, so error checking is included ahead of full equivalence class analysis.

2.2 Testing Complexity

It is worth noting that testing at Level 2 requires a number of tests which is linear in the number of inputs and outputs. Thus 100% compliance at this level is readily achieved through enumeration of inputs and outputs (including exceptions). Thus we have not yet encountered the problem of super-linear explosions in the number of tests required and this level can be achieved in practice for any component.

If a full equivalence class analysis has been performed on the specification and a test set derived from that analysis, then we rate a component at Level 3. This level is independent of any particular implementation and no artefacts of any implementation are explicitly tested - unless implementation constraints are part of the specification.

Full code coverage testing allows a component to be rated at Level 4. Code coverage is relatively easy to achieve and there are a variety of tools which will provide coverage metrics of varying accuracy [Horgan et al., 1994, Lyu et al., 1994, Ltd, 2000]: it was a straightforward exercise to extract coverage measurements from our symbolic execution system [Bundell et al., 2000b].

Although easy to implement and measure, code coverage is known to have significant weaknesses if it is the sole test strategy. It is easy to build exam-

ples of code fragments and associated data sets which provide coverage, yet manifestly do not fully verify a system. It has also been reported that 100% coverage is difficult to achieve in large, 'mature'[3] systems due to the amount of dead or unreachable code.

Level 4.5 extends a simple coverage criterion - and corrects one of its weaknesses - by observing that some conditions might simply cause statements to be skipped - without providing alternative paths, *e.g.* an `if` without a matching `else`. At Level 4.5, we require coverage to include these missing branches also. Verifying this level requires a slightly more sophisticated coverage tool - one that is able to detect the missing branches and add suitable monitors to ensure that the missing branches are taken for some input set.

Level 5 uses Rapps and Weyuker's notion of definition-use paths [Rapps and Weyuker, 1985] to identify cases which may have escaped coverage analysis. In cases where it is practical to identify and verify every possible path through the code for a method, we allow a rating of 5.5 to be claimed. We note that achieving this level is generally impracticable for all but the simplest methods due to time or cost constraints arising from the potentially infinite numbers of paths which require test.

2.2.1 Loops

A method containing an iteration may have an infinite set of paths which can be identified. In practice, a set of representatives from the natural numbers would be chosen. Binder [Binder, 2000] suggests a possible set of criteria :

- minimum - 1
- minimum (*e.g.* 0)
- minimum + 1
- typical
- maximum - 1
- maximum
- maximum + 1
- excluded interior point
- excluded boundary values

This list assumes that there is a well defined maximum and doesn't allow for loops of the form:

```
for( i=0; i<n; i++ ) ...
```

where `n` is an input parameter. In this case, implementation considerations need to be considered also. For example, a loop that constructs many objects should be tested with iteration counts that

- allocate a non-trivial amount of memory that the system running the tests is always able to provide,
- attempt to allocate an amount of memory that the system cannot provide - to ensure that the problem is handled in an accord with the component specifications[4]
- attempt to allocate an amount of memory that tests the system's ability to retain enough memory for error reporting in accord with the requirements.

The last two cases are sensitive to available system resources at the time tests are run (*see also* 2.4).

Excluded points should also be covered in equivalence class analysis at Level 3, so should not need further elaboration as iteration controls.

2.3 Formally derived code

When a formal mathematical proof can be provided that the code is derived from the specification, we rank a method at Level 6.

2.4 Resource bounds

At each level of testing there is also a requirement to consider available resource bounds that may be violated. One example of this is 'memory leakage', *i.e.* the failure of a component to free memory so that successive invocations of it ultimately lead to insufficient memory for normal operation. Temporal execution requirements provide another example: a component may require excessive CPU resources and so interfere with the operation of the operating system and other components.

Resource requirements or bounds may be defined in the component specification so these can be verified directly. However they need to be tested progressively at higher reliability levels since additional paths through the component may be executed and these may potentially have different resource requirements. Where resource requirements are not explicitly specified they still need to be checked in terms of unwanted or unexpected side-effects the component may induce on available resources. For the two examples already cited, the minimum requirement would be a reasonable bound on both memory usage and the return of memory to the operating system on completion and that a reasonable bound on CPU utilisation was met for a reasonable execution time window for the component.

