IEEE P29119-4/DISMay2013
Draft IEEE Standard
Systems and software engineering—Software testing—Part 4: Test techniques

Prepared by the Software and Systems Engineering Standards Committee of the
IEEE Computer Society
and
ISO/IEC JTC 1/SC 7

ISO/IEC TC JTC1/SC SC7 N 1

Date: 2013-04-30

ISO/IEC DIS 29119-4

ISO/IEC TC JTC1/SC SC7/WG 26

Secretariat: ANSI

Software and Systems Engineering — Software Testing — Part 4: Test Techniques

Document type: International Standard
Document subtype:
Document stage: (40) Enquiry
Document language: E

STD Version 2.1

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 29119-4 was prepared by Technical Committee ISO/IEC/TC JTC1, *Information Technology*, Subcommittee SC SC7, *Software and Systems Engineering*.

ISO/IEC 29119 consists of the following parts, under the general title *Software and Systems Engineering — Software Testing*:

- Part 1: Concepts and Definitions
- Part 2: Test Processes
- Part 3: Test Documentation
- Part 4: Test Techniques

Introduction

The purpose of ISO/IEC 29119-4 Test Techniques is to provide an International Standard that defines software test design techniques (also known as test case design techniques or test methods) that can be used during the test design and implementation process that is defined in ISO/IEC 29119-2 Test Processes. ISO/IEC 29119-4 does not prescribe a process for test design and implementation; instead, it describes a set of techniques that can be used within ISO/IEC 29119-2. The intent is to describe a series of techniques that have wide acceptance in the software testing industry.

The test design techniques presented in ISO/IEC 29119-4 can be used to derive test cases that, when executed, can be used to collect evidence that test item requirements have been met and/or that defects are present in a test item (i.e. that requirements have not been met). Risk-based testing could be used to determine the set of techniques that are applicable in specific situations (risk-based testing is covered in ISO/IEC 29119-1 and ISO/IEC 29119-2).

Each technique follows the test design and implementation process that is defined in ISO/IEC 29119-2 and shown below in Figure 1. Of the activities in this process, ISO/IEC 29119-4 provides guidance on how to implement the following activities in detail for each technique that is described:

— Derive Test Conditions (TD2),

— Derive Test Coverage Items (TD3), and

— Derive Test Cases (TD4).

A test condition is a testable aspect of a test item, such as a function, transaction, feature, quality attribute or structural element identified as a basis for testing. This determination can be achieved by agreeing with stakeholders which attributes are to be tested or by applying one or more test design techniques.

NOTE The value of test results and test coverage calculations can be diminished if test conditions do not reflect requirements in enough detail.

EXAMPLE 1 If a test completion criterion for state transition testing was identified that required coverage of all states then the test conditions could be the states the test item can be in. Other examples of test conditions are equivalence classes and boundaries between them or decisions in the code.

Test coverage items are attributes of each test condition that can be covered during testing. A single test condition may be the basis for one or more test coverage items.

EXAMPLE 2 If a specific boundary is identified as a test condition then the corresponding test coverage items could be the boundary itself and immediately either side of the boundary.

A test case is a set of preconditions, inputs (including actions, where applicable) and expected results, developed to determine whether or not the covered part of the test item has been implemented correctly.

Specific (normative) guidance on how to implement the other activities in the test design & implementation process of ISO/IEC 29119-2, including activities TD1 (Identify Feature Sets), TD5 (Assemble Test Sets) and TD6 (Derive Test Procedures) is not included in clauses 5 or 6 of this standard because the process is the same for all techniques.

ISO/IEC TR 19759 (SWEBOK) defines two types of requirements: functional requirements and quality requirements. ISO/IEC 25010 (ISO/IEC 25010:2011) defines eight quality characteristics (including functionality) that can be used to identify types of testing that may be applicable for testing a specific test item. Annex A provides example mappings of test design techniques that apply to testing quality characteristics defined in ISO/IEC 25010.

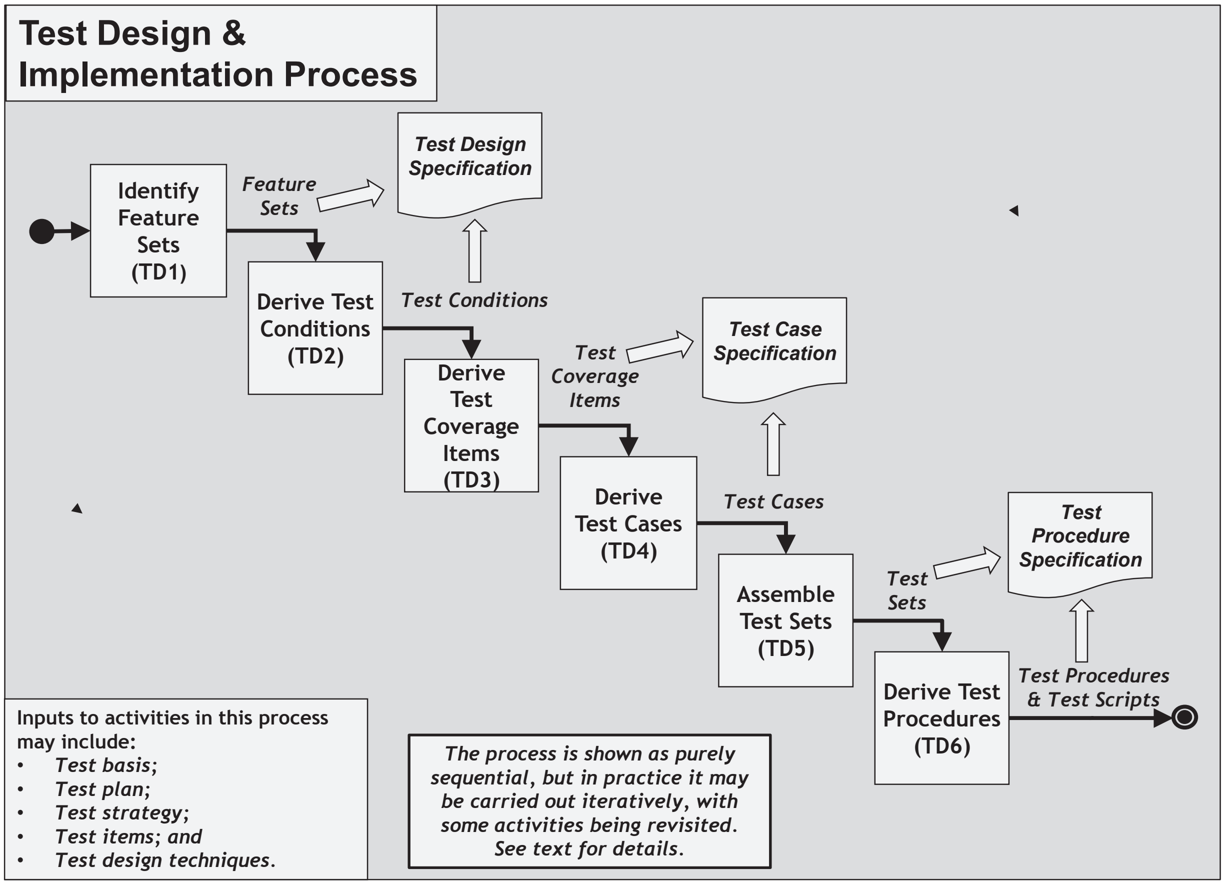

Test Design & Implementation Process
Identify Feature Sets (TD1)
Feature Sets
Test Design Specification
Derive Test Conditions (TD2)
Test Conditions
Derive Test Coverage Items (TD3)
Test Coverage Items
Test Case Specification
Derive Test Cases (TD4)
Test Cases
Assemble Test Sets (TD5)
Test Sets
Test Procedure Specification
Derive Test Procedures (TD6)
Test Procedures & Test Scripts
Inputs to activities in this process may include:
• Test basis;
• Test plan;
• Test strategy;
• Test items; and
• Test design techniques.
The process is shown as purely sequential, but in practice it may be carried out iteratively, with some activities being revisited. See text for details.

Figure 1 – ISO/IEC 29119-2 Test Design and Implementation Process

Experience-based testing practices like exploratory testing and other test practices such as model-based testing are not defined in ISO/IEC 29119-4 because this standard only describes techniques for designing test cases. Test practices such as exploratory testing are described in ISO/IEC 29119-1.

Templates and examples of test documentation that are produced during the testing process are defined in ISO/IEC 29119-3 Test Documentation. The test techniques in ISO/IEC 29119-4 do not describe how test cases should be documented (e.g. they do not include information or guidance on assigning unique identifiers, test case descriptions, priorities, traceability or pre-conditions). Information on how to document test cases can be found in ISO/IEC 29119-3.

This standard aims to provide stakeholders with the ability to perform software testing in any organization.

Software and Systems Engineering — Software Testing — Part 4: Test Techniques

1 Scope

ISO/IEC 29119-4 defines test design techniques that can be used during the test design and implementation process that is defined in ISO/IEC 29119-2.

This document is intended for, but not limited to, testers, test managers and developers, particularly those responsible for managing and implementing software testing.

2 Conformance

2.1.1 Intended Usage

The normative requirements in ISO/IEC 29119-4 are contained in clauses 5 and 6. It is recognized that particular projects or organizations may not need to use all of the techniques defined by this standard. Therefore, implementation of this standard typically involves selecting set of techniques suitable for the project or organization. There are two ways that an organizations or individual can claim conformance to the provisions of this standard. The organization shall assert whether it is claiming full or tailored conformance to this standard.

2.1.2 Full Conformance

Full conformance is achieved by demonstrating that all of the requirements (i.e. shall statements) of the chosen (non-empty) set of techniques have been satisfied.

EXAMPLE An organization could choose to conform only to one technique, such as boundary value analysis. In this scenario, the organization would only be required to provide evidence that they have met the requirements of that one technique in order to claim conformance to ISO/IEC 29119-4.

2.1.3 Tailored Conformance

Tailored conformance is achieved by demonstrating that the chosen subset of requirements from the chosen (non-empty) set of techniques have been satisfied. Where tailoring occurs, justification shall be provided whenever the normative requirements of a technique defined in clauses 5 and 6 are not followed completely (either directly or by reference). All tailoring decisions shall be recorded with their rationale, including the consideration of any applicable risks. Tailoring shall be agreed by the relevant stakeholders.

Any alternate test design technique that an organization wishes to claim conformance to (that is not already defined in this standard) shall satisfy the following criteria:

— The technique shall be freely available in the public domain.

— A source reference shall be provided.

— The technique shall be documented in the same manner as the other test techniques in clause 5.

— Associated test measurement techniques shall be documented in accordance with clause 6.1.1, if technically feasible.

3 Normative References

ISO/IEC 29119-4 does not require the use of any external normative references (i.e. there are no external standards or other referenced documents cited within “shall” statements of this standard that make them indispensable for the application of this standard). Standards useful for the implementation and interpretation of ISO/IEC 29119-4 are listed in the bibliography.

4 Terms and Definitions

For the purposes of this document, the terms and definitions given in ISO/IEC/IEEE 24765 *Systems and software engineering — Vocabulary* and the following apply.

NOTE Use of the terminology in ISO/IEC 29119-4 is for ease of reference and is not mandatory for conformance with the standard. The following terms and definitions are provided to assist with the understanding and readability of ISO/IEC 29119-4. Only terms critical to the understanding of ISO/IEC 29119-4 are included. This clause is not intended to provide a complete list of testing terms. The systems and software engineering vocabulary ISO/IEC/IEEE 24765 can be referenced for terms not defined in this clause. All terms defined in this clause are also intentionally included in ISO/IEC 29119-1, as that standard includes all terms that are used in ISO/IEC 29119 parts 1, 2, 3, 4 and 5.

4.1
Backus-Naur Form
formal metalanguage used for defining the syntax of a formal language

4.2
base choice
see base value

4.3
base value
input parameter value used in ‘base choice testing’ that is normally selected based on being a representative or typical value for the parameter. Also called base choice

4.4
c-use
see computation data use

4.5
computation data use
where the value of a variable is read in any statement other than a conditional expression. Also called c-use

4.6
condition
Boolean expression containing no Boolean operators

EXAMPLE “A < B” is a condition but “A and B” is not.

[SOURCE: BS 7925-1:1998, 3.45, modified — added quotation marks to example and removed “a” from start of definition]

4.7
control flow
abstract representation of all possible sequences of events in a test item’s execution

[SOURCE: BS 7925-1:1998, 3.50, modified — replaced “program’s” with “test item’s” and removed “an” from start of definition]

4.8
control flow sub-path
sequence of executable statements within a test item

4.9
data definition
see variable definition

4.10
data definition c-use pair
data definition and computation data use, where the data use uses the value defined in the data definition

[SOURCE: BS 7925-1:1998, 3.59, modified — removed “a” from start of definition]

4.11
data definition p-use pair
data definition and predicate data use, where the data use uses the value defined in the data definition

[SOURCE: BS 7925-1:1998, 3.61, modified — removed “a” from start of definition]

4.12
data definition-use pair
data definition and data use, where the data use uses the value defined in the data definition

[SOURCE: BS 7925-1:1998, 3.63, modified — removed “a” from start of definition]

4.13
data use
executable statement where the value of a variable is accessed

[SOURCE: BS 7925-1:1998, 3.67, modified — removed “an” from start of definition]

4.14
decision
point in a test item at which the control flow has two or more alternative routes

[SOURCE: BS 7925-1:1998, 3.69, modified — replaced “program point” with “point in a test item” and removed “a” from start of definition]

4.15
decision outcome
result of a decision (which therefore determines the control flow alternative taken)

[SOURCE: BS 7925-1:1998, 3.72, modified — removed “the” from start of definition]

4.16
decision rule
combination of conditions (also known as causes) and actions (also known as effects) that produce a specific outcome in decision table testing and cause-effect graphing.

4.17
definition-use path
control flow sub-path from a variable definition to a predicate-use (p-use) or computational-use (c-use) of that variable

4.18
definition-use pair
data definition and subsequent predicate or computational data use, where the data use uses the value defined in the data definition

4.19
entry point
first executable statement within a test item

[SOURCE: BS 7925-1:1998, 3.81, modified — added “the” and replaced “component” with “test item”]

4.20
executable statement
statement which, when compiled, is translated into object code, which will be executed procedurally when the test item is running and may perform an action on program data

[SOURCE: BS 7925-1:1998, 3.89, modified — replaced “program” with “test item” and removed “a” from start of definition]

4.21
exit point
last executable statement within a test item

[SOURCE: BS 7925-1:1998, 3.92, modified — replaced “component” with “test item” and removed “a” from start of definition]

4.22
expected result
observable behaviour of the test item under specified conditions, inferred from or predicted by its test basis or another source

[SOURCE: BSI 7925-1:1998, 3.93, modified — added “the” before “test item”, and replaced “object” with “test item” and replaced “specification” with “test basis”]

4.23
p-use
see predicate data use

4.24
P-V pair
combination of a test item parameter with a value assigned to that parameter, used as a test condition and coverage item in combinatorial test design techniques

4.25
path
sequence of executable statements of a test item, from an entry point to an exit point

[SOURCE: BS 7925-1:1998, 3.138, modified — replaced “component” with “test item” and removed “a” from start of definition]

4.26
predicate
logical expression which evaluates to TRUE or FALSE, normally to direct the execution path in code

[SOURCE: BS 7925-1:1998, 3.145]

4.27
predicate data use
data use associated with the decision outcome of the predicate portion of a decision statement

[SOURCE: BS 7925-1:1998, 3.146, modified — removed "a" from start of definition]

4.28
test model
representation of a test item that is used during the test case design process

4.29
variable definition
statement where a variable is assigned a value. Also called data definition

[SOURCE: BS 7925-1:1998, 3.57, modified — definition moved from "data definition" to "variable definition" and removed "a" from start of definition]

5 Test Design Techniques

5.1 Overview

ISO/IEC 29119-4 defines test design techniques for specification-based testing (clause 5.2), structure-based testing (clause 5.3) and experience-based testing (clause 5.4). In specification-based testing, the test basis (e.g. requirements, specifications, models or user needs) is used as the main source of information to design test cases. In structure-based testing, the structure of the test item (e.g. source code or the structure of a model) is used as the primary source of information to design test cases. In experience-based testing, the knowledge and experience of the tester is used as the primary source of information during test case design. For specification-based testing, structure-based testing and experience-based testing, the test basis is used to generate the expected results. These classes of test design techniques are complementary and their combined application typically results in more effective testing.

Although the techniques presented in ISO/IEC 29119-4 are classified as structure-based, specification-based or experience-based, in practice they can be used interchangeably (e.g. branch testing could be used to design test cases for testing logical paths through the graphical user interface of an Internet-based system). In addition, although each technique is defined independently of all others, in practice they can be used in combination with other techniques.

