



The BCS software component testing proto-standard

S.C. Reid

*Royal Military College of Science, Swindon,
Wiltshire, SN6 8LA, UK*

ABSTRACT

Software testing should provide both a means of assessing the quality, and a degree of confidence, in a software product. However, the actual quality of testing carried out by different organisations, or expected between customers and suppliers, varies alarmingly. Standards currently available fail to define any test design techniques and provide few, if any, means of measuring and comparing the quality of testing performed on software. Without these attributes the user cannot improve the quality of testing undertaken and, hence, the quality of the software product. This paper describes the Software Component Testing proto-Standard, which has been developed to fill this gap by defining a generic test process, test case design techniques, and a set of objective measures for carrying out dynamic testing of software components, as well as providing guidelines on their use.

INTRODUCTION

The Software Component Testing Standard was conceived in early 1989 with the aim of being the first of a number of software testing standards developed by, and for use by, members of the (now BCS) Specialist Interest Group in Software Testing. It is currently envisaged that it will be presented to the British Standards Institution (BSI) in mid 1994, having been issued as a draft on three occasions and undergone external review twice.

This paper initially introduces software component testing and the attributes required of standards in software engineering; the paper is then divided into two main sections. The first section details the experience of developing the standard and requires no extra knowledge of software testing. The second section provides a discussion of some of the technical issues encountered in standardising software component testing and defining best practice in this area.

262 Software Quality Management

WHAT IS SOFTWARE COMPONENT TESTING?

A component is defined for this standard as a minimal program for which a separate specification is available. Component testing is often referred to as unit or module testing, although, sometimes these terms can refer to the testing of larger programs, which are made up of a number of components; this is not the case for component testing. It is solely concerned with that testing corresponding to the dynamic testing at the bottom of the 'V' life cycle model. A more detailed discussion of software component testing is provided in the Technical Issues section.

SOFTWARE ENGINEERING STANDARDS

Cargill [1] provides a pragmatic definition of a standard:

A standard, of any form or type, represents a statement by its authors, who believe that their work will be understood, accepted, and implemented by the market. This belief is tempered by the understanding that the market will act in its own best interests, even if these do not coincide with the standard. A standard is also one of the agents used by the standardization process to bring about market change.

The following are believed to be the attributes of a 'good' software engineering standard; each attribute is discussed below both in terms of software engineering in general and then more specifically in terms of the component testing proto-standard:-

- present consensus view
- objective
- based on process, product and/or resources
- maturity of standardised area
- restricted to a cohesive set of requirements
- provide a technology transfer function

Present Consensus View

To gain acceptance standards must satisfy the common interest rather than the self-interest of the developers. Users must feel they gain from their use in comparison to either using another standard or no standard at all. The ideal way to produce a consensus standard is for representatives of all those who intend to use it to take part in its development.

The proto-standard working group is made up of a diverse set of members representing large and small businesses from a variety of industry sectors, consultants, government agencies and academia. To reinforce this approach the proto-standard is also undergoing an extensive internal and external review process.

Objective

It should be possible to assess conformance to standards otherwise there will be the temptation to either ignore them or conform to them by word, rather than in spirit. Each of the requirements should thus be testable. However, Fenton [2] reports that research carried out for the SMARTIE project on software standards has shown that for only 1.5% of requirements in the software standards investigated was it possible to make an objective assessment of conformance. If it is difficult to test conformance to a standard then purchasers are not able to adequately assess and compare products or services.

The proto-standard contains a defined test process, test case design techniques and test coverage measures. For the test process and procedures auditability will be available through the mandated documentation. The test coverage measures will give quantifiable results for the testing.

Based on Process, Product and/or Resources

Classical engineering standards concentrate on specifying measurable attributes of the products developed and are thus generally product-based. The lack of metrics in software engineering has limited development of product-based software standards leading to standards that are often process-based, lacking objective conformance criteria. This can make it difficult to both apply software standards and to assess conformance. Resource-based standards specify staffing and tool requirements; although found in software, they tend to be fairly general requiring, typically, 'suitably qualified staff', which is obviously open to individual interpretation and therefore not objective. Standards may comprise any combination of the three paradigms.

