Standard for Software Component Testing

Working Draft 3.4
Date: 27 April 2001

produced by the

British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST)

Copyright Notice

This document may be copied in its entirety or extracts made if the source is acknowledged.

This standard is available on the World Wide Web at the following URL:
http://www.testingstandards.co.uk/

© British Computer Society, SIGIST, 2001

© British Computer Society, SIGIST, 2001

Contents

FOREWORD **1**

INTRODUCTION **2**

1 SCOPE **3**

2 PROCESS **5**

3 TEST CASE DESIGN TECHNIQUES **8**

3.1 EQUIVALENCE PARTITIONING 8
3.2 BOUNDARY VALUE ANALYSIS 8
3.3 STATE TRANSITION TESTING 9
3.4 CAUSE-EFFECT GRAPHING 9
3.5 SYNTAX TESTING 9
3.6 STATEMENT TESTING 10
3.7 BRANCH/DECISION TESTING 10
3.8 DATA FLOW TESTING 11
3.9 BRANCH CONDITION TESTING 11
3.10 BRANCH CONDITION COMBINATION TESTING 11
3.11 MODIFIED CONDITION DECISION TESTING 12
3.12 LCSAJ TESTING 12
3.13 RANDOM TESTING 12
3.14 OTHER TESTING TECHNIQUES 12

4 TEST MEASUREMENT TECHNIQUES **13**

4.1 EQUIVALENCE PARTITION COVERAGE 13
4.2 BOUNDARY VALUE COVERAGE 13
4.3 STATE TRANSITION COVERAGE 13
4.4 CAUSE-EFFECT COVERAGE 13
4.5 SYNTAX COVERAGE 14
4.6 STATEMENT COVERAGE 14
4.7 BRANCH AND DECISION COVERAGE 14
4.8 DATA FLOW COVERAGE 14
4.9 BRANCH CONDITION COVERAGE 15
4.10 BRANCH CONDITION COMBINATION COVERAGE 15
4.11 MODIFIED CONDITION DECISION COVERAGE 15
4.12 LCSAJ COVERAGE 16
4.13 RANDOM TESTING 16
4.14 OTHER TEST MEASUREMENT TECHNIQUES 16

ANNEX A PROCESS GUIDELINES (INFORMATIVE) **17**

ANNEX B GUIDELINES FOR TESTING TECHNIQUES AND TEST MEASUREMENT (INFORMATIVE) **26**

B.1 EQUIVALENCE PARTITIONING 26
B.2 BOUNDARY VALUE ANALYSIS 31
B.3 STATE TRANSITION TESTING 35
B.4 CAUSE EFFECT GRAPHING 39
B.5 SYNTAX TESTING 41
B.6 STATEMENT TESTING AND COVERAGE 44
B.7 BRANCH/DECISION TESTING 45
B.8 DATA FLOW TESTING 47
B.9 / B.10 / B.11 CONDITION TESTING 50
B.12 LCSAJ TESTING 54
B.13 RANDOM TESTING 59
B.14 OTHER TESTING TECHNIQUES 60

ANNEX C TEST TECHNIQUE EFFECTIVENESS (INFORMATIVE) **61**

ANNEX D BIBLIOGRAPHY (INFORMATIVE) **63**

ANNEX E DOCUMENT DETAILS (INFORMATIVE) **64**

© British Computer Society, SIGIST, 2001

Foreword

This working draft of the Standard replaces all previous versions. The previous edition was working draft 3.3, dated 28 April 1997. This version has improved formatting and updated contact details. The technical content remains unchanged from the previous version.

Introduction

The history of the standard

A meeting of the Specialist Interest Group on Software Testing was held in January 1989 (this group was later to affiliate with the British Computer Society). This meeting agreed that existing testing standards are generally good standards within the scope which they cover, but they describe the importance of good test case selection, without being specific about how to choose and develop test cases.

The SIG formed a subgroup to develop a standard which addresses the quality of testing performed. Draft 1.2 was completed by November 1990 and this was made a semi-public release for comment. A few members of the subgroup trialled this draft of the standard within their own organisations. Draft 1.3 was circulated in July 1992 (it contained only the main clauses) to about 20 reviewers outside of the subgroup. Much of the feedback from this review suggested that the approach to the standard needed re-consideration.

A working party was formed in January 1993 with a more formal constitution. This has resulted in Working Draft 3.4.

Aims of the standard

The most important attribute of this Standard is that it must be possible to say whether or not it has been followed in a particular case (i.e. it must be auditable). The Standard therefore also includes the concept of measuring testing which has been done for a component as well as the assessment of whether testing met defined targets.

There are many challenges in software testing, and it would be easy to try and address too many areas, so the standard is deliberately limited in scope to cover only the lowest level of independently testable software. Because the interpretation of and name for the lowest level is imprecise, the term "component" has been chosen rather than other common synonyms such as "unit", "module", or "program" to avoid confusion with these more common terms and remain compatible with them.

© British Computer Society, SIGIST, 2001

1 Scope

1.1 Objective

The objective of this Standard is to enable the measurement and comparison of testing performed on software components. This will enable users of this Standard to directly improve the quality of their software testing, and improve the quality of their software products.

1.2 Intended audience

The target audience for this Standard includes:

- testers and software developers;
- managers of testers and software developers;
- procurers of software products or products containing software;
- quality assurance managers and personnel;
- academic researchers, lecturers, and students;
- developers of related standards.

1.3 Approach

This Standard prescribes characteristics of the test process.

The Standard describes a number of techniques for test case design and measurement, which support the test process.

1.4 What this Standard covers

1.4.1 Specified components. A software component must have a specification in order to be tested according to this Standard. Given any initial state of the component, in a defined environment, for any fully-defined sequence of inputs and any observed outcome, it shall be possible to establish whether or not the component conforms to the specification.

1.4.2 Dynamic execution. This Standard addresses dynamic execution and analysis of the results of execution.

1.4.3 Techniques and measures. This Standard defines test case design techniques and test measurement techniques. The techniques are defined to help users of this Standard design test cases and to quantify the testing performed. The definition of test case design techniques and measures provides for common understanding in both the specification and comparison of software testing.

1.4.4 Test process attributes. This Standard describes attributes of the test process that indicate the quality of the testing performed. These attributes are selected to provide the means of assessing, comparing and improving test quality.

1.4.5 Generic test process. This Standard defines a generic test process. A generic process is chosen to ensure that this Standard is applicable to the diverse requirements of the software industry.

© British Computer Society, SIGIST, 2001

1.5 What this Standard does not cover

1.5.1 Types of testing. This Standard excludes a number of areas of software testing, for example:

- integration testing;
- system testing;
- user acceptance testing;
- statistical testing;
- testing of non-functional attributes such as performance;
- testing of real-time aspects;
- testing of concurrency;
- static analysis such as data flow or control flow analysis;
- reviews and inspections (even as applied to components and their tests).

A complete strategy for all software testing would cover these and other aspects.

1.5.2 Test completion criteria. This Standard does not prescribe test completion criteria as it is designed to be used in a variety of software development environments and application domains. Test completion criteria will vary according to the business risks and benefits of the application under test.

1.5.3 Selection of test case design techniques. This Standard does not prescribe which test case design techniques are to be used. Only appropriate techniques should be chosen and these will vary according to the software development environments and application domains.

1.5.4 Selection of test measurement techniques. This Standard does not prescribe which test measurement techniques are to be used. Only appropriate techniques should be chosen and these will vary according to the software development environments and application domains.

1.5.5 Personnel selection. This Standard does not prescribe who does the testing.

1.5.6 Implementation. This Standard does not prescribe how required attributes of the test process are to be achieved, for example, by manual or automated methods.

1.5.7 Fault removal. This Standard does not address fault removal. Fault removal is a separate process to fault detection.

1.6 Conformity

Conformity to this Standard shall be by following the testing process defined in clause 2.

1.7 Normative reference

The following standard contains provisions which, through reference in this text, constitute provisions of the Standard. At the time of publication, the edition was valid. All standards are subject to revision, and parties to agreements based on the Standard are encouraged to investigate the possibility of applying the most recent edition of the standard listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 9001:1994, Part 1: 1994 Specification for design/development, production, installation and servicing.

© British Computer Society, SIGIST, 2001

2 Process

2.1 Pre-requisites

Before component testing may begin the component test strategy (2.1.1) and project component test plan (2.1.2) shall be specified.

2.1.1 Component test strategy

2.1.1.1 The component test strategy shall specify the techniques to be employed in the design of test cases and the rationale for their choice. Selection of techniques shall be according to clause 3. If techniques not described explicitly in this clause are used they shall comply with the 'Other Testing Techniques' clause (3.13).

2.1.1.2 The component test strategy shall specify criteria for test completion and the rationale for their choice. These test completion criteria should be test coverage levels whose measurement shall be achieved by using the test measurement techniques defined in clause 4. If measures not described explicitly in this clause are used they shall comply with the 'Other Test Measurement Techniques' clause (4.13).

2.1.1.3 The component test strategy shall document the degree of independence required of personnel designing test cases from the design process, such as:

a) the test cases are designed by the person(s) who writes the component under test;

b) the test cases are designed by another person(s);

c) the test cases are designed by a person(s) from a different section;

d) the test cases are designed by a person(s) from a different organisation;

e) the test cases are not chosen by a person.

2.1.1.4 The component test strategy shall document whether the component testing is carried out using isolation, bottom-up or top-down approaches, or some mixture of these.

2.1.1.5 The component test strategy shall document the environment in which component tests will be executed. This shall include a description of the hardware and software environment in which all component tests will be run.

2.1.1.6 The component test strategy shall document the test process that shall be used for component testing.

2.1.1.7 The test process documentation shall define the testing activities to be performed and the inputs and outputs of each activity.

© British Computer Society, SIGIST, 2001

2.1.1.8 For any given test case, the test process documentation shall require that the following activities occur in the following sequence:

a) Component Test Planning;

b) Component Test Specification;

c) Component Test Execution;

d) Component Test Recording;

e) Checking for Component Test Completion.

2.1.1.9 Figure 2.1 illustrates the generic test process described in clause 2.1.1.8. Component Test Planning shall begin the test process and Checking for Component Test Completion shall end it; these activities are carried out for the whole component. Component Test Specification, Component Test Execution, and Component Test Recording may however, on any one iteration, be carried out for a subset of the test cases associated with a component. Later activities for one test case may occur before earlier activities for another.

2.1.1.10 Whenever an error is corrected by making a change or changes to test materials or the component under test, the affected activities shall be repeated.

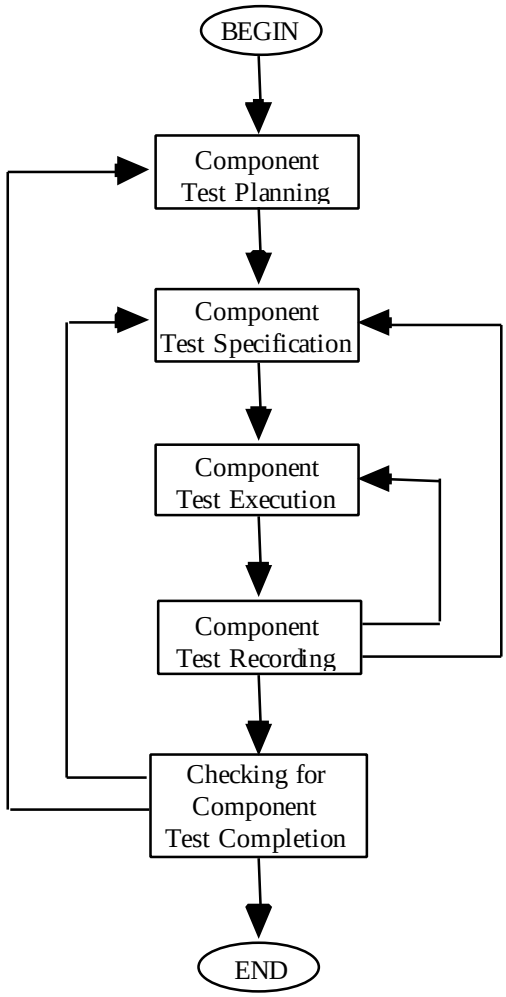


Figure 2.1 Generic Component Test Process

© British Computer Society, SIGIST, 2001

2.1.2 Project component test plan

2.1.2.1 The project component test plan shall specify the dependencies between component tests and their sequence. Their derivation shall include consideration of the chosen approach to component testing (2.1.1.4), but may also be influenced by overall project management and work scheduling considerations.

2.2 Component test planning

2.2.1 The component test plan shall specify how the component test strategy (2.1.1) and project component test plan (2.1.2) apply to the given component under test. This shall include specific identification of all exceptions to the component test strategy and all software with which the component under test will interact during test execution, such as drivers and stubs.

2.3 Component test specification

2.3.1 Test cases shall be designed using the test case design techniques selected in the test planning activity.

2.3.2 The specific test specification requirements for each test case design technique are defined in clause 3. Each test case shall be specified by defining its objective, the initial state of the component, its input, and the expected outcome. The objective shall be stated in terms of the test case design technique being used, such as the partition boundaries exercised.

2.3.3 The execution of each test case shall be repeatable.

2.4 Component test execution

2.4.1 Each test case shall be executed.

2.5 Component test recording

2.5.1 The test records for each test case shall unambiguously record the identities and versions of the component under test and the test specification. The actual outcome shall be recorded. It shall be possible to establish that the all specified testing activities have been carried out by reference to the test records.

2.5.2 The actual outcome shall be compared against the expected outcome. Any discrepancy found shall be logged and analysed in order to establish where the error lies and the earliest test activity that should be repeated in order to remove the discrepancy in the test specification or verify the removal of the fault in the component.

2.5.3 The test coverage levels achieved for those measures specified as test completion criteria shall be recorded.

2.6 Checking for component test completion

2.6.1 The test records shall be checked against the previously specified test completion criteria. If these criteria are not met, the earliest test activity that must be repeated in order to meet the criteria shall be identified and the test process shall be restarted from that point.

2.6.2 It may be necessary to repeat the Test Specification activity to design further test cases to meet a test coverage target.

3 Test Case Design Techniques

3.1 Equivalence Partitioning

3.1.1 Analysis. Equivalence partitioning uses a model of the component that partitions the input and output values of the component. The input and output values are derived from the specification of the component's behaviour.

The model shall comprise partitions of input and output values. Each partition shall contain a set or range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered 'equivalent'). Both valid and invalid values are partitioned in this way.

3.1.2 Design. Test cases shall be designed to exercise partitions. A test case may exercise any number of partitions. A test case shall comprise the following:

- the input(s) to the component;
- the partitions exercised;
- the expected outcome of the test case.

Test cases are designed to exercise partitions of valid values, and invalid input values. Test cases may also be designed to test that invalid output values cannot be induced.

3.2 Boundary Value Analysis

3.2.1 Analysis. Boundary Value Analysis uses a model of the component that partitions the input and output values of the component into a number of ordered sets with identifiable boundaries. These input and output values are derived from the specification of the component's behaviour.

The model shall comprise bounded partitions of ordered input and output values. Each partition shall contain a set or range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered 'equivalent'). Both valid and invalid values are partitioned in this way. A partition's boundaries are normally defined by the values of the boundaries between partitions, however where partitions are disjoint the minimum and maximum values in the range which makes up the partition are used. The boundaries of both valid and invalid partitions are considered.

3.2.2 Design. Test cases shall be designed to exercise values both on and next to the boundaries of the partitions. For each identified boundary three test cases shall be produced corresponding to values on the boundary and an incremental distance either side of it. This incremental distance is defined as the smallest significant value for the data type under consideration. A test case shall comprise the following:

- the input(s) to the component;
- the partition boundaries exercised;
- the expected outcome of the test case.

Test cases are designed to exercise valid boundary values, and invalid input boundary values. Test cases may also be designed to test that invalid output boundary values cannot be induced.