EXAMPLE The test coverage items derived by applying equivalence partitioning could be used to populate input parameters of test cases derived using scenario testing.

ISO/IEC 29119-4 uses the terms specification-based testing and structure-based testing; however, these categories of techniques are also known as "black-box testing" and "white-box testing" (or "clear-box testing") respectively. The terms "black-box" and "white-box" refer to the visibility of the internal structure of the test item. In black-box testing the internal structure of the test item is not visible (hence the black box), whereas for white-box testing the internal structure of the test item is visible. When a technique is applied while utilising a combination of knowledge from the test item's specification and structure, this is often called "grey-box testing".

ISO/IEC 29119-4 defines how the generic test design and implementation process steps TD2 (derive test conditions), TD3 (derive test coverage items), and TD4 (derive test cases) from ISO/IEC 29119-2 (see Introduction) shall be used by each technique. It does not provide context-specific definitions of the techniques that describe how each technique should be used in all situations. Users of ISO/IEC 29119-4 may refer to the informative annex of ISO/IEC 29119-4 for detailed examples that demonstrate how to apply the techniques.

The techniques that are defined in ISO/IEC 29119-4 are shown below in Figure 2. This set of techniques is not exhaustive. There are techniques that are used by testing practitioners or researchers that are not included in ISO/IEC 29119-4.

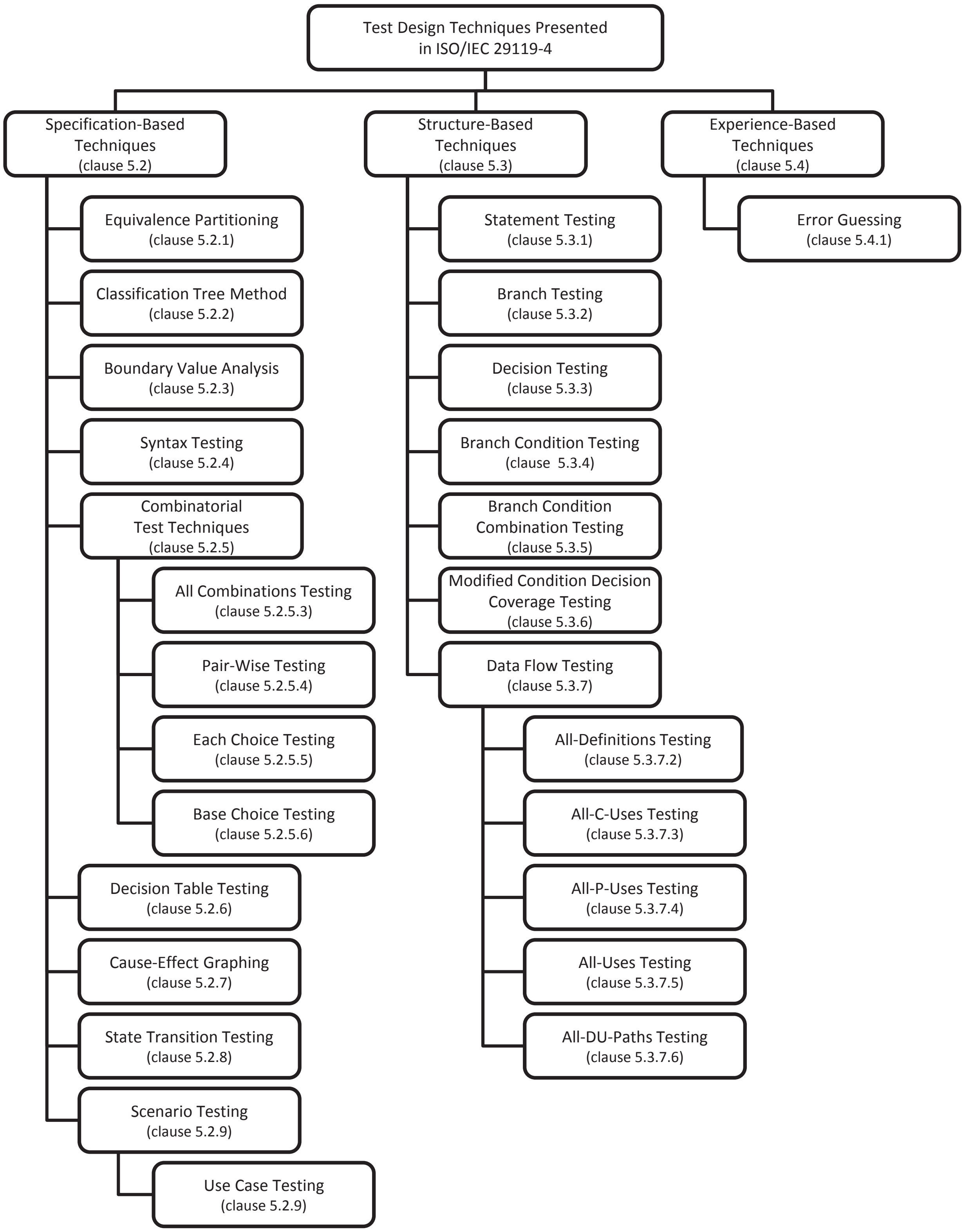

Test Design Techniques Presented in ISO/IEC 29119-4
Specification-Based Techniques (clause 5.2)
Equivalence Partitioning (clause 5.2.1)
Classification Tree Method (clause 5.2.2)
Boundary Value Analysis (clause 5.2.3)
Syntax Testing (clause 5.2.4)
Combinatorial Test Techniques (clause 5.2.5)
All Combinations Testing (clause 5.2.5.3)
Pair-Wise Testing (clause 5.2.5.4)
Each Choice Testing (clause 5.2.5.5)
Base Choice Testing (clause 5.2.5.6)
Decision Table Testing (clause 5.2.6)
Cause-Effect Graphing (clause 5.2.7)
State Transition Testing (clause 5.2.8)
Scenario Testing (clause 5.2.9)
Use Case Testing (clause 5.2.9)
Structure-Based Techniques (clause 5.3)
Statement Testing (clause 5.3.1)
Branch Testing (clause 5.3.2)
Decision Testing (clause 5.3.3)
Branch Condition Testing (clause 5.3.4)
Branch Condition Combination Testing (clause 5.3.5)
Modified Condition Decision Coverage Testing (clause 5.3.6)
Data Flow Testing (clause 5.3.7)
All-Definitions Testing (clause 5.3.7.2)
All-C-Uses Testing (clause 5.3.7.3)
All-P-Uses Testing (clause 5.3.7.4)
All-Uses Testing (clause 5.3.7.5)
All-DU-Paths Testing (clause 5.3.7.6)
Experience-Based Techniques (clause 5.4)
Error Guessing (clause 5.4.1)

Figure 2 – The set of test design techniques presented in ISO/IEC 29119-4

Of the six activities in the test design and implementation process (see Figure 1), test techniques provide unique and specific guidance on the derivation of test conditions (TD2), test coverage items (TD3) and test cases (TD4). Therefore, each technique is defined in terms of these three activities.

There are varying levels of granularity within steps TD2 (derive test conditions), TD3 (derive test coverage items) and TD4 (derive test cases). Within each technique, the term "model" is used to describe the concept of preparing a logical representation of the test item for the purposes of deriving test conditions in step TD2 (e.g. a control flow model is required for deriving test conditions for all structural techniques). Some situations may require the entire model to be a test condition, whereas in other situations, one part of the model may be a test condition.

EXAMPLE 1 In state transition testing, if there is a requirement to cover all states then the entire state model could be the test condition. Alternatively, if there is a requirement to cover specific transitions between states, then each transition could be a test condition.

In addition, since some techniques share underlying concepts, their definitions contain similar text.

EXAMPLE 2 Both equivalence partitioning and boundary value analysis are based on equivalence classes.

In the test case design step (TD4) of each technique, test cases that are created may be “valid” (i.e. they contain input values that the test item should accept as correct) or “invalid” (i.e. they contain at least one input value that the test item should reject as incorrect, ideally with an appropriate error message). In some techniques, such as equivalence partitioning and boundary value analysis, invalid test cases are usually derived using the “one-to-one” approach as it avoids fault masking by ensuring that each test case only includes one invalid input value, while valid test cases are typically derived using the “minimized” approach, as this reduces the number of test cases required to cover valid test coverage items (see 5.2.1.3 and 5.2.3.3).

NOTE Invalid cases are also known as "negative test cases".

Although the techniques defined in ISO/IEC 29119-4 are each described separately (as if they were mutually exclusive), in practice they could be applied in a blended way. For example, boundary value analysis could be used to select test input values, after which pair-wise testing could be used to design test cases from the test input values. Another example is using equivalence partitioning to select the classifications and classes for the classification tree method and then using each choice testing to construct test cases from the classes.

The techniques presented in ISO/IEC 29119-4 could also be used in conjunction with the test types that are presented in Annex A. For example, equivalence partitioning could be used to identify user groups (test conditions) and users (test coverage items) that are to be tested during usability testing.

The normative definitions of the techniques are provided in clause 5. The corresponding normative coverage measures for each technique are presented in clause 6. This is supported by informative examples of each technique in Annexes B, C and D. Although the examples of each technique demonstrate manual application of the technique, in practice, automation can be used to support some types of test design and execution (e.g. statement coverage analyzers can be used to support structure-based testing). Annex A provides examples of how the test design techniques defined in this standard can be applied to testing the quality characteristics that are defined in ISO/IEC 25010.

5.2 Specification-Based Test Design Techniques

5.2.1 Equivalence Partitioning

5.2.1.1 Derive Test Conditions (TD2)

Equivalence partitioning (BS 7925-2:1998; Myers 1979) uses a model of the test item that partitions the inputs and outputs of the test item into equivalence partitions (also called “partitions” or “equivalence classes”), where each equivalence partition shall be defined as a test condition. These equivalence partitions shall be derived from the test basis, where each partition is chosen such that all values within the equivalence partition can reasonably be expected to be treated similarly (i.e. they may be considered "equivalent") by the test item. Equivalence partitions may be derived for both valid and invalid inputs and outputs.

EXAMPLE For a test item expecting lowercase alphabetical characters as (valid) inputs, invalid input equivalence partitions that could be derived include equivalence partitions containing integers, reals, uppercase alphabetical characters, symbols and control characters, depending on the level of rigour required during testing.

NOTE 1 Invalid equivalence partitions can be created for both input and output values for the test item. Invalid output equivalence partitions typically correspond to any outputs that have not been explicitly specified. As these are not specified their identification is a creative activity and hence often results in equivalence partitions based on the subjectivity of the individual tester.

NOTE 2 Domain analysis (Beizer 1995) is often classified as a combination of equivalence partitioning and boundary value analysis.

5.2.1.2 Derive Test Coverage Items (TD3)

Each equivalence partition shall also be identified as a test coverage item (i.e. for equivalence partitioning the test conditions and test coverage items are the same equivalence partitions).

5.2.1.3 Derive Test Cases (TD4)

Test cases shall be derived to exercise the test coverage items (i.e. the equivalence partitions). The following steps shall be used during test case derivation:

1. Decide on an approach for selecting combinations of test coverage items to be exercised by test cases, where two common approaches are (BS 7925-2:1998; Myers 1979):

    a. one-to-one, in which each test case is derived to cover a specific equivalence partition; and

NOTE 1 The number of test cases derived using one-to-one equivalence partitioning equals the number of test coverage items.

    b. minimized, in which equivalence partitions are covered by test cases such that the minimum number of test cases derived covers all equivalence partitions at least once.

NOTE 2 Each test case derived using minimized equivalence partitioning could cover multiple test coverage items.

NOTE 3 Other approaches to selecting combinations of test coverage items to be exercised by test cases are described in clause 5.2.5 (Combinatorial Test Design Techniques).

2. Select test coverage item(s) for inclusion in the current test case based on the approach chosen in step 1;

3. Identify input values to exercise the test coverage items to be covered by the test case and arbitrary valid values for any other input variables required by the test case;

4. Determine the expected result of the test case by applying the input(s) to the test basis; and

5. Repeat steps 2 to 4 until the required level of test coverage is achieved.

5.2.2 Classification Tree Method

5.2.2.1 Derive Test Conditions (TD2)

The classification tree method (Grochtmann and Grimm 1993) uses a model of the test item that partitions the inputs of the test item and represents them graphically in the form of a tree called a classification tree. The test item’s inputs are partition into “classifications”, where each classification is disjoint (non-overlapping) and the set of classifications are complete (all inputs are included). Each classification shall be a test condition. “Classes” that result from decomposing the classifications may be partitioned further into “sub-classes” depending on the level of rigour required in the testing. Classifications and classes may be derived for both valid and invalid input data, depending on the level of test coverage required.

NOTE The process of partitioning in the classification tree method is similar to equivalence partitioning. The key difference is that in the classification tree method, the partitions (which are classifications and classes) must be completely disjoint, whereas in equivalence partitioning, they could overlap depending on how the technique was applied. In addition, the classification tree method also includes the design of a classification tree, which provides a visual representation of the test conditions.

5.2.2.2 Derive Test Coverage Items (TD3)

Test coverage items shall be derived by combining classes using a chosen combination approach.

EXAMPLE Two example approaches for combining classes into test coverage items are:

— minimal, in which classes are included in test coverage items such that the minimum number of test coverage items are derived to cover all classes at least once;

— maximal, in which classes are included in test coverage items such that each possible combination of classes is covered by at least one test coverage item.

NOTE 1 Other approaches to selecting combinations of test coverage items are described in clause 5.2.5 (Combinatorial Test Design Techniques).

NOTE 2 The test coverage items are often illustrated in a combination table (see Figure 8 in clause B.2.2.5).

5.2.2.3 Derive Test Cases (TD4)

Test cases shall be derived to exercise the test coverage items. The following steps shall be followed during test case derivation:

1. Based on the combinations of classes created in step TD3, select one combination for inclusion in the current test case that has not already been covered by a test case;
2. Identify input values for any classes that do not already have an assigned value;
3. Determine the expected result of the test case by applying the input(s) to the test basis; and
4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.2.3 Boundary Value Analysis

5.2.3.1 Derive Test Conditions (TD2)

Boundary value analysis (BS 7925-2:1998; Myers 1979) uses a model of the test item that partitions the inputs and outputs of the test item into a number of ordered sets with identifiable boundaries, where the boundaries of each set are test conditions. The boundaries shall be derived from the test basis.

EXAMPLE If a partition was defined for inputs from 1 to 10, then there are two boundaries, 1 and 10, and these are the test conditions.

5.2.3.2 Derive Test Coverage Items (TD3)

In boundary value analysis, one of the following two options for the derivation of test coverage items shall be applied:

— two-value boundary testing; and

— three-value boundary testing.

For *two-value boundary testing*, two test coverage items shall be derived for each boundary (test condition) corresponding to values on the boundary and an incremental distance *outside* the boundary of the equivalence partition. This incremental distance shall be defined as the smallest significant value for the data type under consideration.

For *three-value boundary testing*, three test coverage items shall be derived for each boundary (test condition) corresponding to values on the boundary and an incremental distance *each* side of the boundary of the equivalence partition. This incremental distance shall be defined as the smallest significant value for the data type under consideration.

NOTE 1 Some partitions could have only a single boundary identified in the test basis. For example, the numerical partition “age $\geq$ 70 years” has a lower boundary but not an obvious upper boundary. In some cases a boundary value imposed by the actual implementation can be used as a boundary value, such as the largest value that is accepted by the input field (such decisions should be documented; for example, in the Test specification documentation).

NOTE 2 Two-value boundary value testing is typically adequate in most situations; however, three-value boundary testing could be required for certain circumstances (e.g. for rigorous testing to check no errors were made in determining the boundaries of variables in the test item by both testers and developers).

NOTE 3 In both two- and three-value boundary value testing, contiguous partitions (partitions that share a boundary) will result in duplicate test coverage items, in which case it is typical practice to only exercise these duplicated values once. For an example of duplicate boundaries, see clause B.2.3.4.3.

5.2.3.3 Derive Test Cases (TD4)

Test cases shall be derived to exercise the test coverage items. The following steps shall be used during test case derivation:

1. Decide on an approach for selecting combinations of test coverage items to be exercised by test cases, where two common approaches are (BS 7925-2:1998; Myers 1979):

   a. one-to-one, in which each test case is derived to exercise a specific boundary value; or

NOTE 1 In one-to-one boundary value analysis, the number of test cases derived equals the number of test coverage items.

   b. minimized, in which boundary values are included in test cases such that the minimum number of test cases are derived to cover all boundary values at least once.

NOTE 2 In minimized boundary value analysis, test cases can cover multiple test coverage items.