The proto-standard is both process and product based, but is *not* resource based. A generic test process is specified along with procedures for test case design to define the *process*. Test coverage measures are defined for measurement of the *product* of the testing. It was felt that specifying staff and tool requirements may dissuade some people from using the proto-standard who did not, or could not, meet them and who might otherwise gain from using it.

Maturity of Standardised Area

Software engineering is a relatively new activity, however, where best practice can be agreed and the technology appears to be stable then the area is ready for standardisation.

Component testing, especially that applied to procedural languages, can boast a long history within the realms of software engineering and the criterion of maturity is easily satisfied: the test process is well understood, and many of the techniques and measures used within it date back to before 1979 and Myers' [3] classic 'Art of Software Testing'; these are now agreed best practice.

264 Software Quality Management

Restricted to a Cohesive Set of Requirements

It has been stated by Fenton [2] that, in general, software standards try to define too much. The need for a cohesive set of requirements is analogous to the use of high cohesion in software design and carries with it the same benefits - greater understandability, localisation of errors, and ease of maintenance.

The dynamic testing of software components was felt to provide just such a set of cohesive requirements that would form a usable standard to fit within a framework of software testing standards.

Provide Technology Transfer Function

BS0, the 'standard for standards' [4], states that "...Standards should be based on the consolidated results of science, technology and experience, and aimed at the promotion of optimum community benefits". Fenton [2] argues that there is "no quantitative evidence to support many of the much-trumpeted methods in software engineering", and that "only inspection techniques had been shown beyond reasonable doubt to be cost-effective". This would suggest that the majority of software-related standards do not conform to BS0 (Nash [5] in 1986 had identified over 1100 software-related standards as either in existence or in development). It can be argued that software engineering is not a mature enough discipline for the 'rigid' rules applied to hardware standards and that software standards should include information which would normally only be included in the guidelines of hardware standards. In this context they can be considered as vehicles for educating the users of the standard.

Despite the obvious maturity, in software terms, of the process and the fact that dynamic component testing is a well understood activity there is no standard for it. A number of standards exist which *include* requirements and guidelines for the dynamic testing of software components. However, the premise for producing a new standard was that none of these standards provided definitions of test case design techniques and measures of test coverage. Nor were they specifically concerned with software component testing; they covered the area as part of a wider activity. The appendix contains brief summaries of the coverage of component testing by a number of these standards to demonstrate the level and 'quality' of this coverage.

THE HISTORY AND DEVELOPMENT OF THE STANDARD

Conception

The development of the Standard was first proposed by Martyn Ould of Praxis and was taken up by the Specialist Interest Group in Software Testing in early 1989. It was recognised that no standards were available for software component testing and the stated initial aim was to produce a standard that defined how 'well' a software component was tested by dynamic testing. The intention was to start with component testing (as this task was considered to be

of a manageable size) and then to subsequently produce further standards to gradually populate a framework of software testing standards.

Work on the standard continued for three years with the working group convening at irregular intervals, normally immediately following Specialist Interest Group meetings. Development during this period can be distinguished by the long periods between meetings (six months was not unknown) when individuals, or small subgroups, drafted or reworked large sections of the proto-standard alone. This inevitably led to difficulties in reaching a consensus on acceptance of the current working draft. Despite this the authors and editors for this period produced the vast majority of the content expected to be included in the issue planned to go to BSI in mid-1994. Three drafts were issued in this initial period:

Issue 0.3 Jan. 1990 The initial issue, by Martyn Ould, for discussion within the Specialist Interest Group, formatted to BS0, included test case design techniques and 'grades of testedness', but no test process.

Issue 1.2 Nov. 1990 The first 'public' version of the standard, now edited by Dorothy Graham, of Grove Consultants. Generally improved, and now including a procedure for using the standard (this was an initial attempt to define a test process). Guideline details were moved to appendices, which were expanded to include proformas for test documentation and comments on use of the proto-standard.