© British Computer Society, SIGIST, 2001

3.3 State Transition Testing

3.3.1 Analysis. State transition testing uses a model of the states the component may occupy, the transitions between those states, the events which cause those transitions, and the actions which may result from those transitions.

The model shall comprise states, transitions, events, actions and their relationships. The states of the model shall be disjoint, identifiable and finite in number. Events cause transitions between states, and transitions can return to the same state where they began. Events will be caused by inputs to the component, and actions in the state transition model may cause outputs from the component.

The model will typically be represented as a state transition diagram, state transition model, or a state table.

3.3.2 Design. Test cases shall be designed to exercise transitions between states. A test case may exercise any number of transitions. For each test case, the following shall be specified:

- the starting state of the component;
- the input(s) to the component;
- the expected outputs from the component;
- the expected final state.

For each expected transition within a test case, the following shall be specified:

- the starting state;
- the event which causes transition to the next state;
- the expected action caused by the transition;
- the expected next state.

Test cases are designed to exercise valid transitions between states. Test cases may also be designed to test that unspecified transitions cannot be induced.

3.4 Cause-Effect Graphing

3.4.1 Analysis. Cause-Effect Graphing uses a model of the logical relationships between causes and effects for the component. Each cause is expressed as a condition, which is either true of false (i.e. a Boolean) on an input, or combination of inputs, to the component. Each effect is expressed as a Boolean expression representing an outcome, or a combination of outcomes, for the component having occurred.

The model is typically represented as a Boolean graph relating the derived input and output Boolean expressions using the Boolean operators: AND, OR, NAND, NOR, NOT. From this graph, or otherwise, a decision (binary truth) table representing the logical relationships between causes and effects is produced.

3.4.2 Design. Test cases shall be designed to exercise rules, which define the relationship between the component's inputs and outputs, where each rule corresponds to a unique possible combination of inputs to the component that have been expressed as Booleans. For each test case the following shall be identified:

- Boolean state (i.e. true or false) for each cause;
- Boolean state for each effect.

3.5 Syntax Testing

3.5.1 Analysis. Syntax Testing uses a model of the formally-defined syntax of the inputs to a component.

The syntax is represented as a number of rules each of which defines the possible means of production of a symbol in terms of sequences of, iterations of, or selections between other symbols.

© British Computer Society, SIGIST, 2001

3.5.2 Design. Test cases with valid and invalid syntax are designed from the formally defined syntax of the inputs to the component.

Test cases with valid syntax shall be designed to execute options which are derived from rules which shall include those that follow, although additional rules may also be applied where appropriate:

- whenever a selection is used, an option is derived for each alternative by replacing the selection with that alternative;
- whenever an iteration is used, at least two options are derived, one with the minimum number of iterated symbols and the other with more than the minimum number of repetitions.

A test case may exercise any number of options. For each test case the following shall be identified:

- the input(s) to the component;
- option(s) exercised;
- the expected outcome of the test case.

Test cases with invalid syntax shall be designed as follows:

- a checklist of generic mutations shall be documented which can be applied to rules or parts of rules in order to generate a part of the input which is invalid;
- this checklist shall be applied to the syntax to identify specific mutations of the valid input, each of which employs at least one generic mutation;
- test cases shall be designed to execute specific mutations.

For each test case the following shall be identified:

- the input(s) to the component;
- the generic mutation(s) used;
- the syntax element(s) to which the mutation or mutations are applied;
- the expected outcome of the test case.

3.6 Statement Testing

3.6.1 Analysis. Statement testing uses a model of the source code which identifies statements as either executable or non-executable.

3.6.2 Design. Test cases shall be designed to exercise executable statements.

For each test case, the following shall be specified:

- the input(s) to the component;
- identification of statement(s) to be executed by the test case;
- the expected outcome of the test case.

3.7 Branch/Decision Testing

3.7.1 Analysis. Branch testing requires a model of the source code which identifies decisions and decision outcomes. A decision is an executable statement which may transfer control to another statement depending upon the logic of the decision statement. Typical decisions are found in loops and selections. Each possible transfer of control is a decision outcome.

3.7.2 Design. Test cases shall be designed to exercise decision outcomes.

For each test case, the following shall be specified:

- the input(s) to the component;
- identification of decision outcome(s) to be executed by the test case;
- the expected outcome of the test case.

© British Computer Society, SIGIST, 2001

3.8 Data Flow Testing

3.8.1 Analysis. Data Flow Testing uses a model of the interactions between parts of a component connected by the flow of data as well as the flow of control.

Categories are assigned to variable occurrences in the component, where the category identifies the definition or the use of the variable at that point. Definitions are variable occurrences where a variable is given a new value, and uses are variable occurrences where a variable is not given a new value, although uses can be further distinguished as either data definition P-uses or data definition C-uses. Data definition P-uses occur in the predicate portion of a decision statement such as while .. do, if .. then .. else, etc. Data definition C-uses are all others, including variable occurrences in the right hand side of an assignment statement, or an output statement.

The control flow model for the component is derived and the location and category of variable occurrences on it identified.

3.8.2 Design. Test cases shall be designed to execute control flow paths between definitions and uses of variables in the component.

Each test case shall include:

- the input(s) to the component;
- locations of relevant variable definition and use pair(s);
- control flow subpath(s) to be exercised;
- the expected outcome of the test case.

3.9 Branch Condition Testing

3.9.1 Analysis. Branch Condition Testing requires a model of the source code which identifies decisions and the individual Boolean operands within the decision conditions. A decision is an executable statement which may transfer control to another statement depending upon the logic of the decision statement. A decision condition is a Boolean expression which is evaluated to determine the outcome of a decision. Typical decisions are found in loops and selections.

3.9.2 Design. Test cases shall be designed to exercise individual Boolean operand values within decision conditions.

For each test case, the following shall be specified:

- the input(s) to the component;
- for each decision evaluated by the test case, identification of the Boolean operand to be exercised by the test case and its value;
- the expected outcome of the test case.

3.10 Branch Condition Combination Testing

3.10.1 Analysis. Branch Condition Combination Testing requires a model of the source code which identifies decisions and the individual Boolean operands within the decision conditions. A decision is an executable statement which may transfer control to another statement depending upon the logic of the decision statement. A decision condition is a Boolean expression which is evaluated to determine the outcome of a decision. Typical decisions are found in loops and selections.

3.10.2 Design. Test cases shall be designed to exercise combinations of Boolean operand values within decision conditions.

For each test case, the following shall be specified:

- the input(s) to the component;
- for each decision evaluated by the test case, identification of the combination of Boolean operands to be exercised by the test case and their values;
- the expected outcome of the test case.

3.11 Modified Condition Decision Testing

3.11.1 Analysis. Modified Condition Decision Testing requires a model of the source code which identifies decisions, outcomes, and the individual Boolean operands within the decision conditions. A decision is an executable statement which may transfer control to another statement depending upon the logic of the decision statement. A decision condition is a Boolean expression which is evaluated to determine the outcome of a decision. Typical decisions are found in loops and selections.

3.11.2 Design. Test cases shall be designed to demonstrate that Boolean operands within a decision condition can independently affect the outcome of the decision.

For each test case, the following shall be specified:

- the input(s) to the component;
- for each decision evaluated by the test case, identification of the combination of Boolean operands to be exercised by the test case, their values, and the outcome of the decision;
- the expected outcome of the test case

3.12 LCSAJ Testing

3.12.1 Analysis. LCSAJ testing requires a model of the source code which identifies control flow jumps (where control flow does not pass to a sequential statement). An LCSAJ (Linear Code Sequence and Jump) is defined by a triple, conventionally identified by line numbers in a source code listing: the start of the linear code sequence, the end of the linear code sequence, and the target line to which control flow is transferred.

3.12.2 Design. Test cases shall be designed to exercise LCSAJs.

For each test case, the following shall be specified:

- the input(s) to the component;
- identification of the LCSAJ(s) to be executed by the test case;
- the expected outcome of the test case.

3.13 Random Testing

3.13.1 Analysis. Random Testing uses a model of the input domain of the component that defines the set of all possible input values. The input distribution (normal, uniform, etc.) to be used in the generation of random input values shall be based on the expected operational distribution of inputs. Where no knowledge of this operational distribution is available then a uniform input distribution shall be used.

3.13.2 Design. Test cases shall be chosen randomly from the input domain of the component according to the input distribution.

A test case shall comprise the following:
- the input(s) to the component;
- the expected outcome of the test case.

The input distribution used for the test case suite shall also be recorded.

3.14 Other Testing Techniques

Other test case design techniques may be used that are not listed in this clause. Any alternative techniques used shall satisfy these criteria:

a) The technique shall be available in the public domain and shall be referenced.
b) The test case design technique shall be documented in the same manner as the other test case design techniques in clause 3.
c) Associated test measurement techniques may be defined as described in clause 4.13.

© British Computer Society, SIGIST, 2001

4 Test Measurement Techniques

In each coverage calculation, a number of coverage items may be infeasible. A coverage item is defined to be infeasible if it can be demonstrated to be not executable. The coverage calculation shall be defined as either counting or discounting infeasible items - this choice shall be documented in the test plan. If a coverage item is discounted justification for its infeasibility shall be documented in the test records.

In each coverage calculation, if there are no coverage items in the component under test, 100% coverage is defined to be achieved by one test case.

4.1 Equivalence Partition Coverage

4.1.1 Coverage Items. Coverage items are the partitions described by the model (see 3.1.1).

4.1.2 Coverage Calculation. Coverage is calculated as follows:

$$\text{Equivalence partition coverage} = \frac{\text{number of covered partitions}}{\text{total number of partitions}} * 100\%$$

4.2 Boundary Value Coverage

4.2.1 Coverage Items. The coverage items are the boundaries of partitions described by the model (see 3.2.1). Some partitions may not have an identified boundary, for example, if a numerical partition has a lower but not an upper bound.

4.2.2 Coverage Calculation. Coverage is calculated as follows:

$$\text{Boundary value coverage} = \frac{\text{number of distinct boundary values executed}}{\text{total number of boundary values}} * 100\%$$

where a boundary value corresponds to a test case on a boundary or an incremental distance either side of it (see 3.2.2).

4.3 State Transition Coverage

4.3.1 Coverage Items. Coverage items are sequences of one or more transitions between states on the model (see 3.3.1).

4.3.2 Coverage Calculation. For single transitions, the coverage metric is the percentage of all valid transitions exercised during test. This is known as 0-switch coverage. For n transitions, the coverage measure is the percentage of all valid sequences of n transitions exercised during test. This is known as (n - *1*) switch coverage.

4.4 Cause-Effect Coverage

4.4.1 Coverage Items. Coverage items are rules, where each rule represents a unique possible combination of inputs to the component that have been expressed as Booleans.

4.4.2 Coverage Calculation. Coverage is calculated as follows:

$$\text{Cause - Effect Coverage} = \frac{\text{Number of rules exercised}}{\text{Total number of rules}} * 100\%$$

4.5 Syntax Coverage

No coverage measure is defined for syntax testing.

4.6 Statement Coverage

4.6.1 Coverage Items. Coverage items are executable statements in the source code.

4.6.2 Coverage Calculation. Coverage is calculated as follows:

$$\text{Statement Coverage} = \frac{\text{Number of executable statements executed}}{\text{Total number of executable statements}} * 100\%$$

4.7 Branch and Decision Coverage

4.7.1 Branch Coverage Items. A *branch* is:

- a conditional transfer of control from any statement to any other statement in the component;
- an unconditional transfer of control from any statement to any other statement in the component except the next statement;
- when a component has more than one entry point, a transfer of control to an entry point of the component.

An entry point is either the first statement of the component or any other statement which may be branched to from outside the component.

4.7.2 Branch Coverage Calculation. Coverage is calculated as follows:

$$\text{Branch Coverage} = \frac{\text{number of executed branches}}{\text{total number of branches}} * 100\%$$

4.7.3 Decision Coverage Items. Decision Coverage uses the model of the component described for Branch Testing in clause 3.7.1. Coverage items are decision outcomes.

Decision Coverage is only defined for components with one entry point.

4.7.4 Decision Coverage Calculation. Coverage is calculated as follows:

$$\text{Decision Coverage} = \frac{\text{number of executed decision outcomes}}{\text{total number of decision outcomes}} * 100\%$$

4.8 Data Flow Coverage

4.8.1 Coverage Items. The coverage items are the control flow subpaths from a variable definition to the variable's corresponding p-uses, c-uses, or their combination.

4.8.2 Coverage Calculation. For the purposes of these coverage calculations a definition-use pair is defined as a *simple subpath* between a definition of a variable and a use of that variable and coverage is calculated using the formula:

Coverage = (N/T) * 100%, where N and T are defined in the subsequent subclauses.

A *simple subpath* is a subpath through a component's control flow graph where no parts of the subpath are visited more than necessary.

4.8.2.1 All-definitions. This measure is defined with respect to the traversal of the set of subpaths from each variable definition to some use (either p-use or c-use) of that definition.

N = Number of exercised definition-use pairs from distinct variable definitions
T = Number of variable definitions

© British Computer Society, SIGIST, 2001

4.8.2.2 All-c-uses. This measure is defined with respect to the traversal of the set of subpaths from each variable definition to every c-use of that definition.

N = Number of exercised definition-c-use pairs
T = Number of definition-c-use pairs

4.8.2.3 All-p-uses. This measure is defined with respect to the traversal of the set of subpaths from each variable definition to every p-use of that definition.

N = Number of exercised definition-p-use pairs
T = Number of definition-p-use pairs

4.8.2.4 All-uses. This measure is defined with respect to the traversal of the set of subpaths from each variable definition to every use (both p-use and c-use) of that definition.

N = Number of exercised definition-use pairs
T = Number of definition-use pairs

4.8.2.5 All-du-paths. This measure is defined with respect to the traversal of the set of subpaths from each variable definition to every use (both p-use and c-use) of that definition.

N = Number of exercised simple subpaths between definition-use pairs
T = Number of simple subpaths between definition-use pairs

4.9 Branch Condition Coverage

4.9.1 Coverage Items. Branch Condition Coverage uses a model of the component described in clause 3.9.1. Coverage items are Boolean operand values within decision conditions.

Branch Condition Coverage is only defined for components with one entry point.

4.9.2 Coverage Calculation. Coverage is calculated as follows:

$$\text{Branch Condition Coverage} = \frac{\text{number of Boolean operand values executed}}{\text{total number of Boolean operand values}} * 100\%$$

4.10 Branch Condition Combination Coverage

4.10.1 Coverage Items. Branch Condition Combination Coverage uses a model of the component described in clause 3.10.1. Coverage items are unique combinations of the set of Boolean operand values within each decision condition.

Branch Condition Combination Coverage is only defined for components with one entry point.

4.10.2 Coverage Calculation. Coverage is calculated as follows:

Branch Condition Combination Coverage

$$= \frac{\text{number of Boolean operand value combinations executed}}{\text{total number of Boolean operand value combinations}} * 100\%$$

4.11 Modified Condition Decision Coverage

4.11.1 Coverage Items. Modified Condition Decision Coverage uses a model of the component described in clause 3.11.1. Coverage items are Boolean operand values within decision conditions.

Modified Condition Decision Coverage is only defined for components with one entry point.

4.11.2 Coverage Calculation. Coverage is calculated as follows:

Modified Condition Decision Coverage

$$= \frac{\text{number of Boolean operand values shown to independently affect the decision}}{\text{total number of Boolean operands}} * 100\ \%$$

4.12 LCSAJ Coverage

4.12.1 Coverage Items. Coverage items are LCSAJs for the component (see 3.12.1).

4.12.2 Coverage Calculation Coverage is calculated as follows:

$$\text{LCSAJ Coverage} = \frac{\text{number of executed LCSAJs}}{\text{total number of LCSAJs}} * 100\%$$

4.13 Random Testing

No coverage measure is defined for random testing.