NOTE 3 Other approaches to selecting combinations of test coverage items to be exercised by test cases are described in clause 5.2.5 (Combinatorial Test Design Techniques).

2. Select test coverage item(s) for inclusion in the current test case based on the approach chosen in step 1;

3. Identify arbitrary valid values for any other input variables required by the test case that were not already selected in step 2;

4. Determine the expected result of the test case by applying the input(s) to the test basis; and

5. Repeat steps 2 to 4 until the required level of test coverage is achieved.

5.2.4 Syntax Testing

5.2.4.1 Derive Test Conditions (TD2)

Syntax testing (Beizer 1995; Burnstein 2003) uses a formally-defined syntax model of the inputs to a test item that is represented as a number of rules, where each rule defines the format of an input parameter to the test item in terms of sequences of, iterations of, or selections between, syntax elements. The syntax may be represented diagrammatically or in a textual format. The test condition in syntax testing shall either be the model that describes the syntax of the inputs to the test item or the syntax of each individual input parameter.

EXAMPLE 1 An abstract syntax tree can be used to represent formal syntax diagrammatically.

EXAMPLE 2 Backus-Naur Form can be used to represent formal syntax in a textual format.

5.2.4.2 Derive Test Coverage Items (TD3)

In syntax testing the test coverage items shall be “options” and/or “mutations” of the defined syntax.

The following guidelines may be used to derive “options” (although alternate guidelines may be used where appropriate):

— whenever selection is mandated by the syntax, an “option” is derived for each alternative provided for that selection;

EXAMPLE 1 For the input parameter “colour = Blue | Red | Green” (where | represents the “OR” Boolean operator), three options “Blue”, “Red” and “Green” will be derived as test coverage items.

— whenever an iteration is mandated by the syntax, at least two “options” are derived for the iteration; one with the minimum number of repetitions and the other with more than the minimum number of repetitions;

EXAMPLE 2 For the input parameter “letter = $[A – Z | a – z]^+$” (where “$^+$” represents “one or more”), two options, “one letter” and “more than one letter”, will be derived as test coverage items.

— whenever iteration is mandated with a maximum number of repetitions, at least two “options” are derived for the iteration; one with the maximum number of repetitions and the other with more than the maximum number of repetitions.

EXAMPLE 3 For the input parameter “letter = $[A – Z | a – z]^{100}$” (where “^{100}” represents the fact that a letter can be chosen up to 100 times), two options “100 letters” and “more than 100 letters” will be derived as test coverage items.

The following guideline may be used to derive “mutations” (although alternate guidelines may be used where appropriate):

— for any input, the defined syntax may be mutated to derive invalid inputs (“mutations”).

EXAMPLE 4 For the input parameter “colour = Blue | Red | Green”, one “mutation” could be to introduce an invalid value for the parameter as the test coverage item, such as selecting the value “Yellow”, which does not appear in the input parameter list. Other example mutations are provided in Annex B, clause B.2.4.5.

5.2.4.3 Derive Test Cases (TD4)

Test cases for syntax testing shall be derived to cover the chosen options and mutations. The following steps shall be used during test case derivation:

1. Decide on an approach for selecting combinations of test coverage items to be exercised by test cases, where two common approaches are:

    a. one-to-one, in which each test case is derived to exercise a specific option and/or mutation; or

b. minimized, in which options and/or mutations are included in test cases such that the minimum number of test cases are derived to cover all options and/or mutations at least once.

NOTE Other approaches to selecting combinations of test coverage items to be exercised by test cases are described in clause 5.2.5 (Combinatorial Test Design Techniques).

2. Select test coverage item(s) for inclusion in the current test case;
3. Identify input values to exercise the test coverage item(s) to be covered by the test case and arbitrary valid values for any other input variables required by the test case;
4. Determine the expected result of the test case by applying the input(s) to the test basis; and
5. Repeat steps 1 to 4 until the required level of test coverage is achieved.

5.2.5 Combinatorial Test Design Techniques

5.2.5.1 Overview

Combinatorial test design techniques are used to systematically derive a meaningful and manageable subset of test cases that cover the test conditions and test coverage items that are derivable during testing. The combinations of interest are defined in terms of test item parameters and the values these parameters can take.

5.2.5.2 Derive Test Conditions (TD2)

The test item parameters represent particular aspects of the test item that are relevant to the testing, and often correspond to the input parameters to the test item, but other aspects may also be used.

EXAMPLE 1 In configuration testing, the parameters could be various environment factors, such as operating system and browser.

Each test item parameter can take on various values. For use in this technique the set of values needs to be finite and manageable. Some test item parameters may be naturally constrained to only take on a small set of possible values while other test item parameters may be far less constrained.

EXAMPLE 2 A test item parameter that is naturally constrained to a small number of values is a “day_of_week” parameter that can only take on seven specific values “[Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday]”.

EXAMPLE 3 A test item parameter that is far less constrained is a parameter consisting of any Real number, which exists along a potentially infinite number line.

For unconstrained test item parameters, it may be necessary to apply other testing techniques, such as equivalence partitioning or boundary value analysis, to reduce a large set of possible values for a parameter to a manageable subset.

The test conditions for combinatorial testing are the same for all the combinatorial test design techniques; they shall be a chosen test item parameter (P) taking on a specific value (V), resulting in a P-V pair.

5.2.5.3 All Combinations Testing

5.2.5.3.1 Derive Test Coverage Items (TD3)

In ‘all combinations’ testing (Grindal, Offutt and Andler 2005) the set of test coverage items shall be the set of all unique combinations of P-V pairs, where each combination is made up of one P-V pair for each test item parameter.

5.2.5.3.2 Derive Test Cases (TD4)

Test cases shall be derived in which each test case exercises one unique combination of P-V pairs. The following steps shall be used during test case derivation:

1. Select test coverage item(s) for inclusion in the current test case that have not already been covered by a test case;

2. Determine the expected result of the test case by applying the input(s) to the test basis; and

3. Repeat steps 1 and 2 until the required level of test coverage is achieved.

NOTE The minimum number of test cases required to achieve 100% all combinations testing corresponds to the product of the number of P-V pairs for each test item parameter.

5.2.5.4 Pair-wise Testing

5.2.5.4.1 Derive Test Coverage Items (TD3)

In pair-wise testing (Grindal, Offutt and Andler 2005) (also known as "all pairs" testing), the test coverage items shall be unique pairs of P-V pairs, where each P-V pair within the pair is for a different test item parameter. Instead of all possible combinations of the parameters (as was required for all combinations testing), this technique covers all possible pairs of the selected values within the total set, thereby achieving extensive coverage with a potentially smaller set of test cases.

5.2.5.4.2 Derive Test Cases (TD4)

Having first identified the P-V pairs, test cases shall be derived to exercise pairs of P-V pairs, where each test case exercises one or more unique pairs.

The following steps shall be used during test case derivation:

1. Select test coverage item(s) for inclusion in the current test case, where each pair of P-V pairs covers a different pair of parameters that have not yet been included in a test case;

2. Identify arbitrary valid values for any other parameters present in the test case;

3. Determine the expected result of the test case by applying the input(s) to the test basis; and

4. Repeat steps 1 to 3 until all unique pairs of P-V pairs have been exercised.

The minimum number of test cases required to achieve 100% pair-wise testing is not easily calculated. A near-optimal set may be considered acceptable and may be calculated using one of the following three options:

— manually determine a near-optimal set using an algorithm;

— use an automated tool (implementing an algorithm) to determine a near-optimal set; or

— use orthogonal arrays (Mandl 1985) to determine a near-optimal set.

5.2.5.5 Each Choice Testing

5.2.5.5.1 Derive Test Coverage Items (TD3)

In each choice (or 1-wise) testing (Grindal, Offutt and Andler 2005), the test coverage items shall be the set of P-V pairs.

5.2.5.5.2 Derive Test Cases (TD4)

Test cases shall be derived to exercise P-V pairs, where each test case exercises one or more P-V pairs that have not been previously included in a test case.

The following steps shall be used during test case derivation:

1. Select test coverage item(s) for inclusion in the current test case, where at least one selected test coverage item has not been included in a prior test case;
2. Identify arbitrary valid values for any other parameters present in the test case;
3. Determine the expected result of the test case by applying the input(s) to the test basis; and
4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

The minimum number of test cases required to achieve 100% each choice testing corresponds to the maximum number of values any one of the test item parameters can take.

5.2.5.6 Base Choice Testing

5.2.5.6.1 Derive Test Coverage Items (TD3)

In base choice testing (Grindal, Offutt and Andler 2005), the test coverage items shall be sets of P-V pairs for each of the input parameters, where all but one parameter is set to its "base" value and the final parameter is set to one of its other valid values.

NOTE There are a number of approaches for choosing the base values for each parameter. For example, they can be chosen from the operational profile, from the typical path in scenario testing, from the test coverage items that are derived during equivalence partitioning or from the default (most frequently used) values for the parameter.

5.2.5.6.2 Derive Test Cases (TD4)

Having first identified the P-V pairs, base choices for each of the parameters shall be chosen. Test cases shall be derived by setting all but one parameter to its base choice and then setting the final parameter to a valid value until the required level of test coverage of PV-pairs is achieved.

The following steps shall be used during test case derivation:

1. Derive the base-choice test case by setting each parameter to its "base" value;
2. Create a new test case by setting one parameter to a valid (non-base choice) value while keeping the remaining set of parameters to their base choice values;
3. Determine the expected result of the new test case by applying the inputs to the test basis; and
4. Repeat steps 2 and 3 until the required level of test coverage is achieved.

5.2.6 Decision Table Testing

5.2.6.1 Derive Test Conditions (TD2)

Decision table testing (BS 7925-2:1998, Myers 1979) uses a model of the logical relationships (decision rules) between conditions (causes) and actions (effects) for the test item in the form of a decision table, where:

— each condition is an input or a combination of inputs to the test item expressed as a Boolean; and

— each action is an outcome or a combination of outcomes for the test item expressed as a Boolean.

The test conditions shall be the conditions and actions.

5.2.6.2 Derive Test Coverage Items (TD3)

In decision table testing, each decision rule, which defines the relationship between a unique combination of the test item's conditions and actions, is a test coverage item.

5.2.6.3 Derive Test Cases (TD4)

Test cases shall be derived to exercise the decision rules (test coverage items), where each test case defines the relationship between the inputs and outputs, and each decision rule corresponds to a unique combination of inputs. The following steps shall be used during test case derivation:

1. Select test coverage item(s) from the decision table for implementation as a test case;

2. Identify input values to satisfy the input condition(s) of the decision rule(s) to be covered by the test case and arbitrary valid values for any other input variables required to execute the test case;

3. Determine the expected result of the test case by applying the input(s) to the decision table; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

NOTE If the decision table contains dependent input conditions then this could result in infeasible combinations (e.g. "age less than 18" and "age greater than 65" both set to true). In this situation such infeasible decision rules should be identified and documented and are typically not used to derive test cases (e.g. they could be marked as "don't care").

5.2.7 Cause-Effect Graphing

5.2.7.1 Derive Test Conditions (TD2)

Cause-effect graphing (BS 7925-2:1998, Myers 1979) uses a model of the logical relationships (decision rules) between causes (e.g. inputs) and effects (e.g. outputs) for the test item in the form of a cause-effect graph, where:

— each cause is an input or a combination of inputs to the test item expressed as a Boolean; and

— each effect is an outcome or a combination of outcomes for the test item expressed as a Boolean.

The test conditions shall be the causes and effects.

5.2.7.2 The cause-effect graph illustrates relationships between causes and effects using Boolean operators and optionally cause and effect constraints (see figures 14 and 15 in Annex B).Derive Test Coverage Items (TD3)

In cause-effect graphing, each decision rule, which defines the relationship between a unique combination of the test item's causes and effects, is a test coverage item.

5.2.7.3 Derive Test Cases (TD4)

Test cases shall be derived to exercise the test coverage items. A corresponding decision table may be produced from the cause-effect graph and used to derive the test cases. The following steps shall be used during test case derivation:

1. Select test coverage item(s) to be implemented in the current test case;

2. Identify input values to exercise the test coverage item(s) to be covered by the test case and arbitrary valid values for any other input variables required to execute the test case;

3. Determine the expected result of the test case by applying the input(s) to the cause-effect graph and/or decision table; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.2.8 State Transition Testing

5.2.8.1 Derive Test Conditions (TD2)

State transition testing (BS 7925-2:1998, Copeland 2004) uses a model of the states the test item may occupy, the transitions between states, the events which cause transitions and the actions that may result from the transitions. The states of the model shall be discrete, identifiable and finite in number. In state transition testing, the test conditions may be all states of the state model, all transitions of the state model or the entire state model, depending on the coverage requirements of testing. The model may be represented as a state transition diagram or a state table (although other representations may also be used).

5.2.8.2 Derive Test Coverage Items (TD3)

In state transition testing, test coverage items will change depending on the chosen test completion criterion and test design approach. Possible test completion criterions include but are not limited to the following:

— states, in which test coverage items shall be derived to enable all states in the state model to be "visited";

— single transitions (0-switch coverage), in which test coverage items shall be derived to cover valid single transitions in the state model;

— all transitions, in which test coverage items shall be derived to cover *both* valid transitions in the state model and "invalid" transitions (transitions from states initiated by events in the state model for which no valid transition is specified);

— multiple transitions (N-switch coverage), in which test coverage items shall be derived to cover N+1 valid sequential transitions in the state model.

NOTE 1-switch coverage is a popular variant of N-switch coverage that requires pairs of transitions to be exercised.

5.2.8.3 Derive Test Cases (TD4)

Test cases for state transition testing shall be derived to exercise the test coverage items. The following steps shall be used during test case derivation:

1. Select test coverage item(s) for inclusion in the current test case;

2. Identify input values to exercise the test coverage item(s) to be covered by the test case;

3. Determine the expected result of the test case by applying the input(s) to the test basis (the expected result may be defined in terms of outputs and the states visited as described in the state model); and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.2.9 Scenario Testing

5.2.9.1.1 Derive Test Conditions (TD2)

Scenario testing uses a model of the sequences of interactions between the test item and other systems (in this context users are often considered to be other systems) for the purpose of testing usage flows involving the test item. Test conditions shall either be one sequence of interactions (i.e. one scenario) or all sequences of interactions (i.e. all scenarios).

In scenario testing, this step shall include identification of:

— the “main” scenario which is the typical sequence of actions that are expected of the test item or an arbitrary choice when no typical sequence of actions is known; and

— “alternate” scenarios that represent alternate (non-main) scenarios that may be taken through the test item.

NOTE 1 Alternate scenarios can include abnormal use, extreme or stress conditions and exceptions.

NOTE 2 Scenario testing is typically used for conducting “end-to-end testing” during functional testing, such as during system testing or user acceptance testing.

One common form of scenario testing called use case testing (Bath 2008; Hass 2008) utilizes a use case model of the test item that describes how the test item interacts with one or more actors for the purpose of testing sequences of interactions (i.e. scenarios) involving the test item.

NOTE 3 In use case testing, the use case model describes how various actions are performed by the test item as a result of various triggers from the actors. An actor may be a user or another system.

NOTE 4 Transaction flow testing (Beizer 1995) is often classified as a type of scenario testing.

5.2.9.1.2 Derive Test Coverage Items (TD3)

The test coverage items shall be the main and alternate scenarios (i.e. the test coverage items are the same as the test conditions). Therefore, no further action is required at this step for this technique.

5.2.9.1.3 Derive Test Cases (TD4)

Test cases for scenario testing shall be derived by covering each scenario (test coverage item) with at least one test case. The following steps shall be used during test case derivation:

1. Select test coverage item(s) to exercise in the current test case;
2. Identify input values to exercise the test coverage item(s) covered by the test case;
3. Determine the expected result of the test case by applying the input(s) to the test basis; and
4. Repeat steps 1 to 3 until the required level of test coverage of the test coverage item(s) is achieved.

5.2.9.2 Random TestingDerive Test Conditions (TD2)

Random testing (BS 7925-2:1998; Craig and Jaskiel 2002; Kaner 1988) uses a model of the input domain of the test item that defines the set of all possible input values. An input distribution for the generation of random input values shall be chosen. The domain of all possible inputs shall be the test condition for random testing.

EXAMPLE Types of input distributions include the normal distribution, uniform distribution and operational profile.

5.2.9.3 Derive Test Coverage Items (TD3)

There are no recognised test coverage items for random testing.

5.2.9.4 Derive Test Cases (TD4)

Test cases for random testing shall be chosen by randomly selecting input values from the input domain of the test item (or pseudo-randomly if using a tool) according to the chosen input distribution. The following steps shall be used during test case derivation:

1. Select an input distribution for the selection of test inputs;
2. Generate random values for the test inputs based on the input distribution chosen in step 1;
3. Determine the expected result of the test case by applying the input(s) to the test basis; and
4. Repeat steps 2 and 3 until the required testing has been completed.