What is 'reasonable' in this context is very component (and usually platform and architecture) dependent, but there is a clear requirement on the testing process at each level to verify bounded resource usage and release of unneeded resources at completion to achieve the 'R' tag. It is very likely that this assessement must be made for specfic platforms and these would be stated as part of the test results: a prospective user of the component would then need to assess how far the testing environment diverged from the intended platform. However a user would know that a component was capable of performing with some 'reasonable' resource bounds.

If the component was destined to be part of system with tight-memory constraints or a real-time system then resource requirements could very directly impact the performance of the component and would be clearly defined as part of the specification. In this case, the significance of an 'R' tag would be explicitly defined.

2.5 Full component ratings

In Table 1, ratings are specified for a single method of a component. However, components will generally present interfaces with more than one method. To determine a rating for a component, each method

[3] *sic*: we are amazed that a system containing dead code can be labelled 'mature'!

[4] It is conceivable, but not likely, that an untidy crash is allowed when the input parameters are unrealistic! A *complete* specification will prescribe a behaviour in this situation.

is tested and the minimum rating achieved for any one of its methods is the component's rating. However, this method of ranking may not be very relevent in practice and the frequency of usage of each method could be more appropriate. Complex sequences consisting of several method calls are included in our treatment by noting that, in an operation [Bundell et al., 2000b] or sequence of method calls which must be tested - because the specification clearly distinguishes its behaviour from that of other operations - the state of the objects which are inputs to the final method in this sequence is an equivalence class for the final method. That is, the operation is necessary to produce an input equivalence class of the final method. Of course, that equivalence class could be tested independently by artificially constructing the input objects. We counter this by noting that the outputs of the penultimate method in the operation belong to one of its output equivalence classes and must be generated in tests on the penultimate method. Applying this argument backwards shows that all methods in an operation must be tested with the inputs - or producing the outputs - appropriate to the operation. Thus we may focus on individual methods - noting that all operations required by the specification must necessarily be tested in order for Level 3 to be achieved.

2.6 100% compliance

A possible criticism of our levels is that we assume 100% compliance at every stage and make no allowance for partial compliance. This is partly to address a weakness noted by Pfleeger *et al.*[Pfleeger et al., 1994] when reviewing software engineering standards: "The first and most startling result of our work is that many standards are not really standards at all. Many 'standards' are reference or subjective requirements, suggesting that they are really guidelines (since degree of compliance cannot be evaluated)." Secondly, the ratings are designed for components - not systems. The relatively small size of an individual component makes it feasible to attain full compliance at a target level.

A tester failing to reach 100% compliance at Level X has almost certainly reached compliance at Level $X - \delta$ and thus the component is rated as obtaining the $X - \delta$ Level.

3 BS7925-2

BS7925-2 'Standard for Software Component Testing' was accepted as a British Standard in 1998 [Institution, 1998]. It is accompanied by the much smaller BS7925-1 which is simply a glossary of terms used in software testing.

3.1 Testing Strategies

BS7925-2 allows a tester to choose from a number of defined testing strategies. It provides a definition of each strategy and an example of the tests that it might generate.

The following techniques are included in the standard

1. Equivalence Partitioning
2. Boundary Value Analysis
3. State Transition Testing
4. Cause-Effect Graphing
5. Syntax Testing
6. Statement Testing
7. Branch/Decision Testing
8. Data Flow Testing
9. Branch Condition Testing
10. Branch Condition Combination Testing
11. Modified Condition Decision Testing
12. LCSAJ Testing
13. Random Testing
14. *Other Testing Techniques*

It remains primarily a process standard as it does not prescribe any levels of compliance for software components even though simple metrics for reporting coverage using the various techniques are specified. It also allows a tester considerable choice in the technique to be used - merely requiring a 'rationale for their choice' to be made.

The scope of techniques which comply with the standard is made extremely broad by the inclusion of category 14 'Other Testing Techniques'.

4 Correlating Test Strategies

If the proposed testing levels is to be useful for comparing components developed using different methodologies and thus different test strategies, it must be possible to correlate these levels with any efficient testing strategy[5]. Since BS7952-2 contains a comprehensive list of commonly used techniques, in this section we show how all the techniques listed there are related to our testing levels.

4.1 Method execution model

A program enters a method with a defined state: this state consists of:

- values for each input parameter, which includes values of all attributes for any objects for which references are passed to the method, and
- state for the environment.

Ideally, the state of the environment need not be considered, but it does include all variables accessible from the method and, importantly, the current content of all output streams of any type. Thus a method which outputs data or actions - in any form - adds these to output streams or action queues and thus alters the environment state.