Issue 1.3 July 1992 Reformatted to European standard by co-editors Dorothy Graham and Mark Fewster (of Performance Software) and incorporating numerous detailed technical changes as a result of comments received. The procedures for using the standard were removed and the 'grades of testedness' were moved to the appendices.

In December 1992 it was agreed that a more formal, or systematic, approach was required to produce the standard. Subsequently a working group, with appointed officers and a constitution, has met on a monthly basis and plans to present a standard to BSI in mid-1994.

The Development Life Cycle Model

The development process for the standard can be represented by the model shown in Figure 1. The first four years' work has been depicted as prototyping due to the relatively unstructured approach to development used in that period; the results of this stage have provided valuable insight into the user requirements and generated definitions of procedures and measures that will be reused later. However, in January 1993, it was felt that a defined process, which started with requirements definition, followed by design and implementation, would provide a better basis for success. Thus a process model, similar in form to the software life cycle, was adopted for the project.

266 Software Quality Management

The Standards - Software Development Analogy

Stevens and Scheffer [6], in their paper on developing standards for the European Space Agency, suggest a three phase process, a "similar approach" to software development:

1. *Production of requirements for the software engineering standards;*
2. *Definition of an abstract structure, style guide, and information constraints on that structure;*
3. *Implementation within the structure.*

Robin Bloomfield in his keynote address to the 1993 Software Engineering Standards Symposium made the same observation from his experience in the production of a number of standards including MoD Def Stan 00-55 and 00-56. He argued that not only were the life cycle phases analogous, but that the similarity extended to management of the process as well.

The working group has found that their experience bears out the analogy; the decision to adopt a systematic approach similar to software development has worked so far. It is disappointing, however, that it took four years for us to find a suitable methodology for the development of the standard, and also surprising that it had to be discovered by chance.

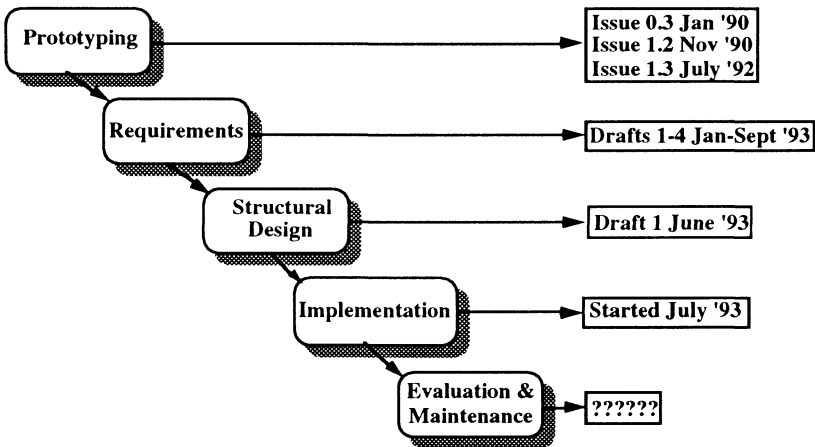


Figure 1: The proto-Standard Development Process

Process improvement

Using a similar analogy, then it could be asked whether the perceived benefits of software process improvement could be applied to standards development. The difficulty, of course, is that for much of standards development, any experience

gained on one development is subsequently lost as those involved take no further part in standards development activities. Perhaps this is because they are only interested in standardising their area of expertise or perhaps one experience of standardisation is enough! In any event the lessons learnt in the process need to be recorded to allow future developers to apply the techniques and methods that appeared to work and avoid those that didn't. To this end the working group plans to produce a separate report from the actual standard describing such details, as suggested by Hawkes et al [7], the developers of DO178B.