NOTE 1 The required testing can be defined in terms of a number of tests executed, an amount of time spent testing or some other measure of completion.

NOTE 2 Step 2 is normally automated.

5.3 Structure-Based Test Design Techniques

5.3.1 Statement Testing

5.3.1.1 Derive Test Conditions (TD2)

A model of the source code of the test item which identifies statements as either executable or non-executable shall be derived (BS 7925-2:1998, Myers 1979). Each executable statement shall be a test condition.

NOTE The identification of non-executable statements could be carried out during step TD4, using an automated tool.

5.3.1.2 Derive Test Coverage Items (TD3)

Each executable statement shall be a test coverage items (i.e. the test coverage items are the same as the test conditions). Therefore, no further action is required at this step for this technique.

5.3.1.3 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing;
2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;
3. Determine the expected result from exercising the control flow sub-path(s) by applying the corresponding test inputs to the test basis; and
4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.3.2 Branch Testing

5.3.2.1 Derive Test Conditions (TD2)

A control flow model of the test item which identifies branches in the control flow of the test item shall be derived (BS 7925-2:1998, Myers 1979). Each branch in the control flow model shall be a test condition.

EXAMPLE "Branches" include arcs, links or edges in a graphical control flow model.

A branch is:

— a conditional transfer of control from any node in the control flow model to any other node; or

— an explicit unconditional transfer of control from any node to any other node in the control flow model; or

— when a test item has more than one entry point, a transfer of control to an entry point of the test item.

NOTE 1 A branch cannot link consecutive statements.

NOTE 2 Complete branch testing covering 100% of all branches requires *all* arcs (links or edges) in the control flow graph to be tested, including sequential statements between an entry and exit point that contains no decisions.

NOTE 3 Branch testing can require testing of both conditional and unconditional branches, including entry and exit points to a test item, depending on the level of test coverage required.

5.3.2.2 Derive Test Coverage Items (TD3)

Each branch in the control flow model shall be a test coverage item (i.e. the test coverage items are the same as the test conditions). Therefore, no further action is required at this step for this technique.

5.3.2.3 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing;
2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;
3. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and
4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

NOTE If there are no decisions in the test item then a single test case is still required.

5.3.3 Decision Testing

5.3.3.1 Derive Test Conditions (TD2)

A control flow model of the test item that identifies decisions shall be derived. Decisions are points in the test item where two or more possible outcomes (and hence sub-paths) may be taken by the control flow (BS 7925-2:1998, Myers 1979). Typical decisions are used for simple selections (e.g. if-then-else in source code), to decide when to exit loops (e.g. while-loop in source code), and in case (switch) statements (e.g. case-1-2-3-...-N in source code). In decision testing, each decision in the control flow model shall be a test condition.

NOTE If there are no decisions in the test item then a single test condition, test coverage item and test case is still required.

5.3.3.2 Derive Test Coverage Items (TD3)

The decision outcomes from each decision shall be identified as test coverage items.

5.3.3.3 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) that reach one or more test coverage items that have not yet been executed during testing;

2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;

3. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.3.4 Branch Condition Testing

5.3.4.1 Derive Test Conditions (TD2)

A control flow model of the test item that identifies decisions and conditions within them shall be derived. In branch condition testing (BS 7925-2:1998, Myers 1979), each decision shall be a test condition.

EXAMPLE In program source code the decision statement “if A OR B AND C then” is a test condition that contains 3 conditions related by logical operators.

5.3.4.2 Derive Test Coverage Items (TD3)

In branch condition testing, all Boolean values (true/false) of the condition(s) within decisions shall be identified as test coverage items.

5.3.4.3 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) that cover one or more test coverage items that have not yet been executed during testing;

2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;

3. Identify a sub-set of test inputs from step 2 to cover the Boolean values of conditions within the decision;

4. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and

5. Repeat steps 1 to 4 until the required level of test coverage is achieved.

5.3.5 Branch Condition Combination Testing

5.3.5.1 Derive Test Conditions (TD2)

A control flow model of the test item which identifies decisions and conditions shall be derived. In branch condition combination testing (BS 7925-2:1998, Myers 1979), each decision shall be a test condition.

EXAMPLE In program source code the decision statement "if A OR B AND C then" is a test condition that contains 3 conditions related by logical operators.

5.3.5.2 Derive Test Coverage Items (TD3)

Each unique feasible combination of Boolean values of conditions within each decision shall be identified as a test coverage item (BS 7925-2:1998, Myers 1979). This includes simple decisions, where combinations consist of two individual Boolean outcomes of a single condition within a decision.

5.3.5.3 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) that reach one or more test coverage items that have not yet been executed during testing;

2. Determine the set of test inputs that will cause the identified control flow sub-path(s) to be exercised;

3. Identify a sub-set of test inputs from step 2 to cover the selected combination of Boolean values of conditions within the decision;

4. Determine the expected result by applying the selected test inputs to the test basis; and

5. Repeat steps 1 to 4 until the required level of test coverage is achieved.

NOTE If there are no decisions in the test item then a single test case is still required.

5.3.6 Modified Condition Decision Coverage (MCDC) Testing

5.3.6.1 Derive Test Conditions (TD2)

A control flow model of the test item that identifies decisions and conditions shall be derived. In modified condition decision coverage (MCDC) testing (BS 7925-2:1998, Myers 1979), each decision shall be a test condition.

EXAMPLE In program source code the decision statement "if A OR B AND C then" is a test condition that contains 3 conditions related by logical operators.

5.3.6.2 Derive Test Coverage Items (TD3)

Each unique feasible combination of individual Boolean values of conditions within a decision that allows a single Boolean condition to independently affect the outcome shall be identified as a test coverage item.

NOTE This includes simple decisions, where combinations consist of two individual Boolean outcomes of a single condition within a decision.

5.3.6.3 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) that reach one or more test coverage items that have not yet been executed during testing;

2. Determine the set of test inputs that will cause the identified control flow sub-path(s) to be exercised;

3. Identify a sub-set of test inputs from step 2 to cover the selected combinations of individual Boolean values of conditions within the decision that allow a single Boolean condition to independently affect the outcome;

4. Determine the expected result by applying the selected test inputs to the test basis; and

5. Repeat steps 1 to 4 until the required level of test coverage is achieved.

NOTE If there are no decisions in the test item then a single test case is still required.

5.3.7 Data Flow Testing

5.3.7.1 Derive Test Conditions (TD2)

In data flow testing (Burnstein 2003), a model of the test item that identifies control flow sub-paths through the test item that link the definition and subsequent use(s) of individual variables shall be derived.

“Definitions” are where a variable is possibly given a new value (sometimes a definition will result in a variable retaining the same value as it had before). The entry point of a test item shall be considered to have a definition of each parameter, each non-local variable, and the input buffer (which may implicitly appear in calls to procedures or functions). A “use” is an occurrence of a variable in which the variable is not given a new value; “uses” can be further distinguished as either “p-uses” (predicate-use) or “c-uses” (computation-use). A p-use denotes the use of a variable in determining the outcome of a condition (predicate) within a decision, such as a while-loop, if- then- else, etc. A c-use occurs when a variable is used as an input to the computation of the definition of any variable or of an output. The exit point of the test item is considered to have a c-use of each parameter, each non-local variable, and the input buffer.

In data flow testing, each definition-use pair for a variable in the test item is a test condition.

There are a number of forms of data flow testing, which are all based on the same test conditions. The five forms defined in ISO/IEC 29119-4 are: all-definitions testing, all-c-uses testing, all-p-uses testing, all-uses testing, and all-du-paths testing.

5.3.7.2 All-Definitions Testing

5.3.7.2.1 Derive Test Coverage Items (TD3)

The control flow sub-paths from each variable definition to some use (either p-use or c-use) of that definition shall be identified as test coverage items. Each sub-path is known as a “definition-use” path. “All-definitions” requires that at least one definition-free sub-path (with relation to a specific variable) from the definition to one of its c-uses or p-uses will have been covered for all variable definitions.

5.3.7.2.2 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify definitions that have not yet been executed during testing;

2. Determine the test inputs that will cause the control flow sub-path(s) from the identified definitions to be exercised;

3. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

NOTE In practice, these steps could require automation.

5.3.7.3 All-C-Uses Testing

5.3.7.3.1 Derive Test Coverage Items (TD3)

The control flow sub-paths from each variable definition to each c-use of that definition shall be identified as test coverage items. “All-C-uses” requires that at least one definition-free sub-path (with relation to a specific variable) from the definition to one of its c-uses will have been covered for all variable definitions.

5.3.7.3.2 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) from variable definitions to subsequent c-use of that definition that have not yet been executed during testing;

2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;

3. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.3.7.4 All-P-Uses Testing

5.3.7.4.1 Derive Test Coverage Items (TD3)

The control flow sub-paths from each variable definition to each p-use of that definition shall be identified as test coverage items. “All-P-uses” requires that at least one definition-free sub-path (with relation to a specific variable) from the definition to one of its p-uses will have been covered for all variable definitions.

5.3.7.4.2 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) from variable definitions to subsequent p-use of that definition that have not yet been executed during testing;

2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;

3. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.3.7.5 All-Uses Testing

5.3.7.5.1 Derive Test Coverage Items (TD3)

The control flow sub-paths from each variable definition to every use (both p-use and c-use) of that definition shall be identified as test coverage items. “All-Uses” requires that at least one definition-free sub-path (with relation to a specific variable) from the definition to each of its uses will have been covered for all variable definitions.

5.3.7.5.2 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) from variable definitions to a subsequent p-use or c-use of that definition that have not yet been executed during testing;

2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;

3. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.3.7.6 All-DU-Paths Testing

5.3.7.6.1 Derive Test Coverage Items (TD3)

The control flow sub-paths from each variable definition to every use (both p-use and c-use) of that definition shall be identified as test coverage items. "All-DU-Paths" requires that all definition-free sub-paths (with relation to a specific variable) from the definition to each of its uses will have been covered for all variable definitions.

NOTE All-DU-Paths testing requires *all* loop-free sub-paths from a variable definition to its use be tested to attempt to achieve 100% test item coverage. This differs from All-Uses testing, which requires only *one* path from each variable definition to its use to be tested to attempt to achieve 100% test item coverage.

5.3.7.6.2 Derive Test Cases (TD4)

The following steps shall be followed during test case derivation:

1. Identify control flow sub-path(s) from variable definitions to every subsequent p-use and c-use of that definition that have not yet been executed during testing;

2. Determine the test inputs that will cause the control flow sub-path(s) to be exercised;

3. Determine the expected result from exercising the control flow sub-path(s) by applying the test inputs to the test basis; and

4. Repeat steps 1 to 3 until the required level of test coverage is achieved.

5.4 Experience-Based Testing

5.4.1 Error Guessing

5.4.1.1 Derive Test Conditions (TD2)

Error guessing (BS 7925-2:1998, Myers 1979) involves the design of a checklist of defect types that may exist in the test item, allowing the tester to identify inputs to the test item that may cause failures, if those defects exist in the test item. Each defect type shall be a test condition.

NOTE The checklist of defect types could be derived by various means, such as taxonomies of known errors, information contained in incident management systems, from a tester's knowledge, experience and/or understanding of the test item(s) and/or similar test items or from the knowledge of other stakeholders (e.g. system users or programmers).

5.4.1.2 Derive Test Coverage Items (TD3)

There are no recognised test coverage items for error guessing.

5.4.1.3 Derive Test Cases (TD4)

Test cases for error guessing are typically derived by selecting a defect type from a checklist of defect types that are to be covered and deriving test cases that could detect that defect type in the test item, if it existed. The following steps shall be used during test case derivation:

1. Select a defect type(s) for coverage by the current test case;

2. Identify input values that could be expected to cause a failure corresponding to the selected defect type(s);

3. Determine the expected result of the test case by applying the input(s) to the test basis; and

4. Repeat steps 1 to 3 until the required testing has been completed.

6 Test Coverage Measurement

6.1 Test Measurement for Specification-Based Test Design Techniques

6.1.1 Overview

The coverage measures defined in ISO/IEC 29119-4 are based on differing degrees of coverage that are achievable by test design techniques. Coverage levels can range from 0% to 100%. In each coverage calculation, a number of test coverage items may be infeasible. A test coverage item shall be defined to be infeasible if it can be shown to not be executable or impossible to be covered by a test case. The coverage calculation shall be defined as either counting or discounting infeasible items; this choice will typically be recorded in the Test Plan (defined in ISO/IEC 29119-3). If a test coverage item is discounted, justification for its infeasibility will typically be recorded in a test report. In each coverage calculation, if there are no test coverage items of a given type in a test item, 100% coverage for that type of coverage will be defined as being inapplicable for that test item.

When calculating coverage for any technique, the following formula shall be used:

$$Coverage = \left(\frac{\mathrm{N}}{\mathrm{T}} \times 100 \right) \%$$

where:

— Coverage is the coverage achieved by a specific test design technique;

— N is the number of test coverage items covered by executed test cases;

— T is the total number of test coverage items identified.

Specific values for Coverage, N and T for each technique are defined in the clauses below. The set of coverage measures presented in the following clauses are designed to be used with the test design techniques presented in this standard. They are not designed to be an exhaustive list; there may be other coverage measures that are used by organizations that are not mentioned in this clause.

EXAMPLE Other measures that could be required for assessing the completeness of testing include measuring the overall percentage of requirements that have been covered during testing.

6.1.2 Equivalence Partition Coverage

Coverage for equivalence partitioning shall be calculated using the following definitions:

— Coverage is the equivalence partition coverage;

— N is the number of partitions covered by executed test cases;

— T is the total number of partitions identified.

6.1.3 Classification Tree Method Coverage

Coverage for the classification tree method shall be calculated using the following definitions.

For minimal coverage, the following definitions shall be used:

— Coverage is the minimal coverage classification tree method coverage;

— N is the number of classes covered by executed test cases;

— T is the total number of classes.

For maximal coverage the following definitions shall be used:

— Coverage is the maximal classification tree method coverage;

— N is the number of combinations of classes covered by executed test cases;

— T is the total number of combinations of classes.

6.1.4 Boundary Value Analysis Coverage

Coverage for boundary value analysis shall be calculated using the following definitions:

— Coverage is the boundary value coverage;

— N is the number of distinct boundary values covered by executed test cases;

— T is the total number of boundary values.

The decision to apply two-value or three-value boundary testing should be recorded.

6.1.5 Syntax Testing Coverage

There is currently no industry agreed approach to calculating coverage for syntax testing.

6.1.6 Combinatorial Test Design Technique Coverage

6.1.6.1 All Combinations Testing Coverage

Coverage for all combinations testing shall be calculated using the following definitions:

— Coverage is all combinations coverage;

— N is the number of unique combinations of P-V pairs covered by executed test cases;

— T is the total number of unique P-V pair combinations (the product of the number of P-V pairs for each test item parameter).

NOTE For a definition of P-V pairs, refer to clause 4.23.

6.1.6.2 Pair-wise Testing Coverage

Coverage for pair-wise testing shall be calculated using the following definitions:

— Coverage is pair-wise coverage;

— N is the number of unique pairs of P-V pairs covered by executed test cases;

— T is the total number of unique P-V pairs.

6.1.6.3 Each Choice Testing Coverage

Coverage for each choice testing shall be calculated using the following definitions:

— Coverage is each choice coverage;

— N is the number of P-V pairs covered by executed test cases;

— T is the total number of unique P-V pairs.

6.1.6.4 Base Choice Testing Coverage

Coverage for base choice testing shall be calculated using the following definitions:

— Coverage is base choice coverage;

— N is the number of base choice combinations covered by executed test cases (all but one test item parameter set to the base value, and the remaining test item parameter set to valid values), plus one (for when all test item parameters are set to the base value) if exercised;

— T is the total number of base choice combinations (all but one test item parameter set to the base value, and the remaining test item parameter set to valid values), plus one (for when all test item parameters are set to the base value).

6.1.7 Decision Table Testing Coverage

Coverage for decision table testing shall be calculated using the following formula:

— Coverage is decision table coverage;

— N is the number of decision rules covered by executed test cases;

— T is the total number of feasible decision rules.

6.1.8 Cause-Effect Graphing Coverage

Coverage for cause-effect graphing shall be calculated using the following definitions:

— Coverage is the cause/effect coverage;

— N is the number of feasible decision rules covered by executed test cases;

— T is the total number of feasible decision rules.

6.1.9 State Transition Testing Coverage

Coverage for all states testing shall be calculated using the following definitions:

- Coverage is all states coverage;
- N is the number of states covered by executed test cases;
- T is the total number of states.