On entry, a method will create some additional internal state - values for local variables and any local data streams.

We have considered here the execution of a single method on a class. However, the argument is easily generalized to include sequences of method calls, $m_1, m_2, ..., m_n$. The input state for such a sequence is the input state for the first method call in the sequence, m_1. The output state is the result of the execution of m_n. State resulting from returns from methods m_1 to m_n is simply considered as internal state of a 'composite' method, $M = m_1, m_2, ..., m_n$.

[5] We note that *efficiency* here might have many interpretations: availability of a testing tool might make one testing strategy efficient, as might familiarity and practice in one technique. Thus the reliability index should not require choice of any particular technique.

4.2 Equivalence Classes

An equivalence class is a set of values for inputs and outputs which are equivalent in the specification and processed in the same way by some method, f.

Note that this informal definition is potentially misleading as it implicitly refers to paths that may be taken through the code of f: these paths may derive from implementation considerations and have no mention in the specification. For example, the specification for the `find` operation on a look-up table will generally only require a `true` value if the argument object or key is contained in the table. It will say nothing about how the find method searches the table. Black-box equivalence class analysis will thus have only two output values of interest, `true` and `false`. However, both of these values may be derived by quite different paths through the code, *e.g.* a successful `find` returning true - in a hash table implementation - may have 'hit' in the primary area or entered an overflow area or used a re-hash function. Therefore, we interpret 'in the same way' to mean that *equivalent outputs*, *i.e.* outputs which are not distinguished by the specification are produced.

A complete black-box analysis on every input and output parameter and the environment generates a test set - containing one representative for each class of inputs and corresponding outputs which are indistinguishable in the specification. Successful execution of this test set rates a method at Level 3 in our hierarchy. White-box analysis will in general add more branches to the execution path. If the method contains loops which do not have fixed bounds, then the number of final states - and thus the number of equivalence classes - may become infinite. To reach Level 5.5 - all possible paths have been executed - we allow a tester to apply a set of heuristics to choose representatives of the set of all possible iteration counts.

4.3 Boundary Value Analysis

Boundary Value Analysis is a special form of equivalence class analysis and would usually be performed as an adjunct to equivalence class analysis. It is well-known that many errors arise from incorrect checking of boundary conditions (*e.g.* writing $<$ for $\leq$), thus additional test cases are added in which boundary values of equivalence classes appear - in addition to the cases where an arbitrary representative from the centre of a class is used. Thus, based on the specification, it leads to Level 3. With code available for analysis, boundary value analysis will, if both sides of each boundary are checked, lead to Level 4.5.

4.4 State Transition Testing

A state transition model of a component describes it in terms of its states and the transitions that occur between those states. State based testing strategies are hampered by lack of a definition of a state on which testing can be based: Binder [Binder, 2000] comments: "Most OOA/D definitions of state are untestable: they cannot support an executable determination of exactly what state an object is in.". However, Meyer's definition of object state matches our model and is unambiguous: "The state of an object is defined by all its fields" [Meyer, 1997]. In our model, the input state must include the environment, so that, when considering a method which models part of a state-transition diagram, the 'state' of the S-T diagram includes the state of all objects referenced by the method and the environment. A formal definition of state (*e.g.* Meyer's one) would put objects which have different values of even a single attribute into different states. This generally leads to an unmanageably large number of states. If we examine the usual diagram of a finite state machine in Figure 1, we can infer that the difference between a designer working with a State-Transition model and an OO programmer is primarily the order in which aspects of a system are considered. Whereas a programmer will focus on both the system inputs and the current state (without necessarily distinguishing them) in order to separate operations of the system into equivalence classes, the ST-designer will first focus on the state - attempting to classify states of the system which are conveniently viewed as equivalent first. Then the inputs (messages or events) might be similarly grouped into equivalence classes - using the criterion that all inputs that cause any representative of a state to move to the same resultant state belong in a single equivalence class.