In terms of the SEI process maturity model, as described by Humphrey [8], the first four years of development of the proto-standard followed the lowest level of maturity: Initial Process Maturity, where the process can be described as chaotic. For the reasons given above the next level of maturity, Repeatable Process Maturity, where the process is driven by the previous experience of participants, was not available to the developers of the proto-standard. It is believed that the third level, Defined Process Maturity, is currently being used for development of the proto-standard, with recognised life cycle phases and techniques being used, especially with respect to review procedures. Elevation of the standards development process to the next, and highest, two levels of maturity would require the collection of metrics and the use of process improvement techniques. The attainment of these levels of maturity will require positive action by the standards bodies themselves, such as BSI, which is currently lacking.

Next Stages - Evaluation

As each section (or subsection) is completed it will be reviewed by the working group. Any necessary revisions will be carried out until the section is incorporated into the working draft. Once complete the working draft will be released for external review by volunteers representative of the standard's intended audience. Comments will be invited whenever an individual or an organisation feels a change is necessary or they have some other concern. Finally, the working group shall resolve any issues arising from the comments and forward the 'finished' standard to the relevant BSI technical committee, IST15. The intended review process for the proto-standard is shown in figure 2.

A Glossary of Terms used in Software Testing

The ISO/IEC Directives [9] state:

The same term shall be used throughout each standard or series of standards to designate a given concept.

ISO 2382, an IT terminology standard, is available, which is made up of 22 parts for a wide range of areas within the IT field. Unhappily it does not include software testing. No other standard could be found that supplied a suitable range of definitions (that we considered to be correct) and so the proto-standard initially contained a large section defining terms used within it.

268 Software Quality Management

The definitions required by the software component proto-standard necessarily overlap into other areas of software testing. Because the software component testing standard is intended to be the first of a number of software testing standards it was decided that a generic base standard of software testing terminology should be produced to support this framework. The definitions section of the proto-standard was removed to form the basis for this separate glossary standard.

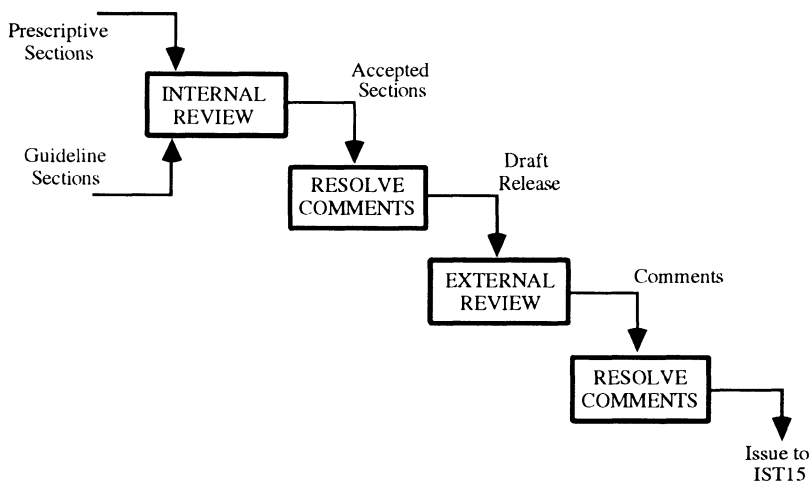


Figure 2: The Review Process

A Software Testing Standards Framework?

As has already been mentioned the software component testing standard was conceived as being the first in a series of software testing standards produced by the BCS Specialist Interest Group in Software Testing. From the experience with the software component testing standard it now appears that this goal may not be attainable in a reasonable timescale. The IEEE has instigated a Master Plan for Software Engineering Standards (MPSE), part of which calls for a new framework of software testing standards. The IEEE are aware of the proto-standard (it is included in the charter for the software testing part of the MPSE) and believe it could form an integral part of their framework.

It has also long been the intention that the proto-standard would align with the ISO 9000 series of quality assurance standards, which require software developers to use software testing standards. It is expected that the software component testing standard would be called from within a quality management system compliant with ISO 9001.

TECHNICAL ISSUES

This section explores the technical issues concerned with software component testing that are addressed as part of the standardisation process.