4.14 Other Test Measurement Techniques

Other test measurement techniques may be used that are not listed in this clause. Any alternative techniques used shall satisfy these criteria:

a) The technique shall be available in the public domain and shall be referenced.
b) The test measurement technique shall be documented in the same manner as the other test measurement techniques in clause 4.
c) Associated test case design techniques may be defined as described in clause 3.13.

© British Computer Society, SIGIST, 2001

Annex A Process Guidelines (informative)

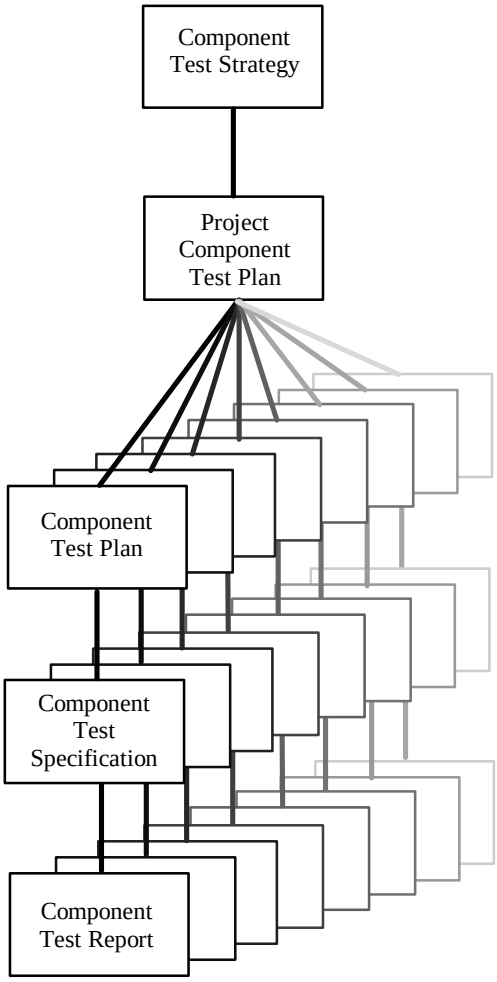
The purpose of this annex is to provide guidance on the requirements specified in clause 2 of this Standard.

These guidelines are supported by the use of example documentation from a fictitious project, named EX. The example documentation provides a basic sample of the full set of documentation which conforms to this Standard. The technical content of the example documentation merely serves to illustrate one interpretation of this Standard and does not suggest that this interpretation is to be preferred over any other. Any set of documentation which meets the requirements of clause 2 of this Standard is acceptable.

The test documentation examples included are:

- Component Test Strategy;
- Project Component Test Plan;
- Component Test Plan;
- Component Test Specification;
- Component Test Report.

The relationships between the different forms of example test documentation can be seen in the following Document Hierarchy:



The Document Hierarchy shows that there is a single Component Test Strategy for Project EX (in this example documented within the project's Quality Plan) and a single Project Component Test Plan for the project. For each component there is a Component Test Plan and corresponding Component Test Specification and Component Test Report.

A.1 Pre-Requisites

Clause 2 of the Standard specifies requirements for the overall test process including component testing. The prerequisites for component testing are an overall Component Test Strategy and a project specific Project Component Test Plan.

A.1.1 Component Test Strategy

A.1.1.1 To comply with this Standard, component testing must be carried out within a pre-defined documented strategy. Many organisations will achieve this through a corporate or divisional Testing Manual or Quality Manual. In other cases, the Component Test Strategy could be defined for one project only and may be presented as part of a specific project Quality Plan.

A.1.1.2 This Standard does not prescribe that the component test strategy documentation need be a single document or a whole document. In the case where a Component Test Strategy is tailored to a specific project the component test strategy documentation could be incorporated into the project component test planning documentation.

A.1.1.3 This Standard requires that the Component Test Strategy is defined in advance but does not prescribe the format of component test strategy documentation.

A.1.1.4 As part of the test strategy, this Standard defines a generic test process comprising a series of test activities and requires this sequence of activities to be followed for a particular test case. This allows, for instance, incremental approaches such as planning, executing and recording the tests for one set of components before another; or planning, executing and recording the application of one test case design technique before another.

A.1.1.5 Further requirements on the management of the testing are given in ISO-9001. Guidance on setting up a test process is available in [IEEE 1008].

A1.1.6 The form of the test completion criterion will typically be a target for test coverage.

A.1.1.7 The following example provides a Component Test Strategy for Project EX developing the EX system. For the purposes of the example, the architectural design process decomposes the system into *functional areas* each containing a number of components. The example Component Test Strategy is provided as a section in the project's Quality Plan.

(Example)

Project EX Quality Plan

Section 6

Component Test Strategy

This section provides the component test strategy for Project EX. Any exceptions to this strategy (for example: omitted or additional design or measurement techniques) must be documented in the Component Test Plan.

Exceptions to this strategy must be approved by the Project Manager and the Quality Manager. The use of any additional design or measurement techniques, where appropriate, is encouraged and can be done without specific approval.

© British Computer Society, SIGIST, 2001

(Example)

1. **Design Techniques**: Component tests shall be designed using the following techniques:

   - Equivalence Partitioning (EP, Std 3.1);
   - Boundary Value Analysis (BVA, Std 3.2);
   - Decision Testing (DT, Std 3.7).

Rationale: EP and BVA have proven cost-effective on previous releases. DT shall be used to complete coverage should EP and BVA fail to meet the coverage completion criteria.

2. **Completion criteria**: A minimum of 100% Equivalence Partition Coverage (EPC, Std 4.1), 100% Boundary Value Coverage (BVC, Std 4.2) and 90% Decision Coverage (DC, Std 4.7, 4.7.3 & 4.7.4) shall be achieved for each component test.

Rationale: Equivalence Partition and Boundary Value Coverage are used to ensure full application of the corresponding test case design techniques, while Decision coverage provides a useful white box check on the overall set of test cases.

3. **Independence**: No independence is required in the production of test plans and test specifications, or in the implementation of tests. All test documentation including test records must be reviewed by the member of the project to whom the originator of the work reports.

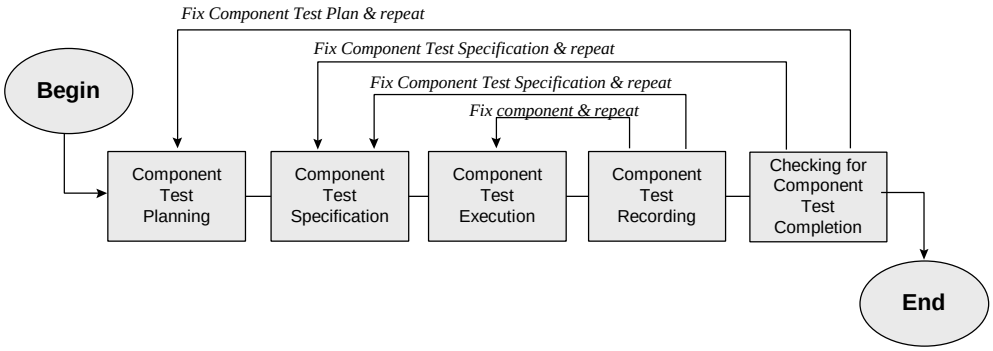
4. **Approach**: Components shall be tested in isolation using stubs and drivers in place of interfacing components.

5. **Environment**: Component tests shall be run in the developer's own workspace. All output shall be logged to file. The Coverage Tool(1) shall be used to analyse the test coverage of each test run.

6. **Error Correction**: Wherever possible, test cases for a component should be executed together in a single test run. The outcome of all test cases should then be analysed. If corrections are needed then the entire component test should be repeated.

7. **Project Component Test Plan**: A Project Component Test Plan shall be produced following completion of the architectural design of the EX system (Std 2.1.2).

8. **Component Test Process**: Items 10 to 14 below detail the component test process.



9. **Component Test Plan**: Individual component tests will be planned following completion of the design for the functional area which a related group of components comprise.

A Component Test Plan can be for a single component or a group of related components which comprise a functional area of the system. The Component Test Plan shall list the test case design techniques to be used when specifying tests for the component, the measurement techniques to be used when executing tests, stubs, drivers and specific test completion criteria.

Exceptions to this Component Test Strategy shall be explicitly identified and approved where approval is required by this Component Test Strategy.

Inputs: Architectural Design, Detailed Design, Project Component Test Plan, Quality Plan;

Outputs: Component Test Plan.

10. **Component Test Specification**: The component test specification shall provide a list of test cases annotated with the associated design elements which each test case exercises. This will help the related measurement criteria to be assessed.

Prior to test execution, the test specification shall be reviewed for completeness.

Inputs: Architectural Design, Detailed Design, Component Test Plan;

Outputs: Component Test Specification.

(Example)

11. **Component Test Execution**: Test execution shall be prepared by coding the drivers and stubs specified in the test plan, compiling and linking with the component under test. The test driver shall include code to control tests and create a log file. Tests are then run.

The Coverage Tool(1) shall be used during test execution to analyse test coverage.

The objective of test execution shall be to execute all specified test cases. Test execution shall complete when either all test cases have been executed or an error is encountered which prevents continuation. Test execution should not be halted for minor errors (the error should be recorded and test execution continued).

Inputs: Component Test Plan, Component Test Specification, Component Code;

Outputs: Test outcomes, Test log file, Coverage analysis report, stub and driver code.

12. **Component Test Recording**: The test log file shall be examined by the tester to compare actual test outcomes with expected test outcomes. Differences shall be investigated. Any due to software or test errors will raise a fault report. Where a test is incomplete, this shall cause a regression in the test process. Where applicable, correction shall be made to the component design and/or code as necessary to correct for fault reports raised during the Component Test Recording activity. The test process shall then be repeated.

A Component Test Record shall be produced each time a test is run, containing the version of component under test, version of the Component Test Specification, date and time of test, the number of test cases run, number of test discrepancies, coverage measurements and cross-references to any fault reports raised.

Inputs: Test Plan, Test Specification, Test log file, Coverage analysis report, Component Code;

Outputs: Component Test Record, Fault Reports.

13. **Checking for Component Test Completion**: The Component Test Record shall be marked to show whether the overall test has passed and completion criteria have been met.

Inputs: Component Test Plan (specifying completion criteria), Component Test Record, Fault Reports;

Outputs: Component Test Record (pass/ fail).

Where coverage has not been achieved, the Component Test Specification will normally be extended with further test cases until the required coverage level is achieved. Exceptionally, with the approval of the Project Manager and the Quality Manager, the completion criteria in the Component Test Plan may be changed to the achieved level.

A.1.1.9 **Notes:**

(1) For the purposes of this example, a generic test coverage measurement tool has been used called "Coverage Tool". In practice, a test strategy would explicitly give the name of any tool used.

A.1.2 Project Component Test Planning

A.1.2.1 This Standard does not prescribe that the Project Component Test Plan documentation need be a single document or a whole document.

A.1.2.2 As a minimum, the Project Component Test Plan should identify any specific adaptations of the component test strategy and to specify any dependencies between component tests.

A.1.2.3 Further guidance on test planning is available in [IEEE 829].

A.1.2.4 The following example provides a Project Component Test Plan for project EX. Strictly speaking, there are no dependencies between component tests because all components are tested in isolation. The hypothetical project EX desires to begin the integration of tested components before all component testing is complete. Thus the sequence of component testing is driven by the requirements of integration testing.

(Example)

Project EX

Project Component Test Plan

1. **Dependencies**: Strict adherence to the Component Test Strategy (isolation testing) removes any

© British Computer Society, SIGIST, 2001

(Example)

dependencies between component tests. Nevertheless, consideration for subsequent integration means that it will be most convenient to complete the component testing of some parts of the system before that of other parts.

The approach selected is to implement the kernel of each functional area of the system so that a minimal working thread can be established as early as possible.

The component test sequence will be:

- LOG 1 - LOG 6;
- REM 1 - REM 6 (timing critical);
- RES 1 - RES 4;
- MIS 1 - MIS 5.

Integration testing can now start in parallel with the remaining component tests. Remaining components will be tested in the sequence:

- RES 5 - RES 8;
- MIS 6 - MIS 10;
- LOG 7 - LOG 12;
- MIS 11 - MIS 20.

A.2 Component Test Planning

A.2.1 This Standard does not prescribe that the Component Test Plan documentation need be a single document or a whole document.

A.2.2 The form of the test completion criterion is not mandated. This will typically be a test coverage target which has been mandated by the Component Test Strategy.

A.2.3 Further guidance on test planning is available in [IEEE 829].

A.2.4 An example test plan for component LOG 3 of project EX is shown below. This could be contained in the same physical document as the previous document, the Project Component Test Plan for all components.

(Example)

Project EX

Component Test Plan

for

Component LOG 3

1. **Design techniques**: EP (Std 3.1), BVA (Std 3.2), DT (Std 3.7).
2. **Measurement Techniques**: EPC (Std 4.1), BVC (Std 4.2) and DC (Std 4.7.3, 4.7.4)
3. **Completion Criteria**:
   - EPC 100% (Std 4.1);
   - BVC 100% (Std 4.2);
   - DC 100% (Std 4.7.3, 4.7.4).

(Example)

Note: the Component Test Strategy requires only 90% Decision coverage, but this is a critical component[1].

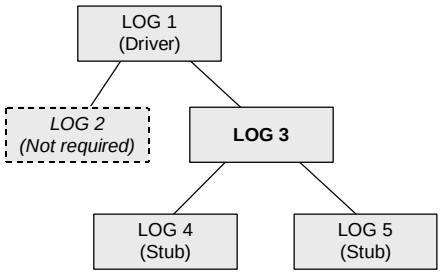
4. **Stubs**: for components LOG 4 and LOG 5 (which are called by LOG 3)[2].
5. **Driver**: in place of component LOG 1 (which calls LOG 3)[2].
6. **Exceptions**: The raised completion criteria of 100% decision coverage does not require approval[1].

Exception approval - not applicable.

A.2.5 **Notes:**

(1) The example Test Strategy has stated that any additional coverage does not require explicit approval.

(2) The calling tree for functional area "LOG" is that "LOG 1" calls "LOG 2" and "LOG 3". "LOG 3" calls "LOG 4" and "LOG 5". As "LOG 2" is not called by "LOG 3", it is not required to test "LOG 3"



A.3 Test Specification

A.3.1 The test specification for a component may be in machine-readable form; some test automation tools employ scripting languages that may meet the requirements of this Standard.

A.3.2 An example Test Specification for component LOG 3 in project EX is given below. Note that the specification has been extended (version 2) to cater for gaps in the coverage of the initial specification and errors as reported in the test record (A.5).

(Example)

Project EX

Component Test Specification

for

Component LOG 3 (Issue A)[1]

Test design table

Test Objective: Input Validation

Initial state of component for each test case: start of new entry

© British Computer Society, SIGIST, 2001

(Example)

	Conditions							
	Valid Classes	Tag	Invalid Classes	Tag	Valid Boundaries	Tag	Invalid Boundaries	Tag
Inputs	1-7	V1	< 1 > 7 non-digit	X1 X2 X3	1 7	VB1 VB2	0 8	XB1 XB2
Outcomes	Beginner (1-5) Advanced(6-7)	V2 V3			5 6	VB3 VB4		

Test cases (2)

Test Case	Input Values	Expected Outcome	New Conditions covered (cross referenced to "Tag" in Test Design Table")
1	3	Beginner	V1, V2
2	7	Advanced	V3, VB2
3	1	Beginner	VB1
4	0	Error message	X1, XB1
5	9	Error message	X2
6	8	Error message	XB2
7	A	Error message	X3
8	5	Beginner	VB3
9	6	Advanced	VB4

Additional/Altered Test cases (Test Specification Version 2) (3)

Test Case	Description	Expected Outcome	Reason
2	7	Experienced	Test fault
9	6	Experienced	Test fault
10	<return> (no input)	Re-display entry screen	Decision Coverage

A.3.3 **Notes:**

(1) The tests are for Issue A of component LOG 3, a component which has not yet been issued, but will be given Issue A status when it has passed its component test.