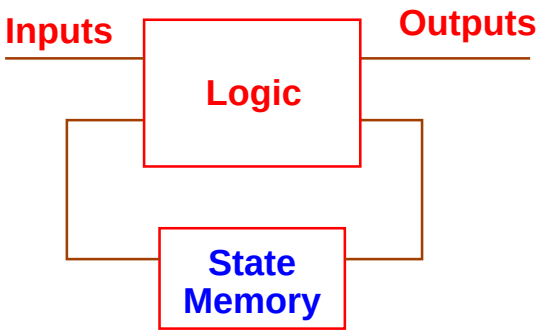

Figure 1: Finite State Machine

In the state model, when a message is sent to an object, a method on that object is invoked: the contents of the message become the inputs for the method. Before the message was received, the system will have been in a named state and the message will cause it to either remain in the same state or move to a new one. An event is similar: it invokes a special type of method - often called a handler. Attributes of the event become inputs to the handler method and the system may or may not move to another state.

Thus although state transition models provide a convenient way of viewing certain types of system, from a testing point of view, they are no different to a system whose behaviour has been analyzed into equivalence classes. Therefore our classification scheme applies equally well: at Level 1, we identify critical transitions (and messages or events that do not cause transitions) and verify them. At Level 2, we ensure that the component raises assertions[6] when events and messages which should not be generated by other components to which the component under test is linked occur. Messages and events with null content - empty strings, 0, *etc.*- are checked at Level 2.2. Error conditions are checked at Level 2.5. Note that many systems for which state transition models are appropriate would treat many 'errors' by the generation of events, *e.g. 'Ring bell on overflow'* - where 'Ring bell' is an event that is indistinguishable from any other. In this case, testing would rate Level 2.5 if behaviour on 'error' conditions is specified and the actions (outputs) are not different from any other action a system might take (or produce). This argument could be applied to almost any software system: to attain a Level 2.5 classification, designers will be re-

[6] Any similar mechanism which flags illegal inputs may be used in place of assertions. For example, an embedded system might use a special device - light, alarm, *etc.*- to flag illegal inputs.

quired to produce an argument that *all* conditions are handled. Analysis into states which aggregate individual states with differing values of individual attributes into a manageable set of states is one part of the equivalence class analysis required for Level 3. The second part determines the contents of messages or events that send systems in the same aggregate state into the same destination aggregate state. The cartesian product of these two sets of values is the set of equivalence classes.

Users of state-transition diagrams are often interested in sequences of valid transitions, but this presents no difficulty as any sequence of transitions may be mapped into a composite method (*cf.* section 4.1).

4.5 Cause-Effect Graphing

A cause-effect graph is a graph linking values of inputs (causes) and values of outputs. In this case, the 'output' is whether or not an action is taken - simply modeled by values in {true,false}. When there are several possible actions, the output takes values in some enumerated set. Arcs link inputs to possible outputs. Each distinct path from a set of input values to an output represents an equivalence class. Since these equivalence classes are easily identified, then tests derived from cause-effect graphs are easily mapped to our levels.

Illegal inputs and exceptions may not appear explicitly in a cause-effect diagram, which would usually be designed to check the (positive[7]) logic of a method. These should either be added to the diagram or treated specially with test sets designed to meet the requirements of Levels 2-2.5.

4.6 Syntax Testing

When the input consists of text which must be parsed by a method before it can process it, syntax testing will be appropriate. It is assumed that the syntax rules to which the input must conform have been derived at some point in the design stage and expressed in a formal notation such as BNF. Paths through the productions of this notation can be determined and the complete set of distinct paths (as with loops, applying some practical constraints when iterative or recursive productions are encountered) forms a set of equivalence classes. Behaviour with nulls would usually be explicitly defined in the formal notation, thus Level 2.2 would usually be achieved without special tests. There will generally be a single 'exception' - failure of the input to obey the production rules of the grammar. However the specification may suggest that illegal characters or other input errors be treated as exceptions. Black box testing to Level 3 would require an exemplar of every alternative of every production visible in the syntax specification. White box testing with access to the code will add more tests related to the techniques used to break the input into tokens, the handling of the input stream (particularly detecting errors in it) *etc.*

4.7 Statement Testing

According to BS7925-2, in this strategy, statements are identified as 'executable or non-executable'. This definition would appear to be undesirably loose: presumably, it was intended to distinguish lines of code which are not explicitly executed (*e.g.* comments, white space, variable definitions, procedure entries and exits, *etc.*) from executable statements. However, it allows a tester to mark dead code as 'non-executable', retain it and claim coverage! This may have been driven by some rather disturbing reports of the amount of dead-code that exists in some supposedly mature software. We would prefer a tighter definition - one that ensured that every instruction of the machine code emitted by a compiler was executed. This definition is designed to ensure that data initialisation and procedure entry and exit code - which may contain branches in some machine models - is not omitted from testing. We also consider that a component containing dead code - for whatever reason - should not reach Level 4: it is a simple matter, in a formal testing environment, to eliminate the dead code (and conditions guarding it) and re-run a full set of equivalence classes. The eliminated code may thus be readily determined to be dead or not.