Coverage for single transitions (0-switch coverage) shall be calculated using the following definitions:

- Coverage is 0-switch coverage;
- N is the number of single valid transitions executed by test cases;
- T is the total number of single valid transitions.

Coverage for all transitions shall be calculated using the following definitions:

- Coverage is all transitions transition coverage;
- N is the number of valid and invalid transitions attempted to be covered by executed test cases;
- T is the total number of valid and invalid transitions between identified states initiated by valid events.

Coverage for N+1 transitions (N switch testing) shall be calculated using the following definitions:

- Coverage is N-switch coverage;
- N is the number of N+1 valid transitions covered by executed test cases;
- T is the total number of sequences of N+1 valid transitions.

6.1.10 Scenario Testing Coverage

Coverage for scenario testing (including use case testing) shall be calculated using the following definitions:

- Coverage is scenario coverage;
- N is the number of scenarios covered by executed test cases;
- T is the total number of scenarios.

6.1.11 Random Testing Coverage

There is currently no industry agreed approach to calculating coverage for random testing.

6.2 Test Measurement for Structure-Based Test Design Techniques

6.2.1 Statement Testing Coverage

Coverage for statement testing shall be calculated using the following definitions:

- Coverage is statement coverage;

— N is the number of executable statements covered by executed test cases;

— T is the total number of executable statements.

6.2.2 Branch Testing Coverage

Coverage for branch testing shall be calculated using the following definitions:

— Coverage is branch coverage;

— N is the number of branches covered by executed test cases;

— T is the total number of branches.

NOTE For situations where there are no branches in the test item a single test is required to achieve 100% branch coverage.

6.2.3 Decision Testing Coverage

Coverage for decision testing shall be calculated using the following definitions:

— Coverage is decision coverage;

— N is the number of decision outcomes covered by executed test cases;

— T is the total number of decision outcomes.

NOTE For situations where there are no decisions in the test item a single test is required to achieve 100% decision coverage.

6.2.4 Branch Condition Testing Coverage

Coverage for branch condition testing shall be calculated using the following definitions:

— Coverage is branch condition coverage;

— N is the number of Boolean values of conditions within decisions covered by executed test cases;

— T is the total number of Boolean values of conditions within decisions.

6.2.5 Branch Condition Combination Testing Coverage

Coverage for branch condition combination testing shall be calculated using the following definitions:

— Coverage is branch condition combination coverage;

— N is the number of combinations of Boolean values of conditions within each decision covered by executed test cases;

— T is the total number of unique combinations of Boolean values of conditions within decisions.

NOTE For situations where there are no decisions in the test item a single test is required to achieve 100% branch condition combination coverage.

6.2.6 Modified Condition Decision (MCDC) Testing Coverage

Coverage for modified condition decision coverage testing shall be calculated using the following definitions:

— Coverage is modified condition decision coverage;

— N is the number of unique feasible combinations of individual Boolean values of conditions within decisions that allow a single Boolean condition to independently affect the decision outcome to be covered by executed test cases;

— T is the total number of unique combinations of individual Boolean values of conditions within decisions that allow a single Boolean condition to independently affect the outcome.

NOTE For situations where there are no decisions in the test item a single test is required to achieve 100% modified condition decision coverage.

6.2.7 Data Flow Testing Coverage

6.2.7.1 All-definitions Testing Coverage

Coverage for all-definitions testing shall be calculated using the following definitions:

— Coverage is all-definitions coverage;

— N is the number of definition-use pairs from distinct variable definitions covered by executed test cases;

— T is the total number of definitions-use pairs.

6.2.7.2 All-c-uses Testing Coverage

Coverage for all-c-uses testing shall be calculated using the following definitions:

— Coverage is all-c-uses coverage;

— N is the number of definition-c-use pairs covered by executed test cases;

— T is the total number of definition-c-use pairs.

6.2.7.3 All-p-uses Testing Coverage

Coverage for all-p-uses testing shall be calculated using the following definitions:

— Coverage is all-p-uses coverage;

— N is the number of definition-p-use pairs covered by executed test cases;

— T is the total number of definition-p-use pairs.

6.2.7.4 All-uses Testing Coverage

Coverage for all-uses testing shall be calculated using the following definitions:

— Coverage is all-uses coverage;

— N is the number of unique definition-use pairs from a definition to both p-use *and* c-use of that definition covered by executed test cases;

— T is the total number of definition-use pairs from each definition to both p-use *and* c-use of that definition.

6.2.7.5 All-du-paths Testing Coverage

Coverage for all-du-paths testing shall be calculated using the following definitions:

— Coverage is all-du-paths coverage;

— N is the number of unique sub-paths from each definition to all p-uses *and* all c-uses of that definition covered by executed test cases;

— T is the total number of simple sub-paths from each definition to both p-use *and* c-use of that definition.

6.3 Test Measurement for Experience-Based Testing Design Techniques

6.3.1 Error Guessing Coverage

There is currently no industry agreed approach to calculating coverage for error guessing.

Annex A
(informative)

Testing Quality Characteristics

A.1 Quality Characteristics

A.1.1 Overview

Software testing can be carried out to collect evidence as to whether required quality criteria have been satisfied by a test item. This informative annex contains examples of how quality characteristics defined in ISO/IEC 25010 (ISO/IEC 25010:2011) can be mapped to the test design techniques defined in ISO/IEC 29119-4. The test design techniques defined in this standard could be used for testing a variety of these characteristics. There may be other quality characteristics that could be considered (e.g. privacy).

ISO/IEC 25010 defines the product quality model shown below in Figure 3, which categorises system/software product quality properties into eight characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. Each characteristic is composed of a set of related sub-characteristics. In some situations, there may be regulatory requirements (e.g. government policies or laws) that mandate that certain quality characteristics are met by a system. Various test design techniques and types of testing can be used to test for each characteristic (see clauses A.3 and A.4).

Clause A.2 provides an explanation of types of testing that can be used to test the quality characteristics presented in Figure 3. A mapping of each quality characteristic to applicable types of testing is presented in clause A.3. The relationship between the quality characteristics and the specification-based and structure-based test design techniques covered in ISO/IEC 29119-4 is explained in clause A.4.

NOTE ISO/IEC 25030 (ISO/IEC 25030:2007) can be used to identify and document software quality requirements that are applicable to a test item. These can then be used to identify quality characteristics in ISO/IEC 25010 (ISO/IEC 25010:2011) and corresponding types of testing that apply to testing each quality requirement. ISO/IEC 25020, ISO/IEC 25021, ISO/IEC 25022, ISO/IEC 25023 and ISO/IEC 25024 can then be used for identifying and defining quality measure elements and quality measures that can be used during testing to produce quantifiable testing results.

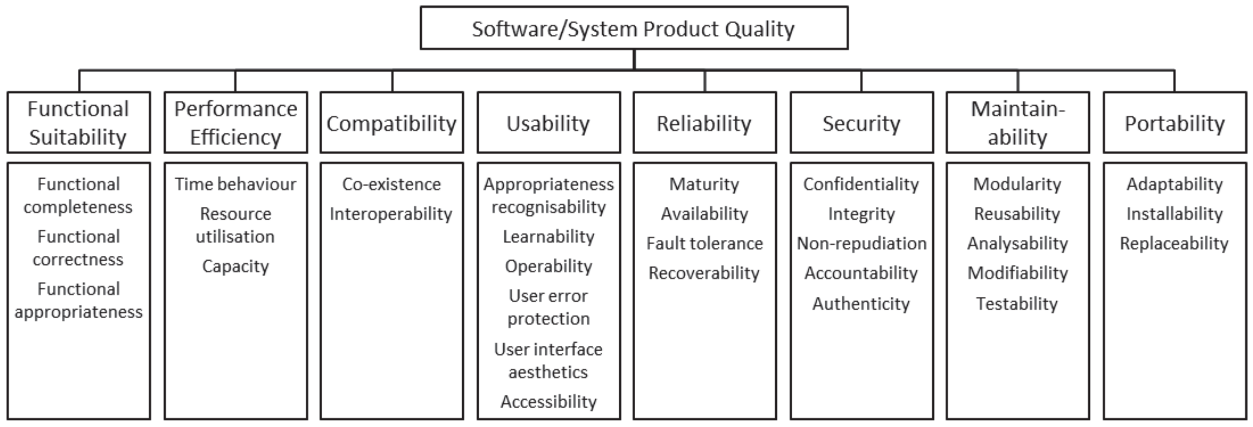


Figure 3 – ISO/IEC 25010 product quality model

A.2 Quality-Related Types of Testing

A.2.1 Accessibility Testing

The purpose of accessibility testing is to determine whether the test item can be operated by users with the widest range of characteristics and capabilities (ISO/IEC 25010:2011), including specific accessibility requirements (e.g. due to age, visual impairment or hearing impairment). Accessibility testing uses a model of the test item that specifies its accessibility requirements, including any accessibility design standards to which the test item must conform. Accessibility requirements are concerned with the ability of a user with specific accessibility needs to achieve accessibility objectives. For example, this could include a requirement for the test item to support visual and/or hearing impaired users.

NOTE The World Wide Web Consortium (W3C) defines standards for accessibility, including accessibility of web applications and devices. Visit <http://www.w3.org/standards/> for more information.

A.2.2 Backup/Recovery Testing

The purpose of backup/recovery testing is to determine if, in the event of failure, a test item can be restored from backup to its pre-failure state. Backup/recovery testing uses a model of the test item that specifies its backup and recovery requirements, which specify the need to back up the operational state of a test item at a point in time, including data, configuration and/or environment and restore the state of the test item from that backup. Backup/recovery testing then focusses on testing the correctness of the test item's backup and the correctness of the restored state of the test item against its pre-failure state. Backup/recovery testing can also be used to verify whether the backup and recovery procedures for the test item achieve specified recovery objectives. This type of testing may be carried out as part of a disaster recovery test (see clause A.2.5).

A.2.3 Compatibility Testing

The purpose of compatibility testing is to determine whether the test item can function alongside other independent or dependent (but not necessarily communicating) products in a shared environment (i.e. co-existence). Compatibility testing may also be applied to multiple copies of the same test item or to multiple test items sharing a common environment.

Compatibility requirements for test items typically include one or more of the following sub-requirements:

— Order of installation. Explicit order(s) of installation (otherwise it should be assumed that all possible orders of installation are valid) results in a configuration where each test item will subsequently perform its required functions correctly.

— Order of instantiation. Explicit order(s) of instantiation (otherwise it should be assumed that all possible orders of instantiation are valid) result in a run-time configuration where each test item will subsequently perform its required functions correctly.

— Concurrent use. The ability of two or more test items to perform their required functions while running (but not necessarily communicating) in the same environment.

— Environment constraints. Features of the environment, such as memory, processor, architecture, platform or configuration, that may affect the ability of the test item to perform its required functions correctly.

A.2.4 Conversion Testing

The purpose of conversion testing is to determine whether data or software can continue to provide required capabilities after modifications are made to their format, such as converting a program from one programming language to another or converting a flat data file or database from one format to another. Conversion testing uses a model of the test item that specifies its conversion requirements, including those that must remain invariant through the conversion process, those that are new, modified, or obsoleted by the conversion and any required conversion design standards to which the test item must conform.

A.2.5 Disaster Recovery Testing

The purpose of disaster recovery testing is to determine if, in the event of failure, operation of the test item can be transferred to a different operating site and whether it can be transferred back again once the failure has been resolved. Disaster recovery testing uses a model of the test item (typically a disaster recovery plan) that specifies its disaster recovery requirements, including any required disaster recovery design standards to which the test item must conform. The test item in disaster recovery testing may be an entire operational system, with associated facilities, personnel, and procedures. Disaster recovery testing may cover factors such as procedures to be carried out by operational staff, relocation of data, software, personnel, offices, or other facilities, or recovering data previously backed up to a remote location.

A.2.6 Functional Testing

The purpose of functional testing is to determine whether the functional requirements of the test item have been met. For example, this could include identifying whether a function had been implemented according to its specified requirements. It could be carried out using the specification-based and structure-based test design techniques that are specified in clause 5.

A.2.7 Installability Testing

The purpose of installability testing is to determine if a test item(s) can be installed as required in all specified environments. Installability testing uses a model of the installability requirements of the test item, which are typically specified in terms of the installation process (as described in the installation manual or guidelines), the people who will carry out the installation, the target platform(s) and the test item(s) to be installed.

A.2.8 Interoperability Testing

The purpose of interoperability testing is to determine if a test item can interact correctly with other test items or systems either in the same environment or different environments, including whether the test item can make effective use of information received from other systems. Interoperability testing uses a model of the test item that specifies its interoperability requirements, including interoperability design standards to which the test item must conform. This could include assessing whether a test item running in one environment can interact accurately with another test item or system in another separate environment.

A.2.9 Localization Testing

The purpose of localization testing is to determine whether the test item can be understood within the geographical region it is required to be used in. Localization testing can include (but is not limited to) analysis of whether the user interface and supporting documentation of the test item can be understood by users within each country or region of use.

A.2.10 Maintainability Testing

The purpose of maintainability testing is to determine if a test item can be maintained by using an acceptable amount of effort. Maintainability testing uses a model of the maintainability requirements of the test item, which are typically specified in terms of the effort required to effect a change under the following categories:

— corrective maintenance (i.e. correcting problems);

— perfective maintenance (i.e. enhancements);

— adaptive maintenance (i.e. adapting to changes in environment); and

— preventive maintenance (i.e. actions to reduce future maintenance costs).

A.2.11 Performance-Related Testing

The purpose of this family of techniques is to determine whether a test item performs as required when it is placed under various types and sizes of “load”. This includes performance, load, stress, endurance, volume, capacity and memory management testing, which each use a model of the test item that specifies its performance requirements, including any required performance design standards to which the test item must conform. For example, this may include assessing the performance of the test item in terms of transactions per second, throughput response times, round trip time and resource utilization levels. The "typical" load of the test item under "normal" conditions may be defined in the operational profile of the test item.

There are numerous techniques for assessing the performance of the test item:

— Performance testing is aimed at assessing the performance of the test item when it is placed under a "typical" load.

— Load testing is aimed at assessing the behaviour of the test item (e.g. performance, reliability and stability) when it is placed under conditions of varying loads, usually between anticipated conditions of low, typical and peak usage.

— Stress testing is aimed at assessing the performance of the test item when it is pushed beyond its anticipated peak load or when available resources (e.g. memory, processor, disk) are reduced below specified minimum requirements, to evaluate how it behaves under extreme conditions.

— Endurance testing (also called soak testing) is aimed at assessing whether the test item can sustain the required load for a continuous period of time.

— Volume testing is aimed at assessing the performance of the test item when it is processing specified levels of data. For example, this may include assessing test item performance when its database is nearing maximum capacity.

— Capacity testing (also called scalability testing) is aimed at assessing how the test item will perform under conditions that may need to be supported in the future. For example, this may include assessing what level of additional resources (e.g. memory, disk capacity, network bandwidth) will be required to support anticipated future loads.

— Memory management testing is aimed at assessing how the test item will perform in terms of the amount (normally maximum) of memory used (e.g. hard disk memory, RAM and ROM), the type of memory (e.g. dynamic or allocated/static) and/or defined levels of memory leakage experienced during testing. Memory requirements will typically be specified in terms of specific operating conditions (e.g. a peak memory requirement over a particular period of operation under defined transaction loads may be specified).

A.2.12 Portability Testing

The purpose of portability testing is to determine the degree of ease or difficulty to which a test item can be effectively and efficiently transferred from one hardware, software or other operational or usage environment to another. Portability testing uses a model of the test item that specifies its portability requirements, including any required portability design standards to which the test item must conform. Portability requirements are concerned with the ability to transfer the test item from one environment to another, or to alter the configuration of the existing environment to other required configurations. For example, this could include assessing whether the test item can be operated from a variety of different browsers.

A.2.13 Procedure Testing

The purpose of procedure testing is to determine whether procedural instructions meet user requirements and support the purpose of their use. Procedure testing uses a model of the procedural requirements of the test item as a complete and delivered unit. Procedure requirements define what is expected of any procedural documentation and are written in the form of procedural instructions. These procedural instructions will normally come in the form of one of the following documents:

— a user guide;

— an instruction manual;

— a user reference manual.

This information will normally define how the user is meant to:

— set up the test item for normal usage;

— operate the test item in normal conditions;

— become a competent user of the system (tutorial files);

— trouble-shoot the test item when faults arise;

— re-configure the test item.