(2) Test cases have covered all tags in the Test Design Table with at least one test case, so we have 100% coverage of partitions and boundaries.

(3) When the test cases were executed (see example Component Test Record in A.5.3), two test cases failed due to errors in the test case (2, 9), where the wrong term for the expected outcome ("Advanced" instead of "Experienced") was used. Test specification version 2 corrected these test cases and added one further test case to achieve 100% decision coverage.

A.4 Test Execution

A.4.1 Test execution and coverage measurement are often performed by tools. This Standard does not prescribe how such tools work.

A.5 Test Recording

A.5.1 In the following example Test Report, component LOG 3 of project EX fails the first execution of tests 1 to 9. Test cases 2 and 9 fail due to errors in the test specification. These are corrected and a further test case is specified to complete decision coverage.

A.5.2 After recording results each outcome is classified. Typically, one of the following situations will be recorded:

a) A test has been incorrectly performed and must be repeated;
b) A test has detected a fault in the component; the component must be changed, and the test repeated;
c) A test has detected a fault in the specification of the component; the specification of the component, the component and the test specification (possibly) must be changed, and the test repeated;
d) A test has detected a fault in the test specification; the test specification must be changed, and the test repeated;
e) A test executes and produces the expected outcome; further test cases are performed, if any remain.

A.5.3 It is possible that there is insufficient information to make the classification at this point in the process. In this instance, it may be necessary to extend the test specification and prepare and execute more test cases before deciding what to do. A record of all decisions made should be documented as part of the Test Report.

(Example)

Project EX

Component Test Report

for

Component LOG 3 (Issue A)(1)

Date: 17/1/1997 by D. Tester
Component: LOG 3, Issue A(1)
Component Test Specification LOG 3 Version 1

Test cases executed: 1, 2, 3, 4, 5, 6, 7, 8, 9.
Failed Test cases

Test Case	Expected Outcome	Actual Outcome
2	Advanced	Experienced
9	Advanced	Experienced

Fault report TFR23 raised. Test cases 2 and 9 state incorrect expected outcome.
Decision Coverage achieved: 83%

Date: 20/1/1997 by D. Tester
Component: LOG 3, Issue A(1)

© British Computer Society, SIGIST, 2001

(Example)

Component Test Specification LOG 1 Version 2(2)
Test cases executed: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
All test cases matched expected outcome
Decision Coverage achieved: 100%
Test passed

A.5.4 **Notes:**

(1) The tests are for Issue A of component "LOG 3", a component which has not yet been issued, but will be given Issue A status when it has passed its component test.

(2) When the test cases were executed, two test cases failed due to errors in the test cases (2, 9). Test specification version 2 corrected these test cases (see example Test Specification A.4.2) and added one further test case to achieve 100% decision coverage.

A.6 Checking for Test Completion

A.6.1 A check against the test completion criteria is mandatory at this point. If the criteria are not met, normally additional tests shall be required, alternatively, by allowing iteration of Test Planning, the Standard implicitly permits the relaxation (or strengthening) of test completion criteria. Any changes to the test completion criteria should be documented.

A.6.2 If all test completion criteria are met then the component is released for integration.

Annex B Guidelines for Testing Techniques and Test Measurement (informative)

The purpose of this clause is to provide guidance on the requirements specified in clauses 3 and 4 of this Standard.

Each test technique is based upon a model of the component. Since the easiest way to understand the model is by means of an example, the guidance provided here consists mainly of examples. A variety of applications and programming languages are used. The specification of a component is highlighted by means of an italic font. Functional test case design techniques and measures may be used for any development environment. Structural test case design techniques and measures are based on the underlying structure of the development language used. Those described in this Standard are well-established for procedural 3GL programming languages.

B.1 Equivalence Partitioning

Introduction

Equivalence partitioning is based on the premise that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component. Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition.

Example

Consider a component, *generate_grading*, with the following specification:

> *The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:*
>
> | | | |
> |---|---|---|
> | *greater than or equal to 70* | - | *'A'* |
> | *greater than or equal to 50, but less than 70* | - | *'B'* |
> | *greater than or equal to 30, but less than 50* | - | *'C'* |
> | *less than 30* | - | *'D'* |
>
> *Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.*

Initially the equivalence partitions are identified and then test cases derived to exercise the partitions. Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

The partitions for the two inputs are initially identified. The *valid* partitions can be described by:

```
0 <= exam mark <= 75
0 <= coursework mark <= 25
```

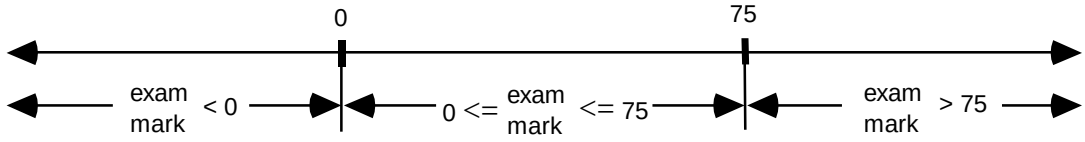
The most obvious *invalid* partitions based on the inputs can be described by:

```
exam mark > 75
exam mark < 0
coursework mark > 25
```

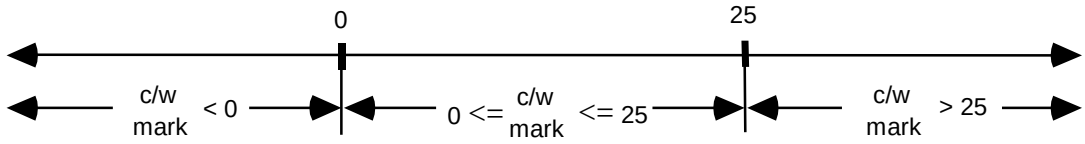

© British Computer Society, SIGIST, 2001

coursework mark < 0

Partitioned ranges of values can be represented pictorially, therefore, for the input, exam mark, we get:



And for the input, coursework mark, we get:



Less obvious invalid input equivalence partitions would include any other inputs that can occur not so far included in a partition, for instance, non-integer inputs or perhaps non-numeric inputs. So, we could generate the following invalid input equivalence partitions:

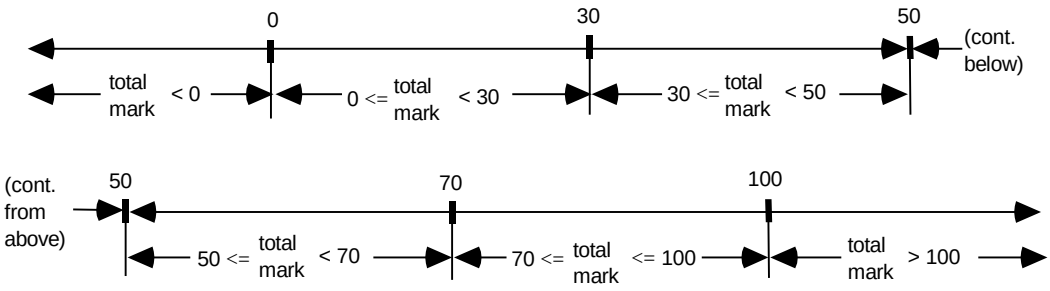
exam mark = real number (a number with a fractional part)
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic

Next, the partitions for the outputs are identified. The *valid* partitions are produced by considering each of the valid outputs for the component:

'A'	is induced by	70 <= total mark <= 100
'B'	is induced by	50 <= total mark < 70
'C'	is induced by	30 <= total mark < 50
'D'	is induced by	0 <= total mark < 30
'Fault Message'	is induced by	total mark > 100
'Fault Message'	is induced by	total mark < 0

where total mark = exam mark + coursework mark. Note that 'Fault Message' is considered as a valid output as it is a *specified* output.

The equivalence partitions and boundaries for total mark are shown pictorially below:



An invalid output would be any output from the component other than one of the five specified. It is difficult to identify unspecified outputs, but obviously they must be considered as if we can cause one then we have identified a flaw with either the component, its specification, or both. For this example

three unspecified outputs were identified and are shown below. This aspect of equivalence partitioning is very subjective and different testers will inevitably identify different partitions which *they* feel could possibly occur.

output = 'E'
output = 'A+'
output = 'null'

Thus the following nineteen equivalence partitions have been identified for the component (remembering that for some of these partitions a certain degree of subjective choice was required, and so a different tester would not necessarily duplicate this list exactly):

0 <= exam mark <= 75
exam mark > 75
exam mark < 0
0 <= coursework mark <= 25
coursework mark > 25
coursework mark < 0
exam mark = real number
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic
70 <= total mark <= 100
50 <= total mark < 70
30 <= total mark < 50
0 <= total mark < 30
total mark > 100
total mark < 0
output = 'E'
output = 'A+'
output = 'null'

Having identified all the partitions then test cases are derived that 'hit' each of them. Two distinct approaches can be taken when generating the test cases. In the first a test case is generated for each identified partition on a one-to-one basis, while in the second a minimal set of test cases is generated that cover all the identified partitions.

The one-to-one approach will be demonstrated first as it can make it easier to see the link between partitions and test cases. For each of these test cases only the single partition being targetted is stated explicitly. Nineteen partitions were identified leading to nineteen test cases.

The test cases corresponding to partitions derived from the input exam mark are:

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	0 <= e <= 75	e < 0	e > 75
Exp. Output	'B'	'FM'	'FM'

Note that the input coursework (c/w) mark has been set to an arbitrary valid value of 15.

© British Computer Society, SIGIST, 2001

The test cases corresponding to partitions derived from the input coursework mark are:

Test Case	4	5	6
Input (exam mark)	40	40	40
Input (c/w mark)	8	-15	47
total mark (as calculated)	48	25	87
Partition tested (of c/w mark)	0 <= c <= 25	c < 0	c > 25
Exp. Output	'C'	'FM'	'FM'

Note that the input exam mark has been set to an arbitrary valid value of 40.

The test cases corresponding to partitions derived from possible invalid inputs are:

Test Case	7	8	9	10
Input (exam mark)	48.7	q	40	40
Input (c/w mark)	15	15	12.76	g
total mark (as calculated)	63.7	not applicable	52.76	not applicable
Partition tested	exam mark = real number	exam mark = alphabetic	c/w mark = real number	c/w mark = alphabetic
Exp. Output	'FM'	'FM'	'FM'	'FM'

The test cases corresponding to partitions derived from the valid outputs are:

Test Case	11	12	13
Input (exam mark)	-10	12	32
Input (c/w mark)	-10	5	13
total mark (as calculated)	-20	17	45
Partition tested (of total mark)	t < 0	0 <= t < 30	30 <= t < 50
Exp. Output	'FM'	'D'	'C'

Test Case	14	15	16
Input (exam mark)	44	60	80
Input (c/w mark)	22	20	30
total mark (as calculated)	66	80	110
Partition tested (of total mark)	50 <= t < 70	70 <= t <= 100	t > 100
Exp. Output	'B'	'A'	'FM'

The input values of exam mark and coursework mark have been derived from the total mark, which is their sum.

The test cases corresponding to partitions derived from the invalid outputs are:

Test Case	17	18	19
Input (exam mark)	-10	100	null
Input (c/w mark)	0	10	null
total mark (as calculated)	-10	110	null+null
Partition tested (output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'

It should be noted that where invalid input values are used (as above, in test cases 2, 3, 5-11, and 16-19) it may, depending on the implementation, be impossible to actually execute the test case. For instance, in Ada, if the input variable is declared as a positive integer then it will not be possible to assign a negative value to it. Despite this, it is still worthwhile *considering* all the test cases for completeness.

It can be seen above that several of the test cases are similar, such as test cases 1 and 14, where the main difference between them is the partition targetted. As the component has two inputs and one output, each test case actually 'hits' three partitions; two input partitions and one output partition.. Thus it is possible to generate a smaller 'minimal' test set that still 'hits' all the identified partitions by deriving test cases that are designed to exercise more than one partition. The following test case suite of eleven test cases corresponds to the minimised test case suite approach where each test case is designed to hit as many new partitions as possible rather than just one. Note that here all three partitions are explicitly identified for each test case.

Test Case	1	2	3	4
Input (exam mark)	60	40	25	15
Input (c/w mark)	20	15	10	8
total mark (as calculated)	80	55	35	23
Partition (of exam mark)	0 <= e <= 75	0 <= e <= 75	0 <= e <= 75	0 <= e <= 75
Partition (of c/w mark)	0 <= c <= 25	0 <= c <= 25	0 <= c <= 25	0 <= c <= 25
Partition (of total mark)	70 <= t <= 100	50 <= t < 70	30 <= t < 50	0 <= t < 30
Exp. Output	'A'	'B'	'C'	'D'

Test Case	5	6	7	8
Input (exam mark)	-10	93	60.5	q
Input (c/w mark)	-15	35	20.23	g
total mark (as calculated)	-25	128	80.73	-
Partition (of exam mark)	e < 0	e > 75	e = real number	e = alphabetic
Partition (of c/w mark)	c < 0	c > 25	c = real number	c = alphabetic
Partition (of total mark)	t < 0	t > 100	70 <= t <= 100	-
Exp. Output	'FM'	'FM'	'FM'	'FM'

© British Computer Society, SIGIST, 2001

Test Case	9	10	11
Input (exam mark)	-10	100	'null'
Input (c/w mark)	0	10	'null'
total mark (as calculated)	-10	110	null+null
Partition (of exam mark)	e < 0	e > 75	-
Partition (of c/w mark)	0 <= c <= 25	0 <= c <= 25	-
Partition (of total mark)	t < 0	t > 100	-
Partition (of output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'

The one-to-one and minimised approaches represent the two extremes of a spectrum of approaches to equivalence partitioning. The disadvantage of the one-to-one approach is that it requires more test cases and if this causes problems a more minimalist approach can be used. Normally, however, the identification of partitions is far more time consuming than the generation and execution of test cases themselves and so any savings made by reducing the size of the test case suite are relatively small compared with the overall cost of applying the technique. The disadvantage of the minimalist approach is that in the event of a test failure it can be difficult to identify the cause due to several new partitions being exercised at once. This is a debugging problem rather than a testing problem, but there is no reason to make debugging more difficult than it is already.

Both of the above test case suites achieve 100% equivalence partition coverage as each ensures that all nineteen identified partitions are exercised by at least one test case. Lower levels of coverage would be achieved if all the partitions identified are not all exercised. If all the partitions are not identified, then any coverage measure based on this incomplete set of partitions would be misleading.

B.2 Boundary Value Analysis

Introduction

Boundary Value Analysis is based on the following premise. Firstly, that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component and, secondly, that developers are prone to making errors in their treatment of the boundaries of these classes. Thus test cases are generated to exercise these boundaries.