Statement coverage testing is clearly a necessary condition for complete testing, but it is easily shown to be insufficient. Its major advantage is that it is relatively easy to measure automatically: a variety of tools have been built - for example, ATAC [Horgan et al., 1994] and C-Cover [Ltd, 2000] - because it is a reasonably simple task: we easily extended our symbolic execution system to measure statement coverage [Lee, 2000].

Ensuring that all statements are executed allows a component to be rated at Level 4. However, note that statement coverage testing may omit some tests required for Levels 2 and 3, *e.g.* code for processing nulls and exceptions is often simply omitted, resulting in 100% statement coverage for a component that has serious deficiencies. Thus our rating scheme requires passing at all lower levels also with the effect that Level 4 is often considerably more stringent than simple statement coverage.

4.8 Data Flow Testing

Data flow based testing examines the flow of values through a program. In imperative programming languages, this translates to considering each definition of a variable (definition creates a new variable) and subsequent use. This corresponds to the strongest level of practical testing in our table. Rapps and Weyuker [Rapps and Weyuker, 1985] distinguish between predicate (P-use) and computation (C-use) use, but as an automated analyzer [Lee, 2000] can simply detect and check both, we have not separated the two. Thus our Level 5 corresponds to all definition-use paths [Rapps and Weyuker, 1985] - the strongest of the family of d-u paths.

4.9 Branch/Decision Testing

Branch or Decision Testing is closely related to statement coverage: one criterion for branch coverage is equivalent to statement coverage and leads to Level 4. We have added a level to consider the extra tests required if all outcomes of a decision are checked - even those that do not require explicit processing, *e.g.* `if` without `else`:

```
if ( x < 0 ) x = -x;
```

or switches without `default`:

```
switch( switch_state ) {
  case ON: ...; break;
  case OFF: ...; break;
  }
```

In the latter example, testing the default may be omitted if (switch_state == ON) || (switch_state ==

[7] In this context, positive would imply 'producing definite actions or results'.

OFF) is a pre-condition for this method and, as a consequence, the presence of the appropriate `assert` will be checked at Level 2. We would expect explicit documentation justifying the omission of the `default` test to allow a Level 4.2 rating.

4.10 Branch Condition Testing

The strategies described in this and the following two subsections (4.10 to 4.12 are variations of statement coverage techniques and derived from branch testing, *cf.* section 4.9 preceding, which determine tests that govern control paths through branches. In branch condition testing, only the outcome of decisions is considered: the result is equivalent to coverage testing - this technique simply provides a strategy for achieving coverage.

4.11 Branch Condition Combination Testing

Condition combination testing extends condition testing by examining the individual variables that are contained within the branch expression that determines a branch. In addition to testing all branches, it requires that all combinations of values of individual boolean operands within the branch condition be included. Thus it may be said to 'cover' the truth table and thus will be an efficient test generator when the specification is expressed as a truth table.

This criterion is stronger than our Level 4.2 if the branch condition has not been reduced to its minimum form as it will require tests for some combinations of values required neither by equivalence class analysis of inputs and outputs nor path coverage criteria. Thus we have not allocated a higher level for it as the additional tests it prescribes will generally not be justified. This position is supported by the proponents of the next strategy.

4.12 Modified Condition Decision Testing

Noting that all combinations of booleans may lead to test cases which are not reflected in either formal specifications (and thus contribute to equivalence classes derived from input-output combinations) nor internal implementation considerations (and thus contribute to meeting path coverage criteria), modified condition decision testing is a more practical alternative to all combinations. In this technique, boolean operands within a branch condition are checked for their ability to independently affect the branch decision. Only operands which can affect the decision by being independently set to true or false are included in the generation of tests. While stronger than branch testing, we felt that, combined with a correct equivalence class analysis, it would rarely add additional tests and thus did not warrant an extra level in our table. Thus obtaining 100% coverage with this strategy would generate a rating of 4.2.

4.13 Linear Code Sequence and Jump Testing

The LCSAJ technique again provides criteria to define a set of paths which should be followed in testing. Rather than consider data flow, it considers control flow uninterrupted by a branch followed by a branch. It is more powerful than simple branch coverage [Institution, 1998] and can readily be used to identify infeasible paths which can be eliminated from the test set. Accordingly, we rate a component for which 100% of the code sequences identified by LCSAJ have been tested at Level 5.