A.2.14 Reliability Testing

The purpose of reliability testing is to evaluate the ability of the test item to perform its required functions, including evaluating the frequency with which failures occur, when it is used under stated conditions for a specified period of time. Reliability testing uses a model of the test item that specifies its required level of reliability (e.g. mean time to failure, mean time between failure). The model should include a definition of failure and either the operational profile of the test item or an approach to derive the operational profile.

A.2.15 Security Testing

The purpose of security testing is to evaluate the degree to which a test item and its associated data are protected so that unauthorized persons or systems cannot use, read or modify them and authorized persons or systems are granted required access to them. Security testing uses a model of the test item that specifies its security requirements, including any required security design standards to which the test item must conform. Security requirements are concerned with the ability to protect the data and functionality of a test item from unauthorised users and malicious use. For example, this could include assessing whether the test item prevents unauthorised users from accessing data, or whether certain functions of a test item that are only required to be accessible by certain user types are protected from other user types.

There are a number of techniques for assessing the security of a test item:

— Penetration testing involves attempted access to a test item (including its functionality and/or private data) by a tester that is mimicking the actions of an unauthorised user.

— Privacy testing involves attempted access to private data and verification of the audit trail (i.e. trace) that is left behind when users access private data.

— Security auditing is a type of static testing in which a tester inspects, reviews or walks through the requirements and code of a test item to determine whether any security vulnerabilities are present.

— Vulnerability scanning involves the use of automated testing tools to scan a test item for signs of specific known vulnerabilities.

A.2.16 Stability Testing

The purpose of stability testing is to determine whether the test item continues to function correctly over the lifespan of the product. Stability testing uses a model of the stability requirements of the test item, which are typically specified in terms of the capability of the test item to avoid unexpected and/or unwanted effects after modification. For example, this could include an assessment of the number of new defects (i.e. incidents) that are introduced during the repair of existing defects.

A.2.17 Usability Testing

The purpose of usability testing is to evaluate whether specified users can use the test item to achieve assigned goals with effectiveness, efficiency and satisfaction in specified contexts of use. Usability testing therefore uses a model of the test item that specifies its usability requirements, including any required usability design standards to which the test item must conform. Usability requirements specify the usability goals for the test item. Usability goals must be based on test item goals (the reason for having the test item, the difference it is to bring about for the organization or individual, the usability-related purpose and tasks it will aid), and the contexts of use for the test item (who is to use the test item and the environment in which it is to be used, user characteristics and user tasks). Usability goals will be defined for the effectiveness, efficiency and satisfaction for specified users to achieve specified goals in one or more specified contexts of use.

NOTE ISO/IEC 9241 defines standards for defining the requirements for human-systems interaction.

A.3 Mapping Quality Characteristics to Types of Testing

In the table below, the types of testing that were presented in clause A.2 are mapped to the quality characteristics that were presented in Figure 3 (in clause A.1.1).

Table 1 – Mapping of ISO/IEC 25010 product quality characteristics to types of testing

Type of Testing	Quality Characteristic	Sub-Characteristics
Accessibility Testing	Usability	Accessibility
Backup/Recovery Testing	Reliability	Maturity
		Fault tolerance
		Recoverability
Compatibility Testing	Compatibility	Co-existence
Conversion Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Disaster Recovery Testing	Reliability	Maturity
		Fault tolerance
		Recoverability
Functional Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Installability Testing	Portability	Installability
Interoperability Testing	Compatibility	Interoperability
Maintainability Testing	Maintainability	Modularity
		Reusability
		Analysability
		Modifiability
		Testability
Performance-Related Testing	Performance efficiency	Time-behaviour
		Resource utilisation
		Capacity
Portability Testing	Portability	Adaptability
		Installability

Type of Testing	Quality Characteristic	Sub-Characteristics
		Replaceability
Procedure Testing	None	None
Reliability Testing	Reliability	Maturity
		Availability
		Fault tolerance
		Recoverability
Security Testing	Security	Confidentiality
		Integrity
		Non-repudiation
		Accountability
		Authenticity
Stability Testing	Maintainability	Modifiability
Usability Testing	Usability	Appropriateness recognisability
		Learnability
		Operability
		User error protection
		User interface aesthetics
		Accessibility

A.4 Mapping Quality Characteristics to Test Design Techniques

A.4.1 Mapping

The test design techniques described in ISO/IEC 29119-4 can be used to test a variety of the quality characteristics listed in Figure 3. The following table provides an example mapping between them.

Table 2 – Mapping of test design techniques to product quality measures for ISO/IEC 25010 product characteristics

Test Design Technique	ISO/IEC 2510 Quality Characteristic	ISO/IEC 2510 Sub-Characteristics
Specification-Based Test Design Techniques		
Boundary Value Analysis	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Performance Efficiency	Time-behaviour
		Capacity
	Usability	User error protection
	Reliability	Fault tolerance
	Security	Confidentiality
		Integrity
Cause-Effect Graphing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Usability	User error protection
	Compatibility	Co-existence
Classification Tree Method	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Usability	User error protection
Combinatorial Test Design Techniques	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Compatibility	Co-existence
	Performance Efficiency	Time-behaviour

Test Design Technique	ISO/IEC 2510 Quality Characteristic	ISO/IEC 2510 Sub-Characteristics
	Usability	User error protection
Decision Table Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Compatibility	Co-existence
	Usability	User error protection
Equivalence Partitioning	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Usability	User error protection
	Reliability	Availability
	Security	Confidentiality
		Integrity
		Non-repudiation
		Accountability
		Authenticity
Random Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Performance	Time behaviour
		Resource utilisation
		Capacity
	Reliability	Maturity
		Availability
		Fault tolerance
		Recoverability
	Security	Confidentiality
		Integrity
		Non-repudiation
		Accountability
		Authenticity
Scenario Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Usability	Learnability
		Operability
		User error protection
		User interface aesthetics
		Accessibility
		Appropriateness recognisability
State Transition Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Reliability	Maturity
		Availability
		Fault tolerance
		Recoverability
Syntax Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Use Case Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Usability	Learnability

Test Design Technique	ISO/IEC 2510 Quality Characteristic	ISO/IEC 2510 Sub-Characteristics
		Operability
		User error protection
		User interface aesthetics
		Accessibility
		Appropriateness recognisability
Structure-Based Test Design Techniques		
Branch Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Decision Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Branch Condition Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Branch Condition Combination Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Data Flow Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Modified Condition Decision Coverage (MCDC) Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Statement Testing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Experience-Based Test Design Techniques		
Error Guessing	Functional Suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Performance efficiency	Time-behaviour
		Resource utilisation
		Capacity
	Usability	Learnability
		Operability
		User error protection
	Reliability	Fault tolerance

Annex B (informative)

Guidelines and Examples for the Application of Specification-Based Test Design Techniques

B.1 Guidelines and Examples for Specification-Based Testing

B.1.1 Overview

This annex provides guidance on the requirements in clauses 5.2 and 6.1 by demonstrating the application of each individual specification-based test design technique to a separate problem. Each example follows the Test Design and Implementation Process that is defined in ISO/IEC 29119-2. The test basis in each example is written in italics, such as the test basis for “*generate_grading*” in clause B.2.1.2. Although each example is applied in a specification-based testing context, as stated in clause 5.1, in practice most of the techniques defined in ISO/IEC 29119-4 can be used interchangeably (e.g. boundary value analysis could be used to test the inputs to a program through the user interface or the boundaries of variables within program source code).

B.2 Specification-Based Test Design Technique Examples

B.2.1 Equivalence Partitioning

B.2.1.1 Introduction

The aim of equivalence partitioning is to derive a set of test cases that cover the input and output partitions of the test item according to the chosen level of equivalence partition coverage. Equivalence partitioning is based on the premise that the inputs and outputs of a test item can be partitioned into classes that, according to the test basis for the test item, will be treated similarly by the test item. Thus the result of testing a single value from an equivalence partition is considered representative of any other value in the partition.

B.2.1.2 Example

Consider a test item, *generate_grading,* with the following test basis:

The component receives as input an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it outputs a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark, which is 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.

B.2.1.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: generate_grading function

B.2.1.4 Step 2: Derive Test Conditions (TD2)

In equivalence partitioning, the equivalence partitions are test conditions (TCOND). Equivalence partitions are identified from both the inputs and outputs of the test item. Valid and invalid inputs and outputs are considered.

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

TCOND1:	0 <= exam mark <= 75	(for FS1)
TCOND2:	0 <= coursework mark <= 25	(for FS1)

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

TCOND3:	exam mark < 0	(for FS1)
TCOND4:	exam mark > 75	(for FS1)
TCOND5:	coursework mark < 0	(for FS1)
TCOND6:	coursework mark > 25	(for FS1)

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

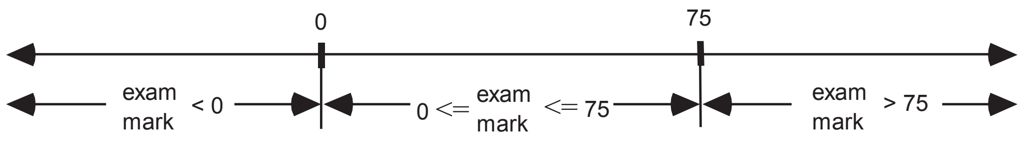


Figure 4 — Input "exam mark"

And for the input, coursework mark, we get:

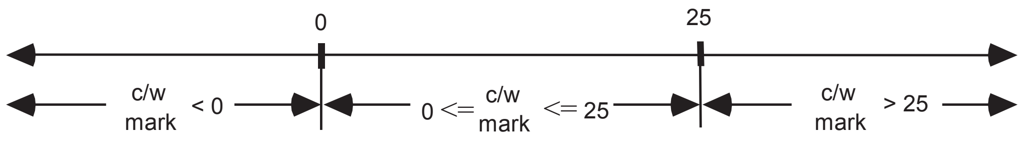


Figure 5 — Input "coursework mark"

Less obvious invalid input partitions could include any other input types, such as non-integer inputs and non-numeric inputs. So, we could generate the following invalid input equivalence partitions:

TCOND7:	exam mark = real number (a number with a fractional part)	(for FS1)
TCOND8:	exam mark = alphabetic	(for FS1)
TCOND9:	exam mark = special character	(for FS1)
TCOND10:	coursework mark = real number	(for FS1)
TCOND11:	coursework mark = alphabetic	(for FS1)
TCOND 12:	coursework mark = special character	(for FS1)

Next, the partitions for the valid outputs are identified:

TCOND13:	'A' is induced by	70 <= total mark <= 100	(for FS1)
TCOND14:	'B' is induced by	50 <= total mark < 70	(for FS1)
TCOND15:	'C' is induced by	30 <= total mark < 50	(for FS1)
TCOND16:	'D' is induced by	0 <= total mark < 30	(for FS1)
TCOND17:	'Fault Message' (FM)	is induced by total mark > 100	(for FS1)
TCOND18:	'Fault Message' (FM)	is induced by total mark < 0	(for FS1)

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:

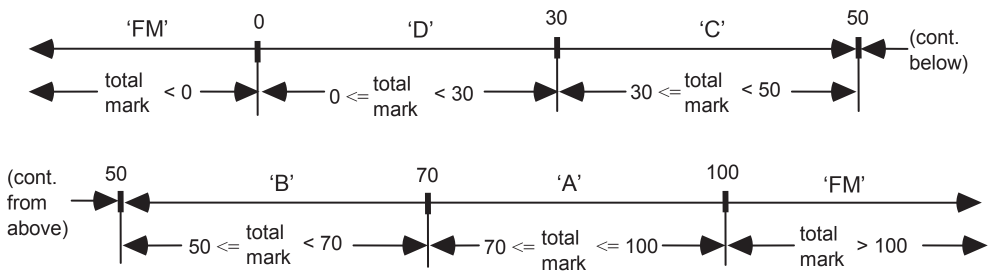


Figure 6 — Equivalence partitions and boundaries for total mark

An invalid output would be any output from the test item other than one of the five specified. It can be challenging to identify unspecified outputs. However, they must be considered because if we can cause one to occur, then we have identified a flaw with the test item, its test basis, 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 may identify different partitions which *they* feel could possibly occur.

TCOND19:	output = 'E', to induce a failure grade	(for FS1)
TCOND20:	output = 'A+' to induce a grade above the upper boundary	(for FS1)
TCOND21:	output = 'null'	(for FS1)

B.2.1.5 Step 3: Derive Test Coverage Items (TD3)

In equivalence partitioning, the test coverage items are the partitions that were derived in the previous step (i.e. test conditions are the same as test coverage items in this technique). Thus, the following test coverage items can be defined.

TCOVER1:	0 <= exam mark <= 75		(for TCOND1)
TCOVER2:	0 <= coursework mark <= 25		(for TCOND2)
TCOVER3:	exam mark < 0		(for TCOND3)
TCOVER4:	exam mark > 75		(for TCOND4)
TCOVER5:	coursework mark < 0		(for TCOND5)
TCOVER6:	coursework mark > 25		(for TCOND6)
TCOVER7:	exam mark = real number (a number with a fractional part)		(for TCOND7)
TCOVER8:	exam mark = alphabetic		(for TCOND8)
TCOVER9:	exam mark = special character		(for TCOND9)
TCOVER10:	coursework mark = real number		(for TCOND10)
TCOVER11:	coursework mark = alphabetic		(for TCOND11)
TCOVER12:	coursework mark = special character		(for TCOND12)
TCOVER13:	'A' is induced by	70 <= total mark <= 100	(for TCOND13)
TCOVER14:	'B' is induced by	50 <= total mark < 70	(for TCOND14)
TCOVER15:	'C' is induced by	30 <= total mark < 50	(for TCOND15)
TCOVER16:	'D' is induced by	0 <= total mark < 30	(for TCOND16)
TCOVER17:	'Fault Message' (FM)	is induced by total mark > 100	(for TCOND17)
TCOVER18:	'Fault Message' (FM)	is induced by total mark < 0	(for TCOND18)
TCOVER19:	output = 'E' to induce a failure a failure grade		(for TCOND19)
TCOVER20:	output = 'A+' to induce a grade above the upper boundary		(for TCOND20)
TCOVER21:	output = 'null'		(for TCOND21)

B.2.1.6 Step 4: Derive Test Cases (TD4)

B.2.1.6.1.1 Options

Having identified partitions and test coverage items to be tested, test cases are derived that attempt to "hit" each test coverage item. Two common approaches for test case design are one-to-one and minimized equivalence partitioning. In the first a test case is generated for each identified partition on a one-to-one basis (see option 4a below), while in the second a minimal set of test cases is generated to cover all the identified partitions (see option 4b below). The preconditions of all test cases for the *generate_grading* function are the same: that the application is ready to take the inputs of exam and coursework mark.

B.2.1.6.1.2 Option 4a: Derive Test Cases for One-to-One Equivalence Partitioning (TD4)

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 test coverage item being targeted is stated explicitly. Twenty-one test coverage items were identified leading to twenty-one test cases.

The test cases corresponding to partitions derived from the input exam mark are shown below. Note that the input coursework mark in the following test case table has been set to an arbitrary valid value of 15. The allocation of an arbitrary valid value to all inputs in the test case (other than the one being tested) has been carried out for all test cases in this clause.

Table 3 — Test cases for input exam mark

Test Case	1	2	3
Input (exam mark)	60	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	75	5	108
Test Coverage Item covered (exam mark)	1	3	4
Partition tested (of exam mark)	0 <= e <= 75	e < 0	e > 75
Exp. Output	'A'	'FM'	'FM'

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

Table 4 — Test cases for input coursework mark

Test Case	4	5	6
Input (exam mark)	40	40	40
Input (c/w mark)	20	-15	47
total mark (as calculated)	60	25	87
Test Coverage Item covered (c/w mark)	2	5	6
Partition tested (of c/w mark)	0 <= c <= 25	c < 0	c > 25
Exp. Output	'B'	'FM'	'FM'

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

Table 5 — Test cases for invalid inputs for exam mark

Test Case	7	8	9
Input (exam mark)	60.5	Q	$
Input (c/w mark)	15	15	15
total mark (as calculated)	75.5	not applicable	not applicable
Test Coverage Item covered (exam mark)	7	8	9
Partition tested	exam mark = real number	exam mark = alphabetic	exam mark = special char
Exp. Output	'FM'	'FM'	'FM'

Table 6 — Test cases for invalid inputs for coursework mark

Test Case	10	11	12
Input (exam mark)	40	40	40
Input (c/w mark)	20.23	G	@
total mark (as calculated)	60.23	not applicable	not applicable
Test Coverage Item covered (c/w mark)	10	11	12
Partition tested	c/w mark = real number	c/w mark = alphabetic	c/w mark = special char
Exp. Output	'FM'	'FM'	'FM'

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

Table 7 — Test cases for valid output total mark

Test Case	13	14	15
Input (exam mark)	60	44	32
Input (c/w mark)	20	22	13
total mark (as calculated)	80	66	45
Test Coverage Item covered (total mark)	13	14	15
Partition tested (of total mark)	70 <= t <= 100	50 <= t < 70	30 <= t < 50
Exp. Output	'A'	'B'	'C'

Table 8 — Test cases for valid output total mark (continued)

Test Case	16	17	18
Input (exam mark)	12	80	-10
Input (c/w mark)	5	60	-10
total mark (as calculated)	17	140	-20
Test Coverage Item covered (total mark)	16	17	18
Partition tested (of total mark)	0 <= t < 30	t > 100	t < 0
Exp. Output	'D'	'FM'	'FM'

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

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

Table 9 — Test cases for invalid output total mark (continued)

Test Case	19	20	21
Input (exam mark)	-10	100	Null
Input (c/w mark)	0	10	Null
total mark (as calculated)	-10	110	Null+Null
Test Coverage Item covered (total mark)	19	20	21
Partition tested (output)	'E'	'A+'	'Null'
Exp. Output	'FM'	'FM'	'FM'

Depending on the implementation, it may be impossible to execute test cases that contain invalid input values (e.g. test cases 2, 3, 5-12, and 17-21 in the example above). 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.