Example

Consider a component, *generate_grading,* with the following specification:

> *The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:*
>
> | | | |
> |---|---|---|
> | *greater than or equal to 70* | - | *'A'* |
> | *greater than or equal to 50, but less than 70* | - | *'B'* |
> | *greater than or equal to 30, but less than 50* | - | *'C'* |
> | *less than 30* | - | *'D'* |
>
> *Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.*

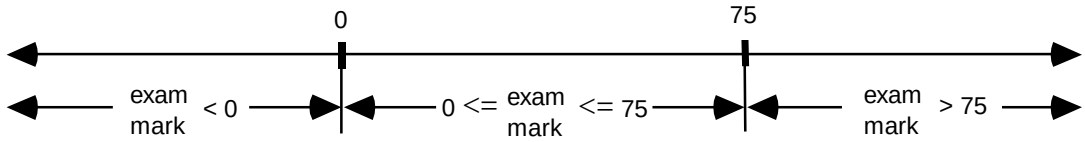
Initially the equivalence partitions are identified, then the boundaries of these partitions are identified, and then test cases are derived to exercise the boundaries. Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

0 <= exam mark <= 75
0 <= coursework mark <= 25

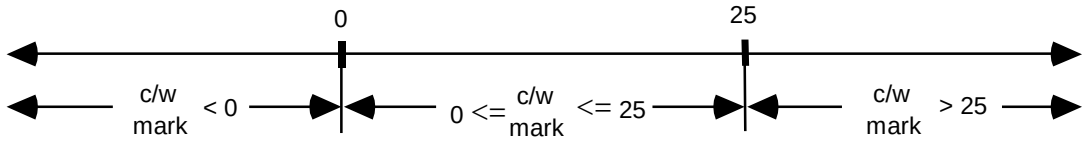
The most obvious *invalid* partitions can be described by:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0

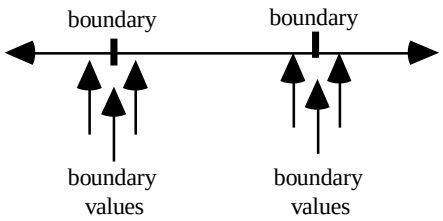
Partitioned ranges of values can be represented pictorially, therefore, for the input, exam mark, the same notation leads to:



And for the input, coursework mark, we get:



For each boundary three values are used, one on the boundary itself and one either side of it, the smallest significant distance away, as shown below:



Thus the six test cases derived from the input exam mark are:

Test Case	1	2	3	4	5	6
Input (exam mark)	-1	0	1	74	75	76
Input (c/w mark)	15	15	15	15	15	15
Boundary tested (exam mark)		0			75	
Exp. Output	'FM'	'D'	'D'	'A'	'A'	'FM'

Note that the input coursework (c/w) mark has been set to an arbitrary valid value of 15.

© British Computer Society, SIGIST, 2001

The test cases derived from the input coursework mark are thus:

Test Case	7	8	9	10	11	12
Input (exam mark)	40	40	40	40	40	40
Input (c/w mark)	-1	0	1	24	25	26
Boundary tested (c/w mark)		0			25	
Exp. Output	'FM'	'C'	'C'	'B'	'B'	'FM'

Note that the input exam mark has been set to an arbitrary valid value of 40.

Less obvious invalid input equivalence partitions would include any other inputs that can occur not so far included in a partition, for instance, non-integer inputs or perhaps non-numeric inputs. In order to be considered an equivalence partition those values within it must be expected, from the specification, to be treated in an equivalent manner by the component. Thus we could generate the following invalid input equivalence partitions:

exam mark = real number
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic
etc.

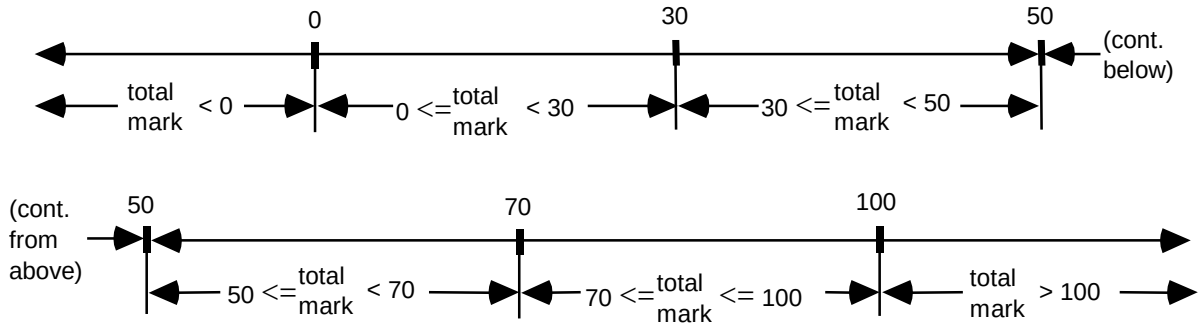
Although these are valid equivalence partitions they have no identifiable boundaries and so no test cases are derived.

Next, the partitions and boundaries for the outputs are identified. The *valid* partitions are produced by considering each of the valid outputs for the component thus:

'A'	is induced by	70 <= total mark <= 100
'B'	is induced by	50 <= total mark < 70
'C'	is induced by	30 <= total mark < 50
'D'	is induced by	0 <= total mark < 30
'Fault Message'	is induced by	total mark > 100
'Fault Message'	is induced by	total mark < 0

where total mark = exam mark + coursework mark.

'Fault Message' is considered here as it is a specified output. The equivalence partitions and boundaries for total mark are shown below:



Thus the test cases derived from the valid outputs are:

Test Case	13	14	15	16	17	18	19	20	21
Input (exam mark)	-1	0	0	29	15	6	24	50	26
Input (c/w mark)	0	0	1	0	15	25	25	0	25
total mark (as calculated)	-1	0	1	29	30	31	49	50	51
Boundary tested (total mark)		0			30			50	
Exp. Output	'FM'	'D'	'D'	'D'	'C'	'C'	'C'	'B'	'B'

Test Case	22	23	24	25	26	27
Input (exam mark)	49	45	71	74	75	75
Input (c/w mark)	20	25	0	25	25	26
total mark (as calculated)	69	70	71	99	100	101
Boundary tested (total mark)		70			100	
Exp. Output	'B'	'A'	'A'	'A'	'A'	'FM'

The input values of exam mark and coursework mark have been derived from the total mark, which is their sum.

An invalid output would be any output from the component other than one of the five specified. It is difficult to identify unspecified outputs, but obviously they must be considered as if we can cause one then we have identified a flaw with either the component, its specification, or both. For this example three unspecified outputs were identified ('E', 'A+', and 'null'), but it is not possible to group these possible outputs into ordered partitions from which boundaries can be identified and so no test cases are derived.

So far several partitions have been identified that appear to be bounded on one side only. These are:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0
total mark > 100
total mark < 0

In fact these partitions are bounded on their other side by implementation-dependent maximum and minimum values. For integers held in sixteen bits these would be 32767 and -32768 respectively.

Thus, the above partitions can be more fully described by:

75 < exam mark <= 32767
-32768 <= exam mark < 0
25 < coursework mark <= 32767
-32768 <= coursework mark < 0
100 < total mark <= 32767
-32768 <= total mark < 0

© British Computer Society, SIGIST, 2001

It can be seen that by bounding these partitions on both sides a number of additional boundaries are identified, which must be tested. This leads to the following additional test cases:

Test Case	28	29	30	31	32	33
Input (exam mark)	32766	32767	32768	-32769	-32768	-32767
Input (c/w mark)	15	15	15	15	15	15
Boundary tested (exam mark)		32767			-32768	
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Test Case	34	35	36	37	38	39
Input (exam mark)	40	40	40	40	40	40
Input (c/w mark)	32766	32767	32768	-32769	-32768	-32767
Boundary tested (c/w mark)		32767			-32768	
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Test Case	40	41	42	43	44	45
Input (exam mark)	16383	32767	1	0	-16384	-32768
Input (c/w mark)	16383	0	32767	-32767	-16384	-1
total mark (as calculated)	32766	32767	32768	-32767	-32768	-32769
Boundary tested (total mark)		32767			-32768	
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

It should be noted that where invalid input values are used (as above, in test cases 1, 6, 7, 12, 13, and 27-45) it may, depending on the implementation, be impossible to actually execute the test case. For instance, in Ada, if the input variable is declared as a positive integer then it will not be possible to assign a negative value to it. Despite this, it is still worthwhile *considering* all the test cases for completeness.

The above test case suite achieves 100% boundary value coverage as it ensures that all identified boundaries are exercised by at least one test case. Lower levels of coverage would be achieved if all the boundaries identified are not all exercised. If all the boundaries are not identified, then any coverage measure based on this incomplete set of boundaries would be misleading.

B.3 State Transition Testing

Introduction

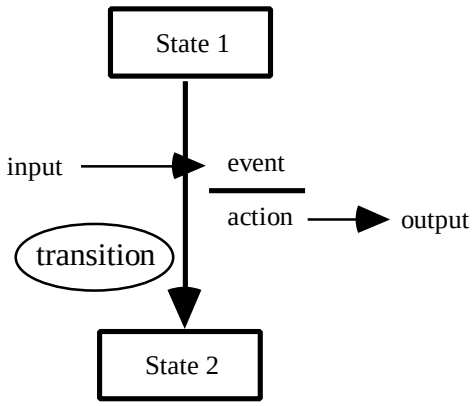
This black box technique is based upon an analysis of the specification of the component to model its behaviour by state transitions. We illustrate the technique by means of a worked example. Naturally, the technique is only effective to the extent that the model captures the specification of the component.

Example

Consider a component, *manage_display_changes*, with the following specification:

> *The component responds to input requests to change an externally held display mode for a time display device. The external display mode can be set to one of four values: Two correspond to displaying either the time or the date, and the other two correspond to modes used when altering either the time or date.*
>
> *There are four possible input requests: 'Change Mode', 'Reset', 'Time Set' and 'Date Set'. A 'Change Mode' input request shall cause the display mode to move between the 'display time' and 'display date' values. If the display mode is set to 'display time' or 'display date' then a 'Reset' input request shall cause the display mode to be set to the corresponding 'alter time' or 'alter date' modes. The 'Time Set' input request shall cause the display mode to return to 'display time' from 'alter time' while similarly the 'Date Set' input request shall cause the display mode to return to 'display date' from 'alter date'.*

A state model is produced for the component to identify its states, transitions, and their events and actions. State transition diagrams (STD) are commonly used as state models and their notation is briefly illustrated opposite.



Events are always caused by input. Similarly, actions are likely to cause output. The output from an action may be essential in order to identify the current state of the component. A transition is determined by the current state and an event and is normally labelled simply with the event and action.

The STD for the component *manage_display_changes* is shown in figure B.3.

© British Computer Society, SIGIST, 2001

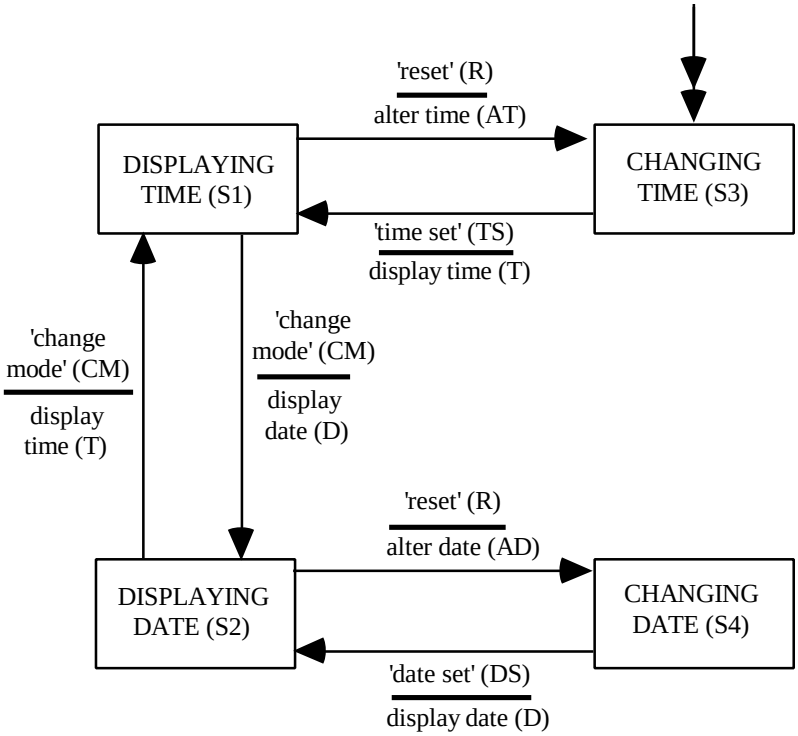


Figure B.3 STD for Manage_Display_Changes

Test cases are initially derived from the state transition diagram to exercise each of the possible transitions (using the abbreviated STD labels):

Test Case	1	2	3	4	5	6
Start State	S1	S1	S3	S2	S2	S4
Input	CM	R	TS	CM	R	DS
Expected Output	D	AT	T	T	AD	D
Finish State	S2	S3	S1	S1	S4	S2

This indicates that for test case 1 the starting state is DISPLAYING TIME (S1), the input is 'change mode' (CM), the expected output is 'display date' (D), and the finish state is DISPLAYING DATE (S2).

This set of six test cases exercises each of the possible transitions and so achieves 0-switch coverage [Chow]. Tests written to achieve this level of coverage are limited in their ability to detect some types of faults because although they will detect the most obvious incorrect transitions and outputs, they will not detect more subtle faults that are only detectable through exercising sequences of transitions.

Tests written to achieve the next level of coverage, 1-switch, exercise all the possible sequential pairs of transitions, of which there are ten in the *manage_display_changes* component:

Test Case	1	2	3	4	5	6	7	8	9	10
Start State	S1	S1	S1	S3	S3	S2	S2	S2	S4	S4
Input	CM	CM	R	TS	TS	CM	CM	R	DS	DS
Exp. Output	D	D	AT	T	T	T	T	AD	D	D
Next State	S2	S2	S3	S1	S1	S1	S1	S4	S2	S2
Input	CM	R	TS	CM	R	CM	R	DS	CM	R
Exp. Output	T	AD	T	D	AT	D	AT	D	T	AD
Finish State	S1	S4	S1	S2	S3	S2	S3	S2	S1	S4

This indicates that test case 1 comprises two transitions. For the first transition the starting state is DISPLAYING TIME (S1), the initial input is 'change mode' (CM), the intermediate expected output is display date (D), and the next state is DISPLAYING DATE (S2). For the second transition, the second input is 'change mode' (CM), the final expected output is display time (T), and the finish state is DISPLAYING TIME (S1).

Note that intermediate states, and the inputs and outputs for each transition, are explicitly defined.

Longer sequences of transitions can be tested to achieve higher and higher levels of switch coverage, dependent on the level of test thoroughness required.

A limitation of the test cases derived to achieve switch coverage is that they are designed to exercise only the valid transitions in the component. A more thorough test of the component will *also* attempt to cause invalid transitions to occur. The STD only explicitly shows the valid transitions (all transitions not shown are considered invalid). A state model that explicitly shows both valid and invalid transitions is the state table. The notation used for state tables is briefly described below:

	Input 1	Input 2	etc.
Start State 1	Entry A	Entry B	etc.
Start State 2	Entry C	Entry D	etc.
etc.	etc.	etc.	etc.

where Entry X = Finish State / Output for the given start state and input.

The state table for the *manage_display_changes* component is shown below:

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/N	S1/N
S2	S1/T	S4/AD	S2/N	S2/N
S3	S3/N	S3/N	S1/T	S3/N
S4	S4/N	S4/N	S4/N	S2/D

Any entry where the state remains the same and the output is shown as null (N) represents a null transition, where any *actual* transition that can be induced will represent a failure. It is the testing of these null transitions that is ignored by test sets designed just to achieve switch coverage. Thus a more complete test set will test both possible transitions and null transitions, which means testing the response of the component to all possible inputs in all possible states. The state table provides an ideal means of directly deriving this set of test cases.