Still worth doing?

Component testing is carried out on software components prior to integration testing with the purpose of ensuring the correctness of the implementation. It is checking that the component does everything required of it in its specification and that its code works properly.

It is recognised best practice in software development to detect and remove faults from the software at the earliest opportunity due to the increasing costs of correction later in the life cycle, supporting the widespread quality goal of minimising the amount of rework.

The proponents of inspections and reviews argue that static testing is far more cost effective than dynamic testing. At the EuroSTAR '93 conference Tom Gilb presented figures that showed that errors were detected statically at 5% of the cost of detecting them dynamically, and stated that on one IBM project inspections had "reduced the quantity of bugs left for testing by 95%". Bill Hetzel and Dave Gelperin [10] point out, however, that this result is predictable as the static testing is invariably carried out prior to dynamic testing and the most difficult to find faults are those left for dynamic testing.

It is concluded that despite the undoubted efficacy of static techniques there still remains a requirement to carry out dynamic testing, if only to detect the 'last 5%'. Whether this is started at the component level, or left until integration or system testing, is another matter. MacNamara [11], of IBM, has suggested that formal component testing can be ignored, given a team of 'proven professionals', however the success, or not, of this approach will only become apparent when the project is completed.

Myers [3] argues that there are three reasons for component testing:

First, module testing is a way of managing the combinatorics of testing, since attention is focused initially on smaller units of the program. Second, module testing eases the task of debugging (the process of pinpointing and correcting a discovered error), since, when an error is found, it is known to exist in a particular module. Finally, module testing introduces parallelism into the testing process by presenting us with the opportunity to test multiple modules simultaneously.

Research needs to be carried out in this area, but in the absence of successful projects that have neglected component testing, it is felt that the traditional arguments for its use are still overwhelming.

The Scope Of The Component Testing Proto-Standard

Objective The objective of the standard is to enable the measurement and comparison of testing performed on software components. This will enable users of the standard to objectively improve the quality of their software testing, and indirectly the quality of their software products.

Target Audience This is anyone involved with software component testing and includes testers, developers, managers, procurers of software, quality assurance personnel, academics and developers of related standards.

Approach The standard prescribes characteristics of the test process. It describes a number of techniques for test case selection from which the developer/tester should choose as appropriate for the particular type of software being developed (see later for a more detailed discussion on choosing techniques and measures).

Constraints

- The component must have a specification, otherwise there is nothing to test the code against (see later for a more detailed discussion of module specification).
- Static analysis of the component, such as data flow analysis and reviews/inspections, is not covered.
- Testing of non-functional attributes of the component, such as usability, performance and maintainability, are not covered.
- The main procedures described are those that are repeatable, ie that when used by different people, they generate test cases with the same potential to reveal errors.
- Integration, systems and acceptance testing are not covered.

A Generic Standard

The standard has been developed to be as widely acceptable as possible. The intention has been to produce a standard that is applicable to all industry sectors producing software for applications ranging from highly safety critical to leisure. The danger in producing a generic standard is that in making it widely-applicable it becomes too general and conformance provides little confidence in the product. To overcome this the standard provides a wide variety of techniques and measures described in sufficient detail to allow both their application and assessment of conformance. Those techniques and measures chosen by the user will depend on both their suitability for the application and the degree of confidence required by the testing. It will then be possible to assess conformance to the standard *for this chosen set of techniques and measures*. This choice is described in more detail later.