B.2.1.6.1.3 Option 4b: Derive Test Cases for Minimized Equivalence Partitioning (TD4)

It can be seen above that several of the test cases are similar, such as test cases 1 and 13, where the main difference between them is the specific test coverage item chosen from the partition targeted. As the test item 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 twelve test cases corresponds to the Minimized 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.

Table 10 — Minimized test cases

Test Case	1	2	3	4
Input (exam mark)	60	50	35	19
Input (c/w mark)	20	16	10	8
total mark (as calculated)	80	66	45	27
Test Coverage Item covered	1, 2, 13	1, 2, 14	1, 2, 15	1, 2, 16
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'

Table 11 — Minimized test cases (continued)

Test Case	5	6	7	8
Input (exam mark)	-10	93	60.5	Q
Input (c/w mark)	-15	47	20.23	G
total mark (as calculated)	-20	140	80.73	-
Test Coverage Item covered	3, 5, 18	4, 6, 17	7, 10, 13	8, 11
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'

Table 12 — Minimized test cases (continued)

Test Case	9	10	11	12
Input (exam mark)	$	-10	100	'Null'
Input (c/w mark)	@	5	10	'Null'
total mark (as calculated)	-	-5	110	Null+Null
Test Coverage Item covered	9, 12	3, 2, 18	4, 2, 17	21
Partition (of exam mark)	e = special char	e < 0	e > 75	-
Partition (of c/w mark)	c = special char	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'	'FM'

The one-to-one and minimized approaches represent the two extremes of a spectrum of approaches to equivalence partitioning. One-to-one test cases are particularly useful for testing error conditions (i.e. when you are trying to force specific error messages to be output), for example, to reduce the chances that one error condition halts processing and/or mask or block other error conditions. On the other hand, 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. 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. Therefore, a common approach is to combine the two approaches by applying minimized equivalence partitioning to design valid test cases and one-to-one equivalence partitioning to design invalid test cases.

B.2.1.7 Step 5: Assemble Test Sets (TD5)

B.2.1.7.1.1 Options

If we assume that it is possible to automatically check an accept/reject response for each test case, but that automation cannot handle fault messages (FM), then we can generate two test sets (TS); one for manual testing and one for automated testing.

B.2.1.7.1.2 Option 5a: Assemble Test Set for One-to-One Equivalence Partitioning (TD5)

TS1: Manual Testing – TEST CASES 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 17, 18, 19, 20, 21.
TS2: Automated Testing – TEST CASES 1, 4, 13, 14, 15, 16.

B.2.1.7.1.3 Option 5b: Assemble Test Set for Minimized Equivalence Partitioning (TD5)

TS3: Manual Testing – TEST CASES 5, 6, 7, 8, 9, 10, 11, 12.
TS4: Automated Testing – TEST CASES 1, 2, 3, 4.

B.2.1.8 Step 6: Derive Test Procedures (TD6)

B.2.1.8.1.1 Options

Test procedures for one-to-one and minimized equivalence partitioning can now be derived.

B.2.1.8.1.2 Option 6a: Derive Test Procedures for One-to-One Equivalence Partitioning (TD6)

For the manual test cases in test set TS1 for one-to-one equivalence partitioning, one test procedure (TP) can be defined as follows:

TP1: manual testing, covering all test cases in TS1, in the order specified in the test set.

For the automated test cases in one-to-one test set TS2, one test script could be written to execute all test cases in the test set, as follows:

TP2: automated testing, covering all test cases in TS2, in the order specified in the test set.

For the automated test procedures TP2, automation code that implements the procedure would need to be written in test automation scripts.

B.2.1.8.1.3 Option 6b: Derive Test Procedures for Minimized Equivalence Partitioning (TD6)

For the manual test cases in minimized test set TS3, one test procedure (TP) can be defined as follows:

TP1: manual testing, covering all test cases in TS3, in the order specified in the test set.

For the automated test cases in minimized test set TS4, one test script could be written to execute all test cases in the test set, as follows:

TP2: automated testing, covering all test cases in TS4, in the order specified in the test set.

B.2.1.9 Equivalence Partition Coverage

Using the formula provided in clause 6.1.2 and the test coverage items derived above:

$$Coverage_{(one\text{-}to\text{-}one\_EP)} = \frac{21}{21} \times 100\% = 100\%$$

$$Coverage_{(\min imized\_EP)} = \frac{21}{21} \times 100\% = 100\%$$

Thus, 100% equivalence partition coverage was achieved for both one-to-one and minimized equivalence partitioning, enabling all twenty-one identified partitions to be exercised by at least one test case. Lower coverage levels would be achieved if not all partitions identified are exercised. If not all partitions are identified, then any coverage measure based on this incomplete set of partitions could be misleading. However, since a different analysis of the test item might identify different equivalence partitions, particularly for "invalid" values, coverage measures for equivalence partitioning must be qualified as being for “identified” partitions.

B.2.2 Classification Tree Method

B.2.2.1 Introduction

The aim of the classification tree method is to derive test cases that cover the input partitions of the test item according to the chosen level of equivalence partition coverage. The classification tree method extends this concept by constructing a classification tree that illustrates the partitions and assists the tester with test design.

B.2.2.2 Example

Consider the test basis for a test item *travel_preference*, which records the travel preferences of staff of an Australian organisation who travel to major Australian capital cities for work purposes. Each set of travel preferences is chosen through a series of radio buttons, which consist of the following input value choices:

Destination =	*Adelaide, Brisbane, Canberra, Darwin, Hobart, Melbourne, Perth, Sydney*
Class =	*First Class, Business Class, Economy*
Seat =	*Aisle, Window*
Meal Preference =	*diabetic, gluten free, lacto-ovo vegetarian, low fat/cholesterol, low lactose, vegan vegetarian*

As some staff may prefer a standard meal, Meal Preference is an optional input (designated as 'null' in test case design steps 1, 2, 3 and 4 below). Any combination of one class from each classification will result in the message "Booking accepted" while any other input will result in an error message stating "invalid input".

B.2.2.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: travel_preference function

B.2.2.4 Step 2: Derive Test Conditions (TD2)

For classification tree testing, test conditions are identified by deriving classifications and classes for each input parameter. In this example, the input "Meal Preference" could be decomposed into two categories "Vegetarian" and "Non-Vegetarian" with corresponding classes.

TCOND1:	Destination = Adelaide, Brisbane, Canberra, Darwin, Hobart, Melbourne, Perth, Sydney	(for FS1)
TCOND2:	Class = First, Business, Economy	(for FS1)
TCOND3:	Seat = Aisle, Window	(for FS1)
TCOND4:	Vegetarian = lacto-ovo, vegan	(for FS1)
TCOND5:	Non-Vegetarian = diabetic, gluten free, low fat/cholesterol, low lactose, null	(for FS1)
TCOND6:	Meal Preference = null	(for FS1)

NOTE Invalid test conditions could also be derived, though these are not demonstrated in this example.

A classification tree can now be developed for these test conditions.

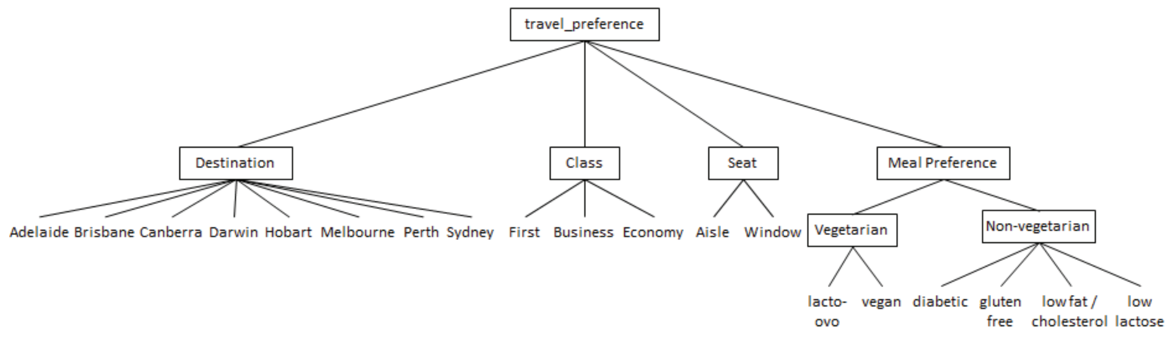


Figure 7 — Classification tree example

B.2.2.5 Step 3: Derive Test Coverage Items (TD3)

Test coverage items are derived by choosing a combination approach and then creating combinations of classes according to that approach. A "combination table" can be constructed under the classification tree to demonstrate which classes are combined to form each test coverage item (e.g. see figure 8 below). The classes that are covered by each test coverage item are marked by a series of tokens (black dots) that run horizontally underneath the classification tree.

If we assume that the chosen combination approach is "minimal", in which each test coverage item covers as many classes as possible until all classes are included in at least one test case, then the following test coverage items could be defined.

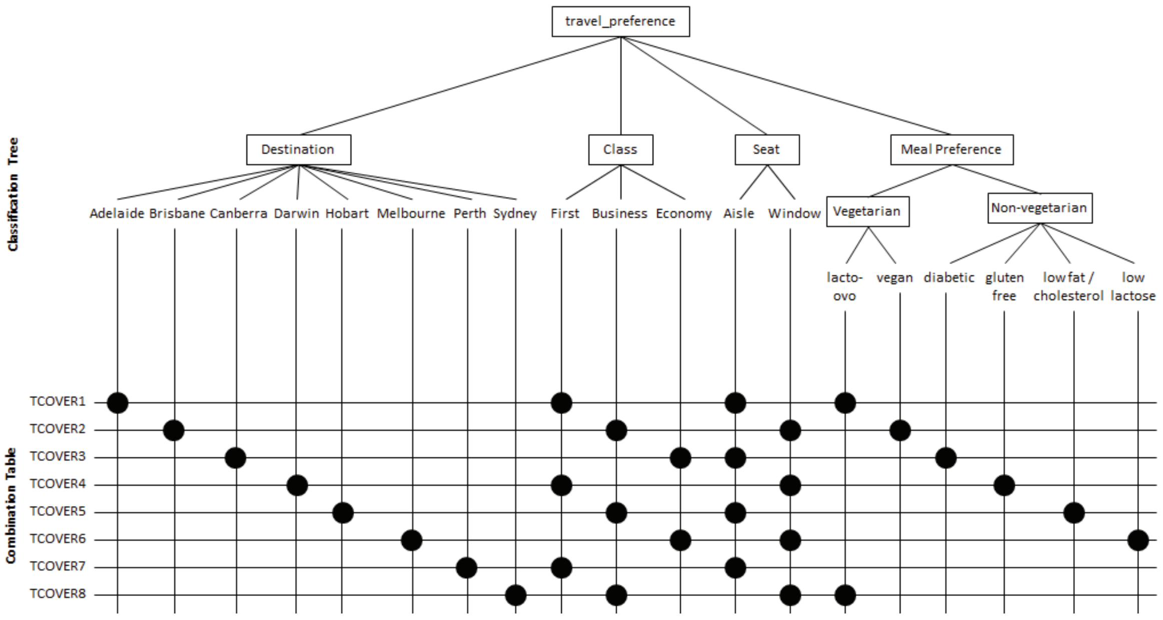


Figure 8 — Example of a classification tree and corresponding combination table

The following lists the test condition coverage achieved by the test coverage items defined in Figure 8.

TCOVER1: covers TCOND1, TCOND2, TCOND3, TCOND4
TCOVER2: covers TCOND1, TCOND2, TCOND3, TCOND4
TCOVER3: covers TCOND1, TCOND2, TCOND3, TCOND5
TCOVER4: covers TCOND1, TCOND2, TCOND3, TCOND5
TCOVER5: covers TCOND1, TCOND2, TCOND3, TCOND5
TCOVER6: covers TCOND1, TCOND2, TCOND3, TCOND5
TCOVER7: covers TCOND1, TCOND2, TCOND3, TCOND6
TCOVER8: covers TCOND1, TCOND2, TCOND3, TCOND4

B.2.2.6 Step 4: Derive Test Cases (TD4)

A set of test cases can now be derived, where each test case covers exactly one test coverage item. Test cases are derived by selecting one test coverage item at a time that has not already been covered by a test case and populating it with test input values that cover the classes of that combination. This is repeated until the required level of coverage is achieved. The expected result is derived by applying the inputs to the test basis. In this particular case, any combination of valid inputs results in the status "Booking accepted”.

Table 13 — Test cases for classification tree testing

Test Case	Input Values				Expected Result	Test Coverage Item(s) Covered
	Destination	Class	Seat	Meal Preference		
1	Adelaide	First	Aisle	lacto-ovo	Booking accepted	TCOVER1
2	Brisbane	Business	Window	vegan	Booking accepted	TCOVER2
3	Canberra	Economy	Aisle	diabetic	Booking accepted	TCOVER3
4	Darwin	First	Window	gluten free	Booking accepted	TCOVER4
5	Hobart	Business	Aisle	low fat/cholesterol	Booking accepted	TCOVER5
6	Melbourne	Economy	Window	low lactose	Booking accepted	TCOVER6
7	Perth	First	Aisle	null	Booking accepted	TCOVER7
8	Sydney	Business	Window	lacto-ovo	Booking accepted	TCOVER8

B.2.2.7 Step 5: Assemble Test Sets (TD5)

Since a small number of test cases were derived in this example, it may be decided that they be combined into the one test set.

TS1: TEST CASES 1, 2, 3, 4, 5, 6, 7, 8

B.2.2.8 Step 6: Derive Test Procedures (TD6)

Since all test cases are in the one test set, we can derive one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

B.2.2.9 Classification Tree Method Coverage

Using the formula provided in clause 6.1.3 and the test coverage items derived above:

$$Coverage_{(classification\_tree\_method)} = \frac{8}{8} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for the classification tree method has been achieved.

B.2.3 Boundary Value Analysis

B.2.3.1 Introduction

The aim of boundary value analysis is to derive a set of test cases that cover boundaries of each input and output partition of the test item according to the chosen level of boundary value coverage. It is based on the following premises. First, that the inputs and outputs of a test item can be partitioned into classes that, according to the test basis for the test item, will be treated similarly by the test item; that the members of some partitions can be ordered from lowest to highest with no discontinuity; and that the boundaries of ordered contiguous partitions are historically an error prone element of software development. Test cases are generated to exercise these boundaries.

The following is an example *three-value boundary testing* with *one-to-one test cases* (see clause 5.2.3.2 and 5.2.3.3 respectively). In order to derive boundaries for a test item, the equivalence partitions of the test item must be identified first, followed by the derivation of the boundary values from each equivalence class.

B.2.3.2 Example

Consider a test item, *generate_grading*, with the following test basis:

The component receives as input an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it outputs a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark, which is 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.

B.2.3.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set (FS) needs to be defined:

FS1: generate_grading function

B.2.3.4 Step 2: Derive Test Conditions (TD2)

B.2.3.4.1 Sub-steps

For boundary value analysis, the test conditions are the boundaries (between partitions) that have been chosen to be covered during testing. To identify the boundaries, equivalence partitions must first be identified (see step 2a below), from which test conditions (the boundaries) can be derived (see step 2b below).

B.2.3.4.2 Step 2a: Identify Equivalence Partitions

Equivalence partitions are identified from the valid and invalid inputs and outputs of the feature set FS1.