© British Computer Society, SIGIST, 2001

There are 16 entries in the table above representing each of the four *possible* inputs that can occur in each of the four *possible* states, making 16 test cases which can be read from the state table as shown below:

	CM	R	TS	DS
S1	S2/D (Test Case 1)	S3/AT (Test Case 2)	S1/N (Test Case 3)	S1/N (Test Case 4)
S2	S1/T (Test Case 5)	S4/AD (Test Case 6)	S2/N (Test Case 7)	S2/N (Test Case 8)
S3	S3/N (Test Case 9)	S3/N (Test Case 10)	S1/T (Test Case 11)	S3/N (Test Case 12)
S4	S4/N (Test Case 13)	S4/N (Test Case 14)	S4/N (Test Case 15)	S2/D (Test Case 16)

which corresponds to:

Test Case	1	2	3	4	5	....	. .	....	12	13	14	15	16
Start State	S1	S1	S1	S1	S2	....	. .	....	S3	S4	S4	S4	S4
Input	CM	R	TS	DS	CM	....	. .	....	DS	CM	R	TS	DS
Exp. Output	D	AT	N	N	T	....	. .	....	N	N	N	N	D
Finish State	S2	S3	S1	S1	S1	....	. .	....	S3	S4	S4	S4	S2

If the above test cases are compared with those produced to achieve 0-switch coverage then it can be seen that by also testing the null transitions an extra 10 test cases are created (3,4,7,8,9,10,12,13,14 and 15).

B.4 Cause Effect Graphing

Introduction

This black box technique is based upon an analysis of the specification of the component to model its behaviour by means of causes and effects. We illustrate the technique by means of a worked example. Naturally, the technique is only effective to the extent that the model captures the specification of the component.

Example

Take a cheque debit function whose inputs are *debit amount*, *account type* and *current balance* and whose outputs are *new balance* and *action code*. *Account type* may be postal ('p') or counter ('c'). The *action code* may be 'D&L', 'D', 'S&L' or 'L', corresponding to 'process debit and send out letter', 'process debit only', 'suspend account and send out letter' and 'send out letter only' respectively. The function has the following specification:

> *If there are sufficient funds available in the account or the new balance would be within the authorised overdraft limit then the debit is processed. If the new balance would exceed the authorised overdraft limit then the debit is not processed and if it is a postal account it is suspended. Letters are sent out for all transactions on postal accounts and for non-postal accounts if there are insufficient funds available (i.e. the account would no longer be in credit).*

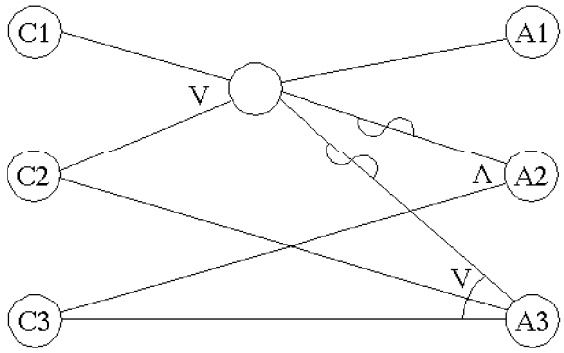
The conditions are:

C1	New balance in credit
C2	New balance overdraft, but within authorised limit
C3	Account is postal

The actions are:

A1	Process debit
A2	Suspend account
A3	Send out letter

A cause-effect graph shows the relationship between the conditions and actions in a notation similar to that used by designers of hardware logic circuits. The specification is modelled by the graph shown in Figure B.4.



This figure uses standard logical symbols for AND and OR (Λ and V respectively) and the wavy line (◡◠) indicates NOT. The arc groups those inputs affected by a logical symbol, where there are more than two inputs involved.

Figure B.4: Cause-effect graph

The code graph is then recast in terms of a decision table. Each column of the decision table is a rule. The table comprises two parts. In the first part each rule is tabulated against the conditions. A 'T' indicates that the condition must be TRUE for the rule to apply and an 'F' indicates that the condition must be FALSE for the rule to apply. In the second part, each rule is tabulated against the actions. A 'T' indicates that the action will be performed; an 'F' indicates that the action will not be performed; an asterisk (*) indicates that the combination of conditions is infeasible and so no actions are defined for the rule.

The example has the following decision table:

Rules:	1	2	3	4	5	6	7	8
C1: New balance in credit	F	F	F	F	T	T	T	T
C2: New balance overdraft, but within authorised limit	F	F	T	T	F	F	T	T
C3: Account is postal	F	T	F	T	F	T	F	T
A1: Process debit	F	F	T	T	T	T	*	*
A2: Suspend account	F	T	F	F	F	F	*	*
A3: Send out letter	T	T	T	T	F	T	*	*

© British Computer Society, SIGIST, 2001

The following test cases would be required to provide 100% cause-effect coverage, and correspond to the rules in the decision table above (no test cases are generated for rules 7 and 8 as they are infeasible):

	CAUSES				EFFECTS	
test case	account type	overdraft limit	current balance	debit amount	new balance	action code
1	'c'	£100	-£70	£50	-£70	'L'
2	'p'	£1500	£420	£2000	£420	'S&L'
3	'c'	£250	£650	£800	-£150	'D&L'
4	'p'	£750	-£500	£200	-£700	'D&L'
5	'c'	£1000	£2100	£1200	£900	'D'
6	'p'	£500	£250	£150	£100	'D&L'

B.5 Syntax Testing

Introduction

This black box technique is based upon an analysis of the specification of the component to model its behaviour by means of a description of the input via its syntax. We illustrate the technique by means of a worked example. The technique is only effective to the extent that the syntax as defined corresponds to the required syntax.

Example

Consider a component that simply checks whether an input *float_in* conforms to the syntax of a floating point number, `float` (defined below). The component outputs *check_res*, which takes the form 'valid' or 'invalid' dependent on the result of its check.

Here is a representation of the syntax for the floating point number, `float` in Backus Naur Form (BNF) :

```
float  =  int "e" int.
int    =  ["+"|"-"] nat.
nat    =  {dig}.
dig    =  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

Terminals are shown in quotation marks; these are the most elementary parts of the syntax - the actual characters that make up the input to the component. `|` separates alternatives. `[]` surrounds an optional item, that is, one for which nothing is an alternative. `{}` surrounds an item which may be iterated one or more times.

B.5.1 Test Cases with Valid Syntax

The first step is to derive the options from the syntax. Each option is followed by a label, in the form [opt_1], [opt_2], etc., to enable it to be identified later.

`float` has no options.
`int` has three options: `nat` [opt_1], `"+" nat` [opt_2] and `"-" nat` [opt_3].
`nat` has two options: a single digit number [opt_4] and a multiple digit number [opt_5].
`dig` has ten options: one for each digit [opt_6 to opt_15].

There are thus fifteen options to be covered. The next step is to construct test cases to cover the options.

The following test cases cover them all:

test case	float_in	option(s) executed	check_res
1	3e2	opt_1	'valid'
2	+2e+5	opt_2	'valid'
3	-6e-7	opt_3	'valid'
4	6e-2	opt_4	'valid'
5	1234567890e3	opt_5	'valid'
6	0e0	opt_6	'valid'
7	1e1	opt_7	'valid'
8	2e2	opt_8	'valid'
9	3e3	opt_9	'valid'
10	4e4	opt_10	'valid'
11	5e5	opt_11	'valid'
12	6e6	opt_12	'valid'
13	7e7	opt_13	'valid'
14	8e8	opt_14	'valid'
15	9e9	opt_15	'valid'

This is by no means a minimal test set to exercise the 15 options (it can be reduced to just three test cases, for example, 2, 3 and 5 above), and some test cases will exercise more options than the single one listed in the 'options executed' column. Each option has been treated separately here to aid understanding of their derivation. This approach may also contribute to the ease with which the causes of faults are located.

B.5.2 Test Cases with Invalid Syntax

The first step is to construct a checklist of generic mutations. A possible checklist is:

m1. introduce an invalid value for an element;

m2. substitute an element with another defined element;

m3. miss out a defined element;

m4. add an extra element.

© British Computer Society, SIGIST, 2001

These generic mutations are applied to the individual elements of the syntax to yield specific mutations. The elements, represented in the form el_1, el_2, etc., of the syntax for `float` can be identified from the BNF representation as shown below:

```
float  =  int "e" int.                          el_1   =  el_2 el_3 el_4.
int    =  ["+"|"-"] nat.                        el_5   =  el_6 el_7.
nat    =  {dig}.                                el_8   =  el_9.
dig    =  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".   el_10  =  el_11.
```

[“+”|”-”] has been treated as a single element because the mutation of optional items separately does not create test cases with invalid syntax (using these generic mutations).

The next step is to construct test cases to cover the mutations:

test case	float_in	mutation	element	check_res
1	xe0	m1	x for el_2	'invalid'
2	0x0	m1	x for el_3	'invalid'
3	0ex	m1	x for el_4	'invalid'
4	x0e0	m1	x for el_6	'invalid'
5	+xe0	m1	x for el_7	'invalid'
6	ee0	m2	el_3 for el_2	'invalid'
7	+e0	m2	el_6 for el_2	'invalid'
8	000	m2	el_2 for el_3	'invalid'
9	0+0	m2	el_6 for el_3	'invalid'
10	0ee	m2	el_3 for el_4	'invalid'
11	0e+	m2	el_6 for el_4	'invalid'
12	e0e0	m2	el_3 for el_6	'invalid'
13	+ee0	m2	el_3 for el_7	'invalid'
14	++e0	m2	el_6 for el_7	'invalid'
15	e0	m3	el_2	'invalid'
16	00	m3	el_3	'invalid'
17	0e	m3	el_4	'invalid'
18	y0e0	m4	y in el_1	'invalid'
19	0ye0	m4	y in el_1	'invalid'
20	0ey0	m4	y in el_1	'invalid'
21	0e0y	m4	y in el_1	'invalid'
22	y+0e0	m4	y in el_5	'invalid'
23	+y0e0	m4	y in el_5	'invalid'
24	+0yeo	m4	y in el_5	'invalid'

Some of the mutations are indistinguishable from correctly formed expansions and these have been discarded. For example, the generic mutation m2 (el_2 for el_4) generates correct syntax as m2 is “substitute an element with another defined element” and el_2 and el_4 are the same (int).

Some of the remaining mutations are indistinguishable from each other and these are covered by a single test case. For example, applying the generic mutation m1 (“introduce an invalid value for an element”) by replacing el_4, which should be an integer, with “+” creates the form “0e+”. This is the same input as generated for test case 11 above.

Many more test cases can be created by making different choices when using single mutations, or combining mutations.

B.5.3 Syntax Test Coverage

No test coverage measures are defined for syntax testing. Any measures of coverage would be based on the rules for generating valid syntax options and the checklist for generating test cases with invalid syntax. Neither the rules nor the checklist are definitive and so any syntax test coverage measures based on a particular set will be specific to that set of rules and checklist.

B.6 Statement Testing and Coverage

Introduction

This structural test technique is based upon the decomposition of the component into constituent statements.

Example

The two principal questions to answer are:

- what is a statement?
- which statements are executable?

In general a statement should be an atomic action, that is a statement should be executed completely or not at all. For instance:

IF *a* THEN *b* ENDIF

is considered as more than one statement because *b* may or may not be executed depending upon the condition *a*. The definition of *statement* used for statement testing need not be the one used in the language definition.

We would expect statements which are associated with machine code to be regarded as executable. For instance, we would expect all of the following to be regarded as executable:

- assignments;
- loops and selections;
- procedure and function calls;
- variable declarations with explicit initialisations;
- dynamic allocation of variable storage on a heap.

However, most other variable declarations can be regarded as non executable.

Consider, the following C code:

```
a;
if (b) {
      c;
      }
d;
```

Any test case with b TRUE will achieve full statement coverage. Note that full statement coverage can be achieved without exercising with b FALSE.

© British Computer Society, SIGIST, 2001

B.7 Branch/Decision Testing

Branch and Decision Coverage are closely related. For components with one entry point 100% Branch Coverage is equivalent to 100% Decision Coverage, although lower levels of coverage may not be the same. Both levels of coverage will be illustrated with one example:

> *The component shall determine the position of a word in a table of words ordered alphabetically. Apart from the word and table, the component shall also be passed the number of words in the table to be searched. The component shall return the position of the word in the table (starting at zero) if it is found, otherwise it shall return "-1".*

The corresponding code is drawn from [K&R]. The three decisions have been highlighted.

```
int binsearch (char *word, struct key tab[], int n) {
      int cond;
      int low, high, mid;
      low = 0;
      high = n - 1;
      while (low <= high) {
            mid = (low+high) / 2;
            if ((cond = strcmp(word, tab[mid].word)) < 0)
                  high = mid - 1;
            else if (cond > 0)
                  low = mid + 1;
            else
                  return mid;
      }
      return -1;
}
```

In this example each decision has two outcomes corresponding to the true and false values of the conditions. It is generally possible for a decision to have more than two outcomes.

Branch coverage may be demonstrated through coverage of the control flow graph of the program. The first step to constructing a control flow graph for a procedure is to divide it into basic blocks. These are sequences of instructions with no branches into the block (except to the beginning) and no branches out of the block (except at the end). The statements in a basic block are guaranteed to be executed together or not at all. The program above has the following basic blocks.

```
      int binsearch (char *word, struct key tab[], int n) {
            int cond;
            int low, high, mid;

B1          low = 0;
            high = n - 1;
B2          while (low <= high) {
B3                mid = (low+high) / 2
                  if ((cond = strcmp(word, tab[mid].word)) < 0)
B4                      high = mid - 1;
B5                else if (cond > 0)
B6                      low = mid + 1;
B7                else
                        return mid;
B8          }
B9          return -1;
      }
```

A control flow graph may be constructed by making each basic block a node and drawing an arc for each possible transfer of control from one basic block to another. These are the possible transfers of control:

B1 ->⁇ 2	B3 -> B4	B5 -> B6	B6 -> B8
B2 -> B3	B3 -> B5	B5 -> B7	B8 -> B2
B2 -> B9	B4 -> B8		

This results in the graph presented in figure B.7. The graph has one entry point, B1, and two exit points, B7 and B9.

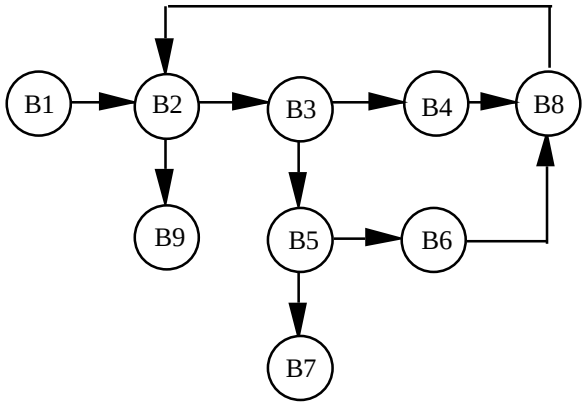


Figure B.7: Control flow graph for `binsearch`

Of course, the above control flow graph would not necessarily be constructed by hand, but a tool would normally be used to show which decisions/branches have been executed.

The decisions are given by the basic blocks having more than one exit arrow, namely B2, B3 and B5. Since each of these three blocks have two exits, we have six decisions to consider.

The branches are given by the arrows in the control flow graph; 10 in total.

For both branches and decisions any individual test of the component will exercise a path and hence potentially many decisions and branches.

Consider a test case which executes the path B1 -> B2 -> B9. This case arises when n=0, that is, when the table being searched has no entries. This path executes one decision (B2 -> B9) and hence provides 1/6 = 16.7% coverage. The path executes 2 out of the 10 branches, giving 20% coverage (which is not the same as the coverage for decisions).

Consider now a test case which executes the path:

B1->B2->B3->B4->B8->B2->B3->B5->B6->B8->B2->B3->B5->B7

This path arises when the search first observes that the entry is in the first half of the table, then the second half of that (i.e., 2nd quarter) and then finds the entry. Note that the two test cases provide 100% decision and branch coverage.

© British Computer Society, SIGIST, 2001

These test cases are shown below:

test case	inputs			decisions exercised (underlined)	expected outcome
	word	tab	n		
1	chas	'empty table'	0	B1 » B2 » B9	-1
2	chas	alf bert chas dick eddy fred geoff	7	B1 » B2 » B3 » B4 » B8 » B2 » B3 » B5 » B6 » B8 » B2 » B3 » B5 » B7	2

Branch and decision coverage are both normally measured using a software tool.

B.8 Data Flow Testing

Introduction

Data flow testing is a structural test technique which aims to execute subpaths from points where each variable in a component is defined to points where it is referenced. These subpaths are known as definition-use pairs (du-pairs). The different data flow coverage criteria require different du-pairs and subpaths to be executed. Test sets are generated here to achieve 100% coverage (where possible) for each of those criteria.

Example

Consider the data flow testing of the following component in Ada:

```
procedure Solve_Quadratic(A, B, C: in Float; Is_Complex: out
                                Boolean; R1, R2: out Float) is

-- Is_Complex is true if the roots are not real.
-- If the two roots are real, they are produced in R1, R2.