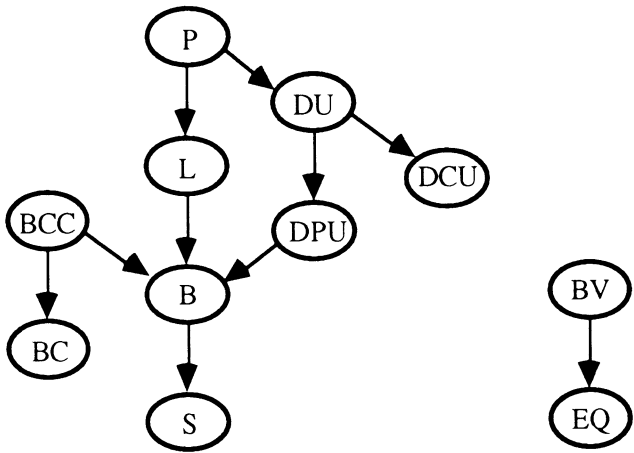
The standard is not resource based to avoid the danger of it being discarded due to either a lack of tools or 'suitable' staff qualifications. It is recognised that



for a number of the testing techniques and measures their use is inappropriate without suitable tool support and where this is the case the standard notes this.

Choosing Appropriate Techniques and Measures

It was suggested that the standard would be most useful if it prescribed the techniques and measures to be used for each type of application. One way of doing this would have been to use levels of criticality, such as those defined in DO178B [12], and prescribe a set of techniques and measures for each such level. This approach was rejected as there is no technique and measure, or set of techniques and measures, that are agreed best practice for each (or any) level.



Key

S	Statement
B	Branch (Decision)
BC	Branch (Decision) Condition
BCC	Branch (Decision) Condition Combination
L	Linear Code Sequence and Jump
P	Path
DPU	Data Definition Predicate-Use
DCU	Data Definition Computation-Use
DU	Data Definition-Use
EQ	Equivalence Partitioning
BV	Boundary Value

Figure 3: Techniques/Coverage Hierarchy

A Hierarchy of Techniques and Measures At present it is only possible to provide a hierarchy of techniques and measures that shows, for *some* of them,

the relative 'confidence' provided by their use. This is shown in figure 3. The hierarchy only shows the *relative* effectiveness of a technique or measure and uses the "subsumes" ordering. This states that a technique or measure *subsumes* (or *includes*) another if, and only if, a test that satisfies the first necessarily satisfies the second. It shows, for instance, that test data that achieves 100% branch coverage will definitely achieve 100% statement coverage. It does not give, however, any quantitative measure of the improved confidence afforded by the technique or measure that subsumes another. The hierarchy shows no implied level of effectiveness by position in the figure other than the subsumes relation shown by the arrows. Only two functional techniques are represented, Boundary Value Analysis and Equivalence Partitioning, as it is not possible to derive subsumes relations between any of the other black box techniques or coverage measures.

Limitations of the Hierarchy Two problems arise from this hierarchy: Firstly, the cost of applying the techniques and measures is not taken into account. Secondly, the subsumes relationship may not take into account the ability of the technique to expose faults in the component. Hamlet [13] has argued, for instance, that the techniques and measures based on data definition-use may be more effective than full path testing (which subsumes them), because they concentrate attention on the important cases that may be trivially treated as just a few of the vast number of cases required to cover all paths.

Black Box, White Box, or Both?

Two types of techniques are available for dynamic software component testing: functional (black box) and structural (white box) testing. With functional testing test cases are generated from the component specification without reference to the generated source code. It is thus possible to generate functional test cases before any coding takes place, although it is also difficult to objectively determine coverage. Structural testing is based on a knowledge of the source code and test cases are generated to satisfy a test coverage criterion.

When comparing functional and structural techniques it is easiest to consider what they miss. Functional techniques rely purely on the specification and so any code produced due to specifics of implementation that are not addressed in the specification will not be tested. Equally any superfluous code will also remain untested using a purely functional approach. On the other hand, structural tests, by using only the code as their basis, cannot identify any errors of omission where, for instance, a specified function has been overlooked by the coder. It is regularly stated that structural testing is merely testing that the code 'does what it say it does'. This is, however, an oversimplification, for the extra tests required by structural testing, as long as they are carried out properly (ie calculating expected results from the specification in advance of execution), provide a valid means of detecting faults.