4.14 Random Testing

Random testing clearly has benefits in testing large systems - where the number of tests required by any formal analysis is extraordinarily large. However, for the smaller, reusable components for which our rating system is designed, we view random testing with suspicion. To obtain coverage of equivalence classes, many more tests than are strictly necessary must be run and this will only give a measure of confidence. Since random testing will not, in itself, ensure 100% compliance with any of our proposed levels, it is not possible to assign the same type of risk measure that an extension of our model would allow. Since the number of tests which are necessary to achieve Level 2 is linear in the number of inputs to a component, then we cannot envisage a situation where a tester would not test to at least this level in a formal and exhaustive manner, *e.g.* check that *each* exception is raised at least once, so that our table would still provide a useful indication of the reliability of a component. Random testing could be used to qualify a Level 2.5 rating with measures of the coverage of input domains achieved.

4.15 Other Testing Techniques

In the preceding sections, we have demonstrated the relation between the list of defined testing strategies in BS7925-2 and our reliability levels: similar arguments can be applied to other testing techniques.

5 Discussion

Our levels have been used to classify components for reliability in the VeriLib library [Software Component Laboratory, 2000]. There, we assign to a component as a whole the level reached by the lowest rated *functional* method in its interface. We acknowledge that a practical component may often contain methods which have been added for diagnostic or integrity checking and do not expect that they will be rated for the purpose of rating the whole component. Integrity checking methods do present a practical difficulty here as they may have been used to verify correct functioning of other methods. This leads to *consistency checking* - in which two (or possibly more) methods are checked against each other - rather than an absolute verification. For example, in a component which uses a complex data structure whose integrity can only be sensibly checked by a program when the data structure contains more than a trivial number of elements, manual checking of a large complex structure is likely to be more error-prone than writing a relatively simple piece of check code. Checking that an array is sorted or that a binary search tree can be walked in order are examples of very simple code that can mechanically check large structures reliably. In these cases, if the (simple) integrity checking routine confirms that the more complex method-under-test performed correctly, then, since both pieces of code are consistent, for practical purposes, we will accept that both the method-under-test and the integrity check are correct. This, however, remains a potential weakness which could be removed in a number of ways:

1. the same standard of verification is applied to the integrity checking routines,
2. individual methods are rated - so that untested methods in a component's interface are patent
3. enumeration of individual methods' dependencies on other methods is required.

In libraries of components, our scheme meets a need to be able to indicate to prospective purchasers of a component:

1. How likely it is that the component will function without error in actual use.
2. The overall risk of using a system composed of many COTS components.

When two components are available which appear to meet a specific need, the rating scheme allows a designer to assess:

1. The relative risk of using one component compared to another.
2. The possible commercial consequence of using one component.
3. The relative value of two components, *i.e.* whether a component with a higher price will, in fact, justify its premium.

In practical use, we observed two software companies who were very pleased to have the table before them in a negotiating session: it gave them a clear basis for discussing a level of testing that would be achieved: they had a set of targets proposed and defined by an independent third party and negotiations were thus reduced to deciding the level that would be reached. There was no need to devote time to discussing strategies, their definitions, consequences, *etc.*

5.1 Risk Models

When failure has some economic consequence, it is common to see reliabilities of (non-software) components expressed in measures such as Mean-Time-Before-Failure, which enables estimates of the same measure for assemblies to be derived. As more components become available and CBSE becomes more popular, similar risk models will be needed for software systems. Measures such as MTBF are difficult to derive for software components as they are dependent on mode of use (*i.e.* the probability that an input exposing a fault will be encountered in a particular environment): non-software components tend to be designed for a narrower range of environments or have only one critical parameter (*e.g.* a bolt, which may be used in a wide variety of situations, has only a maximum loading or possibly MTBF *vs* load curve) making it easier to use simple failure models. The reliability levels proposed here could be used as the foundation for a risk model by assigning a risk parameter appropriate to the style of use to each level. This need to assign empirical parameters to each level in risk models was a factor in our determination - only partly successful! - to keep the number of levels to a minimum. Thus they provide a first step which could lead to useful risk analysis for systems built by CBSE from COTS components.