       Discrim : Float := B*B - 4.0*A*C;                    -- 1
       R1, R2: Float;                                       -- 2
   begin                                                    -- 3
       if Discrim < 0.0 then                                -- 4
          Is_Complex := true;                               -- 5
       else                                                 -- 6
          Is_Complex := false;                              -- 7
       end if;                                              -- 8
       if not Is_Complex then                               -- 9
          R1 := (-B + Sqrt(Discrim))/ (2.0*A);              -- 10
          R2 := (-B - Sqrt(Discrim))/ (2.0*A);              -- 11
       end if;                                              -- 12
   end Solve_Quadratic;                                     -- 13
```

Note that the second line is not a definition (of R1 and R2) but a declaration. (For languages with default initialisation, it would be a definition.)

The first step is to list the variables used in the component. These are: A, B, C, Discrim, Is_Complex, R1 and R2.

Next, each occurrence of a variable in the component is listed and assigned a category (definition, predicate-use or computation-use):

line	category		
	definition	c-use	p-use
0	A,B,C		
1	Discrim	A,B,C	
2			
3			
4			Discrim
5	Is_Complex		
6			
7	Is_Complex		
8			
9			Is_Complex
10	R1	A,B,Discrim	
11	R2	A,B,Discrim	
12			
13			

The next step is to identify the du-pairs and their type (c-use or p-use), by identifying links from each entry in the definition column to each entry for that variable in the c-use or p-use column. Note that we cannot form any definition-use pair for R1 and R2 since they only have a definition within the component.

definition-use pair (start line -> end line)	variable(s)	
	c-use	p-use
0 ---> 1	A,B,C	
0 ---> 10	A,B	
0 ---> 11	A,B	
1 ---> 4		Discrim
1 ---> 10	Discrim	
1 ---> 11	Discrim	
5 ---> 9		Is_Complex
7 ---> 9		Is_Complex

© British Computer Society, SIGIST, 2001

B.8.1 All-definitions

To achieve 100% All-definitions data flow coverage at least one subpath from each variable definition to some use of that definition (either p-use and c-use) must be executed. The following test set would satisfy this requirement:

	All Definitions			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
3	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
4	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

Note that several of the test cases satisfy the requirement for more than one variable, this will apply to each of the subsequent test sets as well. It can also be seen that the same test inputs satisfy the subpath execution criteria for several of the du-pairs (again this will apply to the subsequent test sets as well).

B.8.2 All-c-uses

To achieve 100% 'All-c-uses data flow coverage at least one subpath from each variable definition to every c-use of that definition must be executed. The following test set would satisfy this requirement:

	All-c-uses			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	A,B	0 --> 10	0-1-4-7-9-10	1	2	1	F	-1	-1
3	A,B	0 --> 11	0-1-4-7-9-10-11	1	2	1	F	-1	-1
4	Discrim	1 --> 10	1-4-7-9-10	1	2	1	F	-1	-1
5	Discrim	1 --> 11	1-4-7-9-10-11	1	2	1	F	-1	-1

B.8.3 All-p-uses

To achieve 100% All-p-uses data flow coverage at least one subpath from each variable definition to every p-use of that definition must be executed. The following test set would satisfy this requirement:

	All-p-uses			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
2	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
3	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

B.8.4 All-uses

To achieve 100% All-uses data flow coverage at least one subpath from each variable definition to every use of that definition (both p-use and c-use) must be executed. The following test set would satisfy this requirement:

	All-uses / All du-paths			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	d-u pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	A,B	0 --> 10	0-1-4-7-9-10	1	2	1	F	-1	-1
3	A,B	0 --> 11	0-1-4-7-9-10-11	1	2	1	F	-1	-1
4	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
5	Discrim	1 --> 10	1-4-7-9-10	1	2	1	F	-1	-1
6	Discrim	1 --> 11	1-4-7-9-10-11	1	2	1	F	-1	-1
7	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
8	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

B.8.5 All-du-paths

To achieve 100% All-du-paths data flow coverage every 'simple subpath' from each variable definition to every use of that definition must be executed. This differs from All-uses in that *every* simple subpath between the du-pairs must be executed. There are just two subpaths through the component that are not already identified in the test cases for All-Uses. These are 0-1-4-5-9-10 and 1-4-5-9-10. Both of these subpaths are infeasible and so no test cases can be generated to exercise them.

This form of testing needs to define the data objects considered. Tools will typically regard an array or record as a single data item rather than as a composite item with many constituents. Ignoring the constituents of composite objects reduces the effectiveness of data flow testing.

B.9 / B.10 / B.11 Condition Testing

Introduction

Branch Condition Testing, Branch Condition Combination Testing, and Modified Condition Decision Testing are closely related, as are the associated coverage measures. For convenience, these test case design and test measurement techniques are collectively referred to as condition testing.

Condition testing is based upon an analysis of the conditional control flow within the component and is therefore a form of structural testing.

Example

Consider the following fragment of code:

```
if A or (B and C) then
      do_something;
else
      do_something_else;
end if;
```


© British Computer Society, SIGIST, 2001

The Boolean operands within the decision condition are A, B and C. These may themselves be comprised of complex expressions involving relational operators. For example, the Boolean operand A could be an expression such as X>=Y. However, for the sake of clarity, the following examples regard A, B and C as simple Boolean operands.

Branch Condition Testing and Coverage

Branch Condition Coverage would require Boolean operand A to be evaluated both TRUE and FALSE, Boolean operand B to be evaluated both TRUE and FALSE, and Boolean operand C to be evaluated both TRUE and FALSE.

Branch Condition Coverage may therefore be achieved with the following set of test inputs (note that there are alternative sets of test inputs which will also achieve Branch Condition Coverage):

Case	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	TRUE	TRUE

Branch Condition Coverage can often be achieved with just two test cases, irrespective of the number of actual Boolean operands comprising the condition.

A further weakness of Branch Condition Coverage is that it can often be achieved without testing both the TRUE and FALSE branches of the decision. For example, the following alternative set of test inputs achieve Branch Condition Coverage, but only test the TRUE outcome of the overall Boolean condition. Thus Branch Condition Coverage will not necessarily subsume Branch Coverage.

Case	A	B	C
1	TRUE	FALSE	FALSE
2	FALSE	TRUE	TRUE

Branch Condition Combination Testing and Coverage

Branch Condition Combination Coverage would require all combinations of Boolean operands A, B and C to be evaluated. For the example condition, Branch Condition Combination Coverage can only be achieved with the following set of test inputs:

Case	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	FALSE	FALSE
3	FALSE	TRUE	FALSE
4	FALSE	FALSE	TRUE
5	TRUE	TRUE	FALSE
6	FALSE	TRUE	TRUE
7	TRUE	FALSE	TRUE
8	TRUE	TRUE	TRUE

Branch Condition Combination Coverage is very thorough, requiring 2^n test cases to achieve 100% coverage of a condition containing n Boolean operands. This rapidly becomes unachievable for more complex conditions.

Modified Condition Decision Testing and Coverage

Modified Condition Decision Coverage (MCDC) is a pragmatic compromise which requires fewer test cases than Branch Condition Combination Coverage. It is widely used in the development of avionics software, as required by RTCA/DO-178B.

Modified Condition Decision Coverage requires test cases to show that each Boolean operand (A, B and C) can independently affect the outcome of the decision. This is less than all the combinations (as required by Branch Condition Combination Coverage).

For the example decision condition [A or (B and C)], we first require a pair of test cases where changing the state of A will change the outcome, but B and C remain constant, i.e. that A can independently affect the outcome of the condition:

Case	A	B	C	Outcome
A1	FALSE	FALSE	TRUE	FALSE
A2	TRUE	FALSE	TRUE	TRUE

Similarly for B, we require a pair of test cases which show that B can independently affect the outcome, with A and C remaining constant:

Case	A	B	C	Outcome
B1	FALSE	FALSE	TRUE	FALSE
B2	FALSE	TRUE	TRUE	TRUE

Finally for C we require a pair of test cases which show that C can independently affect the outcome, with A and B remaining constant:

Case	A	B	C	Outcome
C1	FALSE	TRUE	TRUE	TRUE
C2	FALSE	TRUE	FALSE	FALSE

Having created these pairs of test cases for each operand separately, it can be seen that test cases A1 and B1 are the same, and that test cases B2 and C1 are the same. The overall set of test cases to provide 100% MCDC of the example expression is consequently:

Case	A	B	C	Outcome
1 (A1,B1)	FALSE	FALSE	TRUE	FALSE
2 (A2)	TRUE	FALSE	TRUE	TRUE
3 (B2,C1)	FALSE	TRUE	TRUE	TRUE
4 (C2)	FALSE	TRUE	FALSE	FALSE

In summary:

A is shown to independently affect the outcome of the decision condition by test cases 1 and 2;
B is shown to independently affect the outcome of the decision condition by test cases 1 and 3;
C is shown to independently affect the outcome of the decision condition by test cases 3 and 4.

© British Computer Society, SIGIST, 2001

Note that there may be alternative solutions to achieving MCDC. For example, A could have been shown to independently affect the outcome of the condition by the following pair of test cases:

Case	A	B	C	Outcome
A3	FALSE	TRUE	FALSE	FALSE
A4	TRUE	TRUE	FALSE	TRUE

Test case A3 is the same as test case C2 (or 4) above, but test case A4 is one which has not been previously used. However, as MCDC has already been achieved, test case A4 is not required for coverage purposes.

To achieve 100% Modified Condition Decision Coverage requires a minimum of n+1 test cases, and a maximum of 2n test cases, where n is the number of Boolean operands within the decision condition. In contrast, Branch Condition Combination Coverage requires n^2 test cases. MCDC is therefore a practical compromise with Branch Condition Combination Coverage where condition expressions involve more than just a few Boolean operands.

Other Boolean Expressions

One weakness of these condition testing and test measurement techniques is that they are vulnerable to the placement of Boolean expressions which control decisions being placed outside of the actual decision condition. For example:

```
FLAG := A or (B and C);
if FLAG then
      do_something;
else
      do_something_else;
end if;
```

To combat this vulnerability, a practical variation of these condition testing and condition coverage techniques is to design tests for all Boolean expressions, not just those used directly in control flow decisions.

Optimised Expressions

Some programming languages and compilers short circuit the evaluation of Boolean operators.

For example, the C and C++ languages always short circuit the Boolean "and" (&&) and "or" (||) operators, and the Ada language provides special short circuit operators **and then** and **or else**. With these examples, when the outcome of a Boolean operator can be determined from the first operand, then the second operand will not be evaluated.

The consequence is that it will be infeasible to show coverage of one value of the second operand. For a short circuited "and" operator, the feasible combinations are True:True, True:False and False:X, where X is unknown. For a short circuited "or" operator, the feasible combinations are False:False, False:True and True:X.

Other languages and compilers may short circuit the evaluation of Boolean operators in any order. In this case, the feasible combinations are not known. The degree of short circuit optimisation of Boolean operators may depend upon compiler switches or may be outside the user's control.

Short circuited control forms present no obstacle to Branch Condition Coverage or Modified Condition Decision Coverage, but they do obstruct the measurement of Branch Condition Combination Coverage. There are situations where it is possible to design test cases which should achieve 100% coverage (from a theoretical point of view), but where it is not possible to actually measure that 100% coverage has been achieved.

Other Branches and Decisions

The above description of condition testing techniques and condition coverage measures is given in terms of branches or decisions which are controlled by Boolean conditions. Other branches and decisions, such as multi-way branches (implemented by "case", "switch" or "computed goto" statements), and counting loops (implemented by "for" or "do" loops) do not use Boolean conditions, and are therefore not addressed by the descriptions.

There are two options available.

The first option is to assume that the branch or decision is actually implemented as an equivalent set of Boolean conditions (irrespective of how the design is coded), to design test cases which would exercise these conditions using one of the condition testing techniques, and to measure coverage as if the equivalent Boolean conditions were actually coded.

The second option is to only use a condition testing test case design technique and a coverage measure as a supplement to branch testing and branch or decision coverage. Branch testing will address all simple decisions, multi-way decisions, and all loops. Condition testing will then address the decisions which include Boolean conditions.

In practice, a set of test cases which achieves 100% coverage by one of these options will also achieve 100% coverage by the other option. However, lower levels of coverage cannot be compared between the two options.

B.12 LCSAJ Testing

Introduction

An LCSAJ (which stands for Linear Code Sequence And Jump) is defined as a linear sequence of executable code commencing either from the start of a program or from a point to which control flow may jump and terminated either by a specific control flow jump or by the end of the program. It may contain predicates which must be satisfied in order to execute the linear code sequence and terminating jump.

Example

Some reformatting may be needed to allow LCSAJs to be expressed in terms of line numbers. The basic reformatting rule is that each branch must leave from the end of a line and arrive at the start of a line.

© British Computer Society, SIGIST, 2001

Consider the following program which is designed to categorise positive integers into prime and non-prime, and to give factors for those which are non-prime. Note that integer division is used, so Num DIV 2 will give a value of 2 when Num = 5, for example. The code on line 5 calculates the remainder left after Factor is divided into Num. For example, if Factor is 5 and Num is 13, the expression evaluates to 3. For reasons of simplicity, the program makes no checks on the input.