Thus there are inadequacies inherent to both techniques; the solution is to

use a combination. Myers [3] recommends using functional techniques first and then using structural techniques, as necessary, to supplement the test cases already created to achieve the required coverage criterion. Given that a combination is to be used then which techniques should be chosen, ie which of the techniques available within functional and structural testing are most effective?

Several studies have been carried out on the effectiveness of different test case design techniques, by, for instance, Hamlet [13], Basili [14] and Ntafos [15], but they produce conflicting results. Hetzel and Gelperin [10] dispute the results of much of this research anyway. They believe that these 'experiments' fail to take account of several basic issues, such as the effect of testing against imperfect specifications (ie in the real world faults are also found in the specification, which affects the apparent effectiveness of the component testing), the timing of test case design with respect to coding, and the impact of defects found. Probably the most effective course is to use a 'mix' of techniques - but this 'mix' is dependent on the component and the system it is in. Obviously this subject area is in need of further research; perhaps the increase in metrics initiatives will provide the basis for this work. Until this has been done, however, no agreed best practice can be defined and for this reason the standard, although defining a large number of techniques and measures, does not prescribe when and where they should be used; this is left to the individual users.

The Problem with Module Specifications

As an initial constraint the standard mandates that specifications must be deterministic, ie it must be possible to derive the expected results for a given test case. It is also necessary for the specification to define the possible inputs to the component to generate test cases. These are obvious requirements for the specification, however, its form plays a large part in determining the possible quality of testing.

The notation used in the specification will determine, to some extent, the detail provided by the specification, which limits the extent of functional test case design. For instance, a specification using a data-based notation for a process-oriented component will provide few relevant details for the tester.

The amount of useful detail present in the specification is also greatly influenced by the level of abstraction at which it is written relative to its implementation in code. If the specification is written at a high level of abstraction then it will contain few details to aid in functional test case design, conversely a specification written at too low a level of abstraction is problematic. Consider a component specification composed in structured english, say, where there is a direct mapping between the specification and the high level code. There will now be no superfluous, nor implementation specific code added and no omissions occurring, so there is very little difference between functional and

274 Software Quality Management

structural testing for this component. In fact as the code and specification match so closely then it can be argued that only structural testing is possible in this case.

The level of abstraction of the specification is critical to the effectiveness of functional testing; the problem lies not in the code but in the production of the specification. The quality achieved by use (and conformance to) a component testing standard is therefore dependent on the quality of this earlier product of the software life cycle. This reinforces the argument that the proto-standard should form part of a larger set of standards.

CONCLUSION

The experiences involved with development of the software component testing proto-standard have been described and the similarities between the development methodologies for the proto-standard and for software have been discussed. It is suggested that this analogy could be extended from project management to process improvement and a process maturity model.

The difficulty of choosing which test case design techniques and measures to use from the proto-standard has been addressed and it has been concluded that given current technology it is not yet possible, except in the broadest terms, to define best practice in this area. Therefore it has been left to the user of the proto-standard to choose the techniques and measures most suitable for their situation. The level of confidence afforded to users of the proto-standard is thus determined not only by conformance to the proto-standard, but also by this choice.

However, a number of other problems remain that also need to be investigated further. It can be argued that the test coverage measures are only objective for structural coverage and that functional coverage is subjective, making conformance checking difficult in this area. The quality of specifications is outside the scope of the proto-standard, but its affect on the product of component testing needs to be quantified. The proto-standard also mandates that some independence is achieved in the testing, but it is not known how this affects the quality of testing.

APPENDIX: STANDARDS FOR SOFTWARE TESTING

This section summarises the typical coverage of software component testing by a number of standards to show the current level of standardisation in this area:-

The Procurement of Safety-Critical Software in Defence Equipment, Interim Def Stan 00-55, 1991.

This standard requires that all safety functions of the software are exercised and that the following levels of structural coverage of the code are achieved:

- (a) All statements.
- (b) All branches for both true and false conditions, and case statements for each possibility including 'otherwise'.
- (c) All loops for zero, one and many iterations, covering initialisation, typical running and termination conditions.