6 Conclusion

In order to assess the risk of installing a system constructed from components from many different sources, a well accepted standard reliability index such as the one we propose here will become increasingly necessary. A suitable numerical index will enable quantitative fault models to be derived and used to estimate the risk associated with a complex system. A comprehensive risk model might qualify the value produced by using our index with, for example, the size of a component - using any one of the several size or complexity measures available. Our intention in this paper is to highlight the need for some universally accepted reliability index that enables system integrators to compare different components and assess the risk associated with them. In addition, a system integrator - faced with a component that has a low rating - has a basis on which to make a rational 'make or buy' decision. The likely cost to reproduce a component of higher rating can be balanced with the alternative cost. 'Alternative cost' includes not only the purchase price, but also possible testing effort to confirm that a component meets a higher standard, for example, a desired overall system rating and possible costs to rectify or enhance a failing component.

By proposing a rating which requires 100% compliance at its various levels, we believe that

1. compliance with a rating is more easily verified,
2. the rating is more suited to 'black-box' components, whose owners may not be willing to expose internal details and thus permit assessments based on lines of code, estimates of undiscovered errors, *etc.*to be used.

7 Acknowledgments

This work was supported by a grant from Software Engineering Australia (Western Australia) Ltd through the Software Engineering Quality Centres Program of the Department of Communications, Information Technology and the Arts.

References

[ANSI/IEEE, 1987] ANSI/IEEE (1987). *ANSI/IEEE standard 1008-97: IEEE Standard for software unit testing.* IEEE.

[Binder, 2000] Binder, R. V. (2000). *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley.

[Bundell et al., 2000a] Bundell, G. A., Lee, G., Morris, J., Hope, S., Parr, S., and Dromey, R. G. (2000a). *Component Software: A White Paper: Part II. Technical Aspects.* Software Engineering Australia (WA): `http://ciips.ee.uwa.edu.au/Research/SCL/white6.pdf`.

[Bundell et al., 2000b] Bundell, G. A., Lee, G., Morris, J., and Lam, P. (2000b). A software component verification tool. In *Proceedings: International Conference on Software Methods and Tools.* IEEE Computer Society Press / ACM Press.

[Defence, 1997] Defence (1997). *Defence Standard 00-55: The Procurement of Safety Critical Software in Defence Equipment.* HM Government, Ministry of Defence Directorate of Standardisation.

[Fenton and Neil, 1998] Fenton, N. E. and Neil, M. (1998). A strategy for improving safety related software engineering standards. *IEEE Transactions on Software Engineering*, 24(11):1002–1013.

[Horgan et al., 1994] Horgan, J. R., London, S., and Lyu, M. R. (1994). Achieving software quality with testing coverage measures. *IEEE Computer*, 27(9):60–69.

[Institution, 1998] Institution, B. S. (1998). *Standard for Software Component Testing.* BS7925-2.

[Lee, 2000] Lee, G. (2000). *Symbolic Executor for the CTB: work in progress.* Software Component Laboratory, CIIPS, University of Western Australia.

[Lee and Waters, 1999] Lee, G. and Waters, B. (1999). *Assert for Java.* `http://www.verilib.sea.net.au/categories/Debugging.html`.

[Ltd, 2000] Ltd, C. (2000). *C-Cover.* Codework Ltd. C/C++ coverage tool.

[Lyu et al., 1994] Lyu, M. R., Horgan, J. R., and London, S. (1994). A coverage analysis tool for the effectiveness of software testing. *IEEE Transactions On Reliability*, 43(4):527–535.

[Meyer, 1997] Meyer, B. (1997). *Object-oriented Software Construction.* Prentice Hall, New York, NY, second edition.

[Pfleeger et al., 1994] Pfleeger, S. L., Fenton, N., and Page, S. (1994). Evaluating software engineering standards. *Computer*, 27(9):71–79.

[Rapps and Weyuker, 1985] Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375.

[RTCA, 1992] RTCA (1992). *Software Considerations in Airborne Systems and Equipment Certification: DO-178B.* Radio Technical Commission for Aeronautics,(RTCA, Inc).

[Software Component Laboratory, 2000] Software Component Laboratory (2000). *VeriLib: A Source of Reliable Components.* `http://www.verilib.sea.net.au`.

[Weerahandi and Hausman, 1994] Weerahandi, S. and Hausman, R. E. (1994). Software quality measurement based on fault-detection data. *IEEE Transactions on Software Engineering*, 20(9):665–676.