```
1     READ (Num);
2     WHILE NOT End of File DO
3           Prime := TRUE;
4           FOR Factor := 2 TO Num DIV 2 DO
5                 IF Num - (Num DIV Factor)*Factor = 0 THEN
6                       WRITE (Factor, ` is a factor of', Num);
7                       Prime := FALSE;
8                 ENDIF;
9           ENDFOR;
10          IF Prime = TRUE THEN
11                WRITE (Num, ` is prime');
12          ENDIF;
13          READ (Num);
14    ENDWHILE;
15    WRITE (`End of prime number program');
```

Deriving LCSAJs

To progress methodically, firstly list the table of branches as shown below. These are listed with a note of the necessary conditions to satisfy them.

(Note that our convention is to consider the transfer of control as being to the line after the line indicating the end of a control structure. An alternative is to regard control as being transferred to the end line; this would give different line numbers in the LCSAJs, but they would be essentially the same. The branching structure of loops is language dependent and may be different from that used here.)

(2->3)	:	requires NOT End of File
(2->15)	: Jump	requires End of File
(4->5)	:	requires the loop to execute, i.e. Num DIV 2 is greater than or equal to 2
(4->10)	: Jump	requires the loop to be a zero-trip, i.e. Num DIV 2 is less than 2
(5->6)	:	requires the IF statement on line 5 to be true
(5->9)	: Jump	requires the IF statement on line 5 to be false
(9->5)	: Jump	requires a further iteration of the FOR loop to take place
(9->10)	:	requires the FOR loop to have exhausted
(10->11)	:	requires Prime to be true
(10->13)	: Jump	requires Prime to be false
(14->2)	: Jump	must always take place

From this list, find the LCSAJ start points. These are the start of the component (line 1) and lines to which control flow can jump from other than the preceding line (lines 2, 5, 9, 10, 13 and 15). For example, if the IF statement on line 10 is false, control will jump from line 10 to line 13.

The first LCSAJ starts at line 1. The first possible jump following line 1 is (2->15), so our first LCSAJ is (1, 2, 15).

There are other LCSAJs starting from line 1. If we had taken the (2->3) branch, we would have continued executing a linear code sequence with line 3. The next possible jump is (4->10), so our second LCSAJ is (1, 4, 10).

Again, we could have taken the (4->5) branch at this point. At line 5, there is a choice of a jump to line 9 or a branch to the next line. At line 9, there is the choice of a jump back to line 5 or a branch to the next line. At line 10, there is the choice of a jump to line 13 or a branch to the next line. The linear code sequence can extend as far as line 14, where control flow must return to line 2 to test the WHILE condition.

There are 6 LCSAJs starting from line 1. They are (1, 2, 15), (1, 4, 10), (1, 5, 9), (1, 9, 5), (1, 10, 13), and (1, 14, 2). To execute LCSAJ (1, 14, 2) requires a total of 5 branches to execute in sequence, namely (2->3), (4->5), (5->6), (9->10), (10->11) followed by the jump (14->2).

Continue the search for each LCSAJ start line. Since the outer loop will always transfer control from line 14 to line 2, there will be a set of LCSAJs which start from line 2 rather than from line 1. The reasoning for how far they go is exactly the same as for the LCSAJs starting from line 1, so the set of LCSAJs starting from line 2 is (2, 2, 15), (2, 4, 10), (2, 5, 9), (2, 9, 5), (2, 10, 13), and (2, 14, 2).

The complete set of LCSAJs are shown in the next clause. Note that the last LCSAJ is given as (15, 15, exit). We can start from line 15 because it is possible to jump to line 15 when the WHILE condition on line 2 is false. The convention is that any linear code sequence reaching the last line of the component has a `jump' to the program exit.

Test Cases

In this example, some test cases are used as a starting point. When these tests are analysed for LCSAJ coverage, a number of LCSAJs are shown as not covered.

The initial test input consists of one prime number (5), one non-prime number (6), and a special case number (2). All numbers are input to the component at once in a single test (without ending the program and restarting it). The initial test case set is:

Test Case	Input	Expected Outcome
1	5	5 is prime
	6	2 is a factor of 6
		3 is a factor of 6
	2	2 is prime
		End of prime number program

© British Computer Society, SIGIST, 2001

LCSAJ Coverage Analysis

The LCSAJs are shown below, together with coverage achieved by the initial test. Asterisks are used to highlight those LCSAJs which have not yet been covered. LCSAJ coverage is normally measured using a software tool.

LCSAJ			
START LINE	FINISH LINE	JUMP TO LINE	TIMES EXECUTED
1	2	15	0 ***
1	4	10	0 ***
1	5	9	1
1	9	5	0 ***
1	10	13	0 ***
1	14	2	0 ***
2	2	15	1
2	4	10	1
2	5	9	0 ***
2	9	5	1
2	10	13	0 ***
2	14	2	0 ***
5	5	9	0 ***
5	9	5	0 ***
5	10	13	1
5	14	2	0 ***
9	9	5	0 ***
9	10	13	0 ***
9	14	2	1
10	10	13	0 ***
10	14	2	1
13	14	2	1
15	15	exit	1

Number of LCSAJs	23
Number executed	9
Number not executed	14
LCSAJ Coverage	39%

Additional tests may now be devised to maximise the LCSAJ Coverage. These are described below.

LCSAJ	Comments
(1, 2, 15)	A new test is needed with no data, so the End of File is found immediately.
(1, 4, 10)	A new test is needed with a number less than 4 as first in the list.
(1, 9, 5)	A new test is needed with an even number greater than 5 as first in the list.
(1, 10, 13)	A new test is needed with the number 4 as first in the list.
(2, 5, 9)	This will be executed with an odd number greater than 4 which is not the first in the list.
(2, 10, 13)	This will be executed with the number 4 which is not the first in the list.
(5, 5, 9)	This will be executed by a number greater than 6.
(5, 9, 5)	This will be executed by an odd non-prime number greater than 7.
(9, 9, 5)	This will be executed by a number greater than 7.
(9, 10, 13)	This will be executed by an odd non-prime number greater than 8.

Infeasible LCSAJs

After executing line 7, Prime has the value false. The branch from line 10 to 11 requires Prime to be true. Hence, the following 3 LCSAJs whose linear code sequence contains at least the section from line 7 to line 11 are infeasible: (1, 14, 2), (2, 14, 2) and (5, 14, 2).

The remaining LCSAJ (10, 10, 13) is also infeasible. This LCSAJ requires Prime to be false on line 10. However, for the LCSAJ to start at line 10, it is necessary to jump from line 4 to line 10, i.e. a zero-trip loop. This implies that line 3 has been executed in the previous LCSAJ, setting Prime to true. This LCSAJ is not infeasible by itself but it is infeasible in any combination with others.

New Test Sets

Test Case	Input	Expected Output	LCSAJs executed
2	<none>	End of prime number program	(1, 2, 15)
3	2	2 is prime	(1, 4, 10)
	4	2 is a factor of 4	(2, 10, 13)
		End of prime number program	
4	8	2 is a factor of 8	(1, 9, 5)
		4 is a factor of 8	(5, 5, 9)
		End of prime number program	
5	4	2 is a factor of 4	(1, 10, 13)
	11	11 is prime	(2, 5, 9)
			(9, 9, 5)
			(5, 5, 9)
			(9, 10, 13)
		End of prime number program	

© British Computer Society, SIGIST, 2001

By running these extra tests, LCSAJ coverage is maximised with 19 of 23 LCSAJs executed, a measure of 83%.

B.13 Random Testing

Introduction

This black box technique requires no partitioning of the component's input domain, but simply requires test inputs to be chosen from this input domain at random. We illustrate the technique by means of an example.

Example

Consider a component that transforms coordinates, with the following specification:

> *The component shall transform the Cartesian coordinates (x,y) for screen position into their polar equivalent (r,H) using the equations: r= sqrt (x^2+y^2) and cos H = x/r. The origin of the Cartesian coordinates and the pole of the polar coordinates shall be the centre of the screen and the x-axis shall be considered the initial line for the polar coordinates progressing counter-clockwise. All inputs and outputs shall be represented as fixed-point numbers with both a range and a precision. These shall be:*
> *Inputs*
> *x - range -320..+320, in increments of $1/2^6$*
> *y - range -240..+240, in increments of $1/2^7$*
> *Outputs*
> *r - range 0..400, in increments of $1/2^6$*
> *H - range 0..((2*pi)-$1/2^6$), in increments of $1/2^6$*

No information is available about the operational distribution of the inputs to the component, so a uniform distribution is used. For each test case a random test input value is selected for both x and y, using the definitions provided in the component specification and based on a uniform distribution across their defined ranges. From the definitions we can see that in any one random test case x can take one of 40,961 values, while y can take one of 61,441 values. Care should be taken if using an *expected* operational distribution rather than a uniform distribution. An expected distribution that ignores parts of the input domain can lead to unexpected error conditions being left untested.

Random testing may be performed either manually or using automated test tools. Random testing is most cost-effective when fully automated as then very many tests can be run without manual intervention. However, to achieve full automation it must be possible to:

- automatically generate random test inputs; and

either
- automatically generate expected results from the specification;

or
- automatically check test outputs against the specification.

The automatic generation of random test input values is not difficult using a pseudo-random number generator as long as the component's inputs are well-defined. If the test input values are produced using a pseudo-random number generator, then these values do not need to be recorded explicitly as the same set can be reproduced. This is normally possible if a 'seed' value has been used to prime the pseudo-random number generator and this value is recorded.

The automatic generation of expected outputs or the automatic checking of outputs, is however more problematic. Generally it is not practicable to automatically generate expected outputs or automatically check outputs against the specification, however for certain components it is possible, such as where:

- trusted independently-produced software that performs the same function as the component under test is available (presumably not meeting the same constraints such as speed of processing, implementation language, etc.);
- the test is concerned solely with whether the component crashes or not (so the expected result is 'not to crash');
- the nature of the component's output makes it relatively easy to check the result. An example of this is a sort function where it is a simple task to automatically check that the outputs have been sorted correctly;
- it is easy to generate inputs from the outputs (using the inverse of the component's function). An example of this is a square root function where simply squaring the output should produce the input.

The coordinate transformation component can be checked automatically using the inverse function approach. In this case, $r\cos\theta = x$ can be obtained directly from the component's specification. By some analysis $r\sin\theta = y$ can also be deduced. If these two equations are satisfied to a reasonable numerical tolerance then the component has transformed the coordinates correctly.

Even when full automation of random testing is not practicable its use should still be considered as it does not carry the large overhead of designing test cases as required by the non-random techniques.

While no coverage measure is defined, a completion criterion can be set as a target level of probability of non-conforming outcome (e.g. 0.001) together with confidence interval bounds (say 95%).

For components with larger input sets than this small example the "Symbolic Input Attribute Decomposition" (SIAD) tree (ref: *Quality Programming*, C-K Cho, 1987, ISBN 0-471-84899-9) is a useful method for organising the input domain for random sampling before test case design.

B.14 Other Testing Techniques

Any techniques not explicitly defined that are used should be sent to the editors to be considered for inclusion in a later version of the Standard. These details can be submitted in a variety of ways, as described in clause E.1

© British Computer Society, SIGIST, 2001

Annex C Test Technique Effectiveness (informative)

Up to this point the Standard has provided no guidance on either the choice of test case design techniques or test completion criteria (sometimes known as test adequacy criteria), other than that they should be selected from clauses 3 and 4 respectively. The main reason for this is that there is no established consensus on which techniques and criteria are the most effective. The only consensus is that the selection will vary as it should be dependent on a number of factors such as criticality, application area, and cost. Research into the relative effectiveness of test case design and measurement techniques has, so far, produced no definitive results and although some of the theoretical results are presented below it should be recognised that they take no account of cost.

There is no requirement to choose corresponding test case design and test measurement techniques. In fact it is good practice to use functional test case design techniques and structural test measurement techniques. Functional techniques are effective at detecting errors of omission, while structural techniques can only detect errors of commission. So a test plan could typically require boundary value analysis to be used to generate an initial set of test cases, while also requiring 100% branch coverage to be achieved. This diverse approach could, presumably, lead to branch testing being used to generate any supplementary test cases required to achieve coverage of any branches missed by the boundary value analysis test case suite.

Ideally the test coverage levels chosen as test completion criteria should, wherever possible, be 100%. Strict definitions of test coverage levels have sometimes made this level of coverage impracticable, however the definitions in clause 4 have been defined to allow infeasible coverage items to be discounted from the calculations thus making 100% coverage an achievable goal.

With test completion criteria of 100% (and *only* 100%) it is possible to relate some of them in an ordering, where criteria are shown to subsume, or include, other criteria. One criterion is said to subsume another if, for all components and their specifications, every test case suite that satisfies the first criterion also satisfies the second. For example, branch coverage subsumes statement coverage because if branch coverage is achieved (to 100%), then statement coverage is always guaranteed to be achieved (to 100%) as well.

It should be noted that the 'subsumes' relation described here strictly links test coverage criteria (rather than test case design techniques) and so only provides an indirect indication of the relative effectiveness of test case design techniques.

Not all test coverage criteria can be related by the subsumes ordering and the functional and structural criteria are not related at all. This leads to a partial ordering of criteria, illustrated in figure C.1, where an arrow from one criterion to another indicates that the first criterion subsumes the second. Where a test coverage criterion does not appear in the partial orderings then it is not related to any other criterion by the subsumes relation.

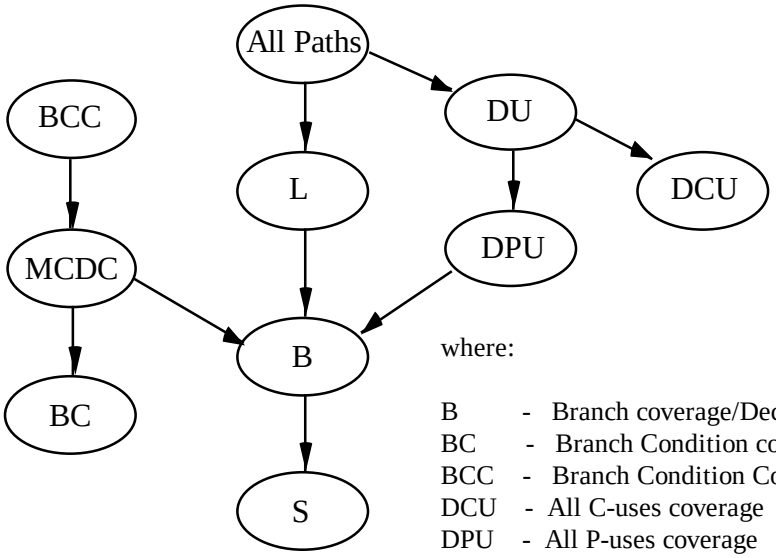


Figure C.1 Partial Ordering of Structural Test Coverage Criteria

Despite its intuitive appeal the subsumes relation suffers a number of limitations that should be considered before using it to choose test completion criteria:

- Firstly it relates only a subset of the available test completion criteria and inclusion in this subset provides no indication of effectiveness, so other criteria not shown in figure C.1 should still be considered.

- Secondly, the subsumes relation provides no measure of the amount by which one criterion subsumes another and subsequently does not provide any measure of relative cost effectiveness.

- Thirdly, the partial orderings only apply to single criteria while it is recommended that more than one criterion is used, with at least one functional and one structural criterion.

- Finally, and most importantly, the subsumes relation does not necessarily order test completion criteria in terms of their ability to expose errors (their test effectiveness). It has been shown, for instance, that 100% path coverage (when achievable) may not be as effective, for some components, as some of the criteria it subsumes, such as those concerned with data flow. This is because some errors are data sensitive and will not be exposed by simply executing the path on which they lie, but require variables to take a particular value as well (E.g. An 'unprotected' division by an integer variable may erroneously be included in a component that will only fail if that variable takes a negative value). Satisfying data flow criteria can concentrate the testing on these aspects of a component's behaviour, thus increasing the probability of exposing such errors. It can be shown that in some circumstances test effectiveness is increased by testing a subset of the paths required by a particular criteria but exercising this subset with more test cases.

The subsumes relation is highly dependent on the definition of full coverage for a criterion and although Figure C.1 is correct for the definitions in clause 4 of this Standard it may not apply to alternative definitions used elsewhere.

© British Computer Society, SIGIST, 2001

Annex D Bibliography (informative)

The following have been referenced in this Standard:

Chow, *T. S. Chow, Testing Software Design Modelled by Finite-State Machines, IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, May 1978.*

IEEE 829, *ANSI/IEEE 829 - 1998 (Software Test Documentation).*

IEEE 1008, *ANSI/IEEE 1008 - 1987 (Software Unit Testing).*

K&R, *Brian W Kernighan and Dennis M Richie, The C Programming Language, Second Edition, Prentice-Hall Software Series, 1988.*

Annex E Document Details (informative)

E.1 Method of commenting on this draft

Comments are invited on this draft so that the document can be improved to satisfy the requirement of an ISO standard.

When making a comment, be sure to include the following information:

- your name and contact details (address, telephone number, etc.);
- the version number of the standard;
- exact part of the standard;
- supporting information, such as the reason for a proposed change.

You can submit comments in a variety of ways, which in order of preference are as follows:

a) by post to Stuart Reid, Cranfield University DoIS/CISE, RMCS, Shrivenham, Swindon, Wilts, SN6 8LA, UK.

b) by fax to +44 1793 785366 (telephone +44 1793 785490).

c) by E-mail to Stuart.Reid@rmcs.cranfield.ac.uk.

E.2 Status

This standard has been amended as a result of the inspection carried out on May 4th 1995 and the comments received from external reviewers.

It has also had contact details updated in April 2001.

E.3 Availability

This standard is available on the World Wide Web at the following URL: http://www.testingstandards.co.uk/.

© British Computer Society, SIGIST, 2001