As (b) subsumes (a) this requirement seems pointless. The lack of specific requirements for functional testing at this level is presumably due to the standard's emphasis on the use of Formal Methods. The guidance part of this standard suggests the use of random test case generation and if this fails to achieve the required structural coverage then extra test cases are generated from the "design" of the code, as opposed to its specification.

Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B, 1992.

This standard emphasises requirements-based testing. It suggests equivalence partitioning, boundary value analysis and state transition testing amongst other functional testing techniques. It requires different levels of structural coverage for the four levels of criticality it defines ranging from statement coverage to modified condition/decision coverage. Test case generation from the code structure is only suggested if functional testing provides an inadequate test set. Independent testing is required for certain techniques and levels of criticality.

Standard for Software Unit Testing - ANSI/IEEE 1008 - 1987

This standard, which applies to software units of any size, states that "...every software feature must be covered by a test case or an approved exception.". It also suggests more specific features, such as states, transitions and data characteristics, which could be tested. For structural coverage it requires that "...every instruction that can be reached and executed must be covered by a test case or an approved exception", although its guidelines suggest branch coverage for critical units, or where specifications are lacking.

Codes of practice for Testing of Computer-Based Systems, BS 5887, 1980.

This standard states that, typically, the test specification will include the test method to be used and that, usually, test plans will be required to provide both 'negative testing' and 'positive testing'. No test case design techniques are mentioned, although it does state that test independence "may be desirable".



REFERENCES

1. Cargill, C.F. *Information Technology Standardization* Digital Press, 1989.
2. Fenton, N. and Page, S. 'Towards the Evaluation of Software Engineering Standards' in *Proceedings of the Software Engineering Standards Symposium*, Brighton, United Kingdom, 1993. IEEE Computer Society Press, 1993.
3. Myers, G.J. *The Art of Software Testing* Wiley Interscience, 1979.
4. *BS0, A Standard for Standards*, British Standards Institution, 1991.
5. Nash, S.H. and Redwine, S.T. 'Map of the World of Software-Related Standards, Guidelines, and Recommended Practices' *Computer Standards and Interfaces*, Vol.6 No.2, pp. 245-265, 1987.
6. Stevens, R. and Scheffer, A. 'Developing a Structured Set of Standards' in *Proceedings of the Software Engineering Standards Symposium*, Brighton, United Kingdom, 1993. IEEE Computer Society Press, 1993.
7. Hawkes, D.J., Struck, W.F. and Tripp, L.L. 'An International Safety-Critical Software Standard for the 1990s' in *Proceedings of the Software Engineering Standards Symposium*, Brighton, United Kingdom, 1993. IEEE Computer Society Press, 1993.
8. Humphrey, W. 'Characterizing the Software Process: A Maturity Framework' *IEEE Software*, Vol.5, No.2, pp. 73-79, 1988.
9. *ISO/IEC Directives - Pt 3. Drafting and presentation of International Standards* ISO, Geneva, 1989.
10. Hetzel, B. and Gelperin, D. 'Software Testing: Some Troubling Issues' *American Programmer*, pp. 22-27, April 1991.
11. MacNamara, G. and Murphy, C.M. 'People and the Pursuit of Excellence' in *Proceedings of the 1st European International Conference on Software Testing, Analysis & Review*, London, United Kingdom, 1993.
12. *RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Dec 1, 1992.
13. Hamlet, R. 'Theoretical Comparison of Testing Methods' *SIGSOFT Software Engineering Notes*, Vol. 14, Iss.8, pp. 28-37, 1989.
14. Basili, V.R. and Selby, R.W. 'Comparing the Effectiveness of Software Testing Strategies' *IEEE Trans. on S/W Eng.*, Vol.SE-13, No.12, pp. 1278-1296, 1987.
15. Ntafos, S.C. 'A Comparison of Some Structural Testing Strategies' *IEEE Trans. on S/W Eng.*, Vol.13, No.6, pp. 868-874, 1988.