The following *valid* equivalence partitions can be identified for the inputs:

EP1:	0 <= exam mark <= 75	(for FS1)
EP2:	0 <= coursework mark <= 25	(for FS1)

The most obvious *invalid* equivalence partitions for the inputs can be identified as:

EP3:	exam mark > 75	(for FS1)
EP4:	exam mark < 0	(for FS1)
EP5:	coursework mark > 25	(for FS1)
EP6:	coursework mark < 0	(for FS1)

Although partitions EP3 to EP6 appear to be bounded on one side only, these partitions are in fact bounded by implementation-dependent minimum and maximum values. For integers held in sixteen bits these would be 32767 and -32768 respectively. Therefore, EP3 to EP6 can be more fully defined as follows:

EP3:	75 < exam mark <= 32767	(for FS1)
EP4:	-32768 <= exam mark < 0	(for FS1)
EP5:	25 < coursework mark <= 32767	(for FS1)
EP6:	-32768 <= coursework mark < 0	(for FS1)

Partitioned ranges of values can be represented pictorially as follows:

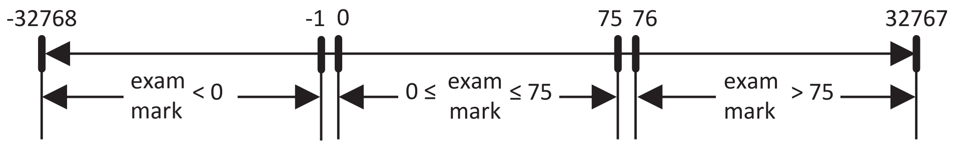


Figure 9 — Equivalence partitions and boundaries of exam mark

And for the input, coursework mark, we get:

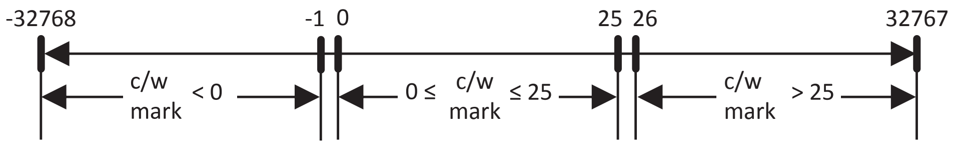


Figure 10 — Equivalence partitions and boundaries of coursework mark

Less obvious invalid input partitions could include any other input types not included in a valid partition, for instance, non-integer inputs or perhaps non-numeric inputs. This aspect of equivalence partitioning can be subjective and as such, each tester may identify different partitions that they feel could be relevant. In order to be considered an equivalence partition those values within it must be expected, from the test basis, to be treated in an equivalent manner by the test item. Thus we could generate the following invalid equivalence partitions for the two input fields:

EP7:	exam mark = real number	(for FS1)
EP8:	exam mark = alphabetic	(for FS1)
EP9:	exam mark = special character	(for FS1)
EP10:	coursework mark = real number	(for FS1)
EP11:	coursework mark = alphabetic	(for FS1)
EP12:	coursework mark = special character	(for FS1)

Although equivalence classes EP7 to EP12 are possible, they have no identifiable boundaries and so no test coverage items or test cases need to be derived for them for this technique.

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

EP13:	'A'	is induced by	70 <= total mark <= 100	(for FS1)
EP14:	'B'	is induced by	50 <= total mark < 70	(for FS1)
EP15:	'C'	is induced by	30 <= total mark < 50	(for FS1)
EP16:	'D'	is induced by	0 <= total mark < 30	(for FS1)
EP17:	'Fault Message' (FM)		is induced by total mark > 100	(for FS1)
EP18:	'Fault Message' (FM)		is induced by total mark < 0	(for FS1)

where total mark = exam mark + coursework mark.

Similar to the inputs, the output is bounded on either side by implementation-dependent maximum and minimum values. Assuming the output is stored in integers held in sixteen bits from -32768 to 32767, EP17 and EP18 could be redefined as follows.

EP17:	100 < total mark <= 32767	(for FS1)
EP18:	-32768 <= total mark < 0	(for FS1)

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

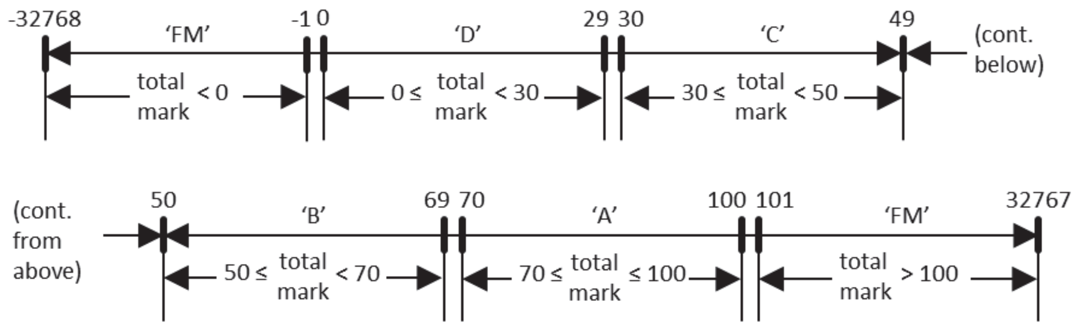


Figure 11 — Equivalence partitions and boundaries of total mark

An *invalid* output would be any output from the test item other than one of the five specified. It may be difficult to identify unspecified outputs, but obviously they must be considered as if we can cause one then we have identified a defect with the test item, its test basis, 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.

B.2.3.4.3 Step 2b: Derive Test Conditions

Once equivalence partitions for each input and output field have been identified, the test conditions, which are the boundaries of each equivalence partition, can now be identified.

For the *valid* partitions of the input fields exam mark and coursework mark, the following test conditions can be identified. Note that duplicate boundaries (e.g. the boundary “0” that lies on the edges of EP1 and EP4) are only covered by one test condition.

TCOND1:	exam mark = 0	(for EP1 and EP4)
TCOND2:	exam mark = 75	(for EP1 and EP3)
TCOND3:	coursework mark = 0	(for EP2 and EP6)
TCOND4:	coursework mark = 25	(for EP2 and EP5)

For the *valid* equivalence partitions for total mark, the following boundaries can be identified.

TCOND5:	total mark = 0	(for EP16 and EP18)
TCOND6:	total mark = 29	(for EP15 and EP16)
TCOND7:	total mark = 30	(for EP15 and EP16)
TCOND8:	total mark = 49	(for EP14 and EP15)
TCOND9:	total mark = 50	(for EP14 and EP15)
TCOND10:	total mark = 69	(for EP13 and EP14)
TCOND11:	total mark = 70	(for EP13 and EP14)
TCOND12:	total mark = 100	(for EP13 and EP17)
TCOND13:	total mark = 101	(for EP13 and EP17)

For the *invalid* partitions of the input fields, the following boundaries can be identified.

TCOND14:	exam mark = 32767	(for EP3)
TCOND15:	exam mark = -32768	(for EP4)
TCOND16:	coursework mark = 32767	(for EP5)
TCOND17:	coursework mark = -32768	(for EP6)

Finally, for the *invalid* partitions of the output field total mark, the following boundaries can be identified.

TCOND18:	total mark = 32767	(for EP17)
TCOND19:	total mark ‘FM’ = -32768	(for EP18)

B.2.3.5 Step 3: Derive Test Coverage Items (TD3)

If 3-value boundary value analysis is applied, the test coverage items are the values that are on the boundary of the equivalence partition, and either side of the boundary, using the smallest significant distance away, as shown below:

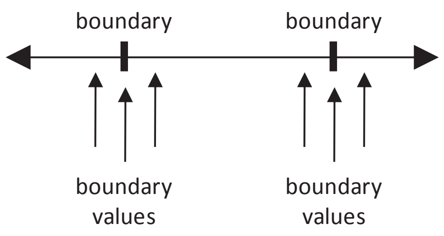


Figure 12— Test coverage items for 3-value boundary value analysis

NOTE 1 Alternatively, 2-value boundary value analysis could also be performed, which would result in a smaller number of test coverage items and test cases being derived.

For the boundaries that were identified in the previous step as test conditions, the following valid test coverage items (TCOVER) can be identified. As we are using integers in this example, the test coverage items are one either side of each boundary.

NOTE 2 If the example involved numerical data types involving decimals (e.g. real numbers), then the test coverage items for boundary value analysis would be the smallest significant value for the data type under consideration.

TCOVER1:	exam mark = -1	(from TCOND1)
TCOVER2:	exam mark = 0	(from TCOND1)
TCOVER3:	exam mark = 1	(from TCOND1)
TCOVER4:	exam mark = 74	(from TCOND2)
TCOVER5:	exam mark = 75	(from TCOND2)
TCOVER6:	exam mark = 76	(from TCOND2)
TCOVER7:	coursework mark = -1	(from TCOND3)
TCOVER8:	coursework mark = 0	(from TCOND3)
TCOVER9:	coursework mark = 1	(from TCOND3)
TCOVER10:	coursework mark = 24	(from TCOND4)
TCOVER11:	coursework mark = 25	(from TCOND4)
TCOVER12:	coursework mark = 26	(from TCOND4)

For the output fields, the following *valid* test coverage items can be identified:

TCOVER13:	total mark = -1	(from TCOND5)
TCOVER14:	total mark = 0	(from TCOND5)
TCOVER15:	total mark = 1	(from TCOND5)
TCOVER16:	total mark = 28	(from TCOND6)
TCOVER17:	total mark = 29	(from TCOND6 & TCOND7)
TCOVER18:	total mark = 30	(from TCOND6 & TCOND7)
TCOVER19:	total mark = 31	(from TCOND7)
TCOVER20:	total mark = 48	(from TCOND8)
TCOVER21:	total mark = 49	(from TCOND8 & TCOND9)
TCOVER22:	total mark = 50	(from TCOND8 & TCOND9)
TCOVER23:	total mark = 51	(from TCOND9)
TCOVER24:	total mark = 68	(from TCOND10)
TCOVER25:	total mark = 69	(from TCOND10 & TCOND11)
TCOVER26:	total mark = 70	(from TCOND10 & TCOND11)
TCOVER27:	total mark = 71	(from TCOND11)
TCOVER28:	total mark = 99	(from TCOND12)
TCOVER29:	total mark = 100	(from TCOND12 & TCOND13)
TCOVER30:	total mark = 101	(from TCOND12 & TCOND13)
TCOVER31:	total mark = 102	(from TCOND13)

Note that test conditions EP7 to EP12 have not been covered by any test coverage items, since they have no identifiable boundaries (as stated in the previous step).

For the remaining invalid partitions that were identified (i.e. for the boundaries of test conditions TCOND14 to TCOND19 that have not yet been covered), the following invalid test coverage items can be identified:

TCOVER32:	exam mark = 32766	(from TCOND14)
TCOVER33:	exam mark = 32767	(from TCOND14)
TCOVER34:	exam mark = 32768	(from TCOND14)
TCOVER35:	exam mark = -32769	(from TCOND15)
TCOVER36:	exam mark = -32768	(from TCOND15)
TCOVER37:	exam mark = -32767	(from TCOND15)
TCOVER38:	coursework mark = 32766	(from TCOND16)

TCOVER39:	coursework mark = 32767	(from TCOND16)
TCOVER40:	coursework mark = 32768	(from TCOND16)
TCOVER41:	coursework mark = -32769	(from TCOND17)
TCOVER42:	coursework mark = -32768	(from TCOND17)
TCOVER43:	coursework mark = -32767	(from TCOND17)
TCOVER44:	total mark = 32766	(from TCOND18)
TCOVER45:	total mark = 32767	(from TCOND18)
TCOVER46:	total mark = 32768	(from TCOND18)
TCOVER47:	total mark = -32769	(from TCOND19)
TCOVER48:	total mark = -32768	(from TCOND19)
TCOVER49:	total mark = -32767	(from TCOND19)

B.2.3.6 Step 4: Derive Test Cases (TD4)

Test cases can now be derived to cover the required percentage of test coverage items that were identified in the previous step. For example, if 100% boundary coverage is required, then test cases must be derived to cover all test coverage items. One-to-one boundary value analysis could be used to derive one test case per test coverage item, or minimized boundary value analysis could be used to derive the minimum number of test cases required to cover all test coverage items.

The preconditions of all test cases for the *generate_grading* function are the same: that the application is ready to take the inputs of exam and coursework mark.

If 100% boundary coverage is required for this example, and one-to-one boundary value analysis is used to derive test cases, then six test cases that can be derived for the input exam mark are shown in the tables below. Each test case is derived by selecting one boundary value (test coverage item) for inclusion in each test case, allocating an arbitrary valid value to all other inputs present in the test case and then determining the expected result of the test.

Table 14 — Boundary values for exam mark

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
total mark (as calculated)	14	15	16	89	90	91
Test Coverage Item covered (exam mark)	1	2	3	4	5	6
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, as these test cases are focused on exercising the input exam mark boundaries.

And the test cases derived from the input coursework mark are:

Table 15 — Boundary values for coursework mark

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
total mark (as calculated)	39	40	41	64	65	66
Test Coverage Item covered (c/w mark)	7	8	9	10	11	12
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.

The test cases derived from the valid outputs are:

Table 16 — Test cases for total mark

Test Case	13	14	15	16	17	18	19
Input (exam mark)	-1	0	0	28	29	15	6
Input (c/w mark)	0	0	1	0	0	15	25
total mark (as calculated)	-1	0	1	28	29	30	31
Test Coverage Item covered (total mark)	13	14	15	16	17	18	19
Boundary tested (total mark)	0			29	29, 30		30
Exp. Output	'FM'	'D'	'D'	‘D’	'D'	'C'	'C'

Table 17 — Test cases for total mark (continued)

Test Case	20	21	22	23	24	25	26	27
Input (exam mark)	23	24	50	26	48	49	45	71
Input (c/w mark)	25	25	0	25	20	20	25	0
Total Mark (as calculated)	48	49	50	51	68	69	70	71
Test Coverage Item covered (total mark)	20	21	22	23	24	25	26	27
Boundary tested (total mark)	49	49, 50		50	69	69, 70		70
Exp. Output	‘C’	'C'	'B'	'B'	‘B’	'B'	'A'	'A'

Table 18 — Test cases for total mark (continued)

Test Case	28	29	30	31
Input (exam mark)	74	75	75	75
Input (c/w mark)	25	25	26	27
Total Mark (as calculated)	99	100	101	102
Test Coverage Item covered (total mark)	28	29	30	31
Boundary tested (total mark)	100	100, 101		101
Exp. Output	'A'	'A'	'FM'	‘FM’

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

The following test cases are also required to cover the additional test coverage items TCOVER32 to TCOVER49, which were identified at the extreme edges of equivalence classes:

Table 19 — Test cases for exam mark

Test Case	32	33	34	35	36	37
Input (exam mark)	32766	32767	32768	-32769	-32768	-32767
Input (c/w mark)	15	15	15	15	15	15
total mark (as calculated)	32781	32782	32783	-32754	-32753	-32752
Test Coverage Item covered (exam mark)	32	33	34	35	36	37
Boundary tested (exam mark)	32767			-32768		
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Table 20 — Test cases for coursework mark

Test Case	38	39	40	41	42	43
Input (exam mark)	40	40	40	40	40	40
Input (c/w mark)	32766	32767	32768	-32769	-32768	-32767
total mark (as calculated)	32806	32807	32808	-32729	-32728	-32727
Test Coverage Item covered (c/w mark)	38	39	40	41	42	43
Boundary tested (c/w mark)	32767			-32768		
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Table 21 — Test cases for total mark

Test Case	44	45	46	47	48	49
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
Test Coverage Item covered (total mark)	44	45	46	47	48	49
Boundary tested (total mark)	32767			-32768		
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

It should be noted that when invalid input values are used (as above, in test cases 1, 6, 7, 12, 13, and 30-49) 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 for 3-value boundary value analysis as it enables all identified test coverage items to be 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.2.3.7 Step 5: Assemble Test Sets (TD5)

If we assume that it is possible to automatically check an accept/reject response for each valid test case, but that automation cannot handle fault messages (FM) that are generated by invalid test cases, then we can generate two test sets (TS); one for manual testing and one for automated testing.

TS1: Manual Testing – TEST CASES 1, 6, 7, 12, 13, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48 and 49.
TS2: Automated Testing – TEST CASES 2, 3, 4, 5, 8, 9, 10, 11, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28 and 29.

B.2.3.8 Step 6: Derive Test Procedures (TD6)

For the manual test cases in test set TS1, one test procedure (TP) can be defined as follows:

TP1: manual testing, covering all test cases in TS1, in the order specified in the test set.

For the automated test cases in test set TS2, one test script could be written to execute all test cases in the test set, as follows:

TP2: automated testing, covering all test cases in TS2, in the order specified in the test set.

For the automated test procedure TP2, automation code to execute the procedure would need to be written.

B.2.3.9 Boundary Value Analysis Coverage

Using the formula provided in clause 6.1.4 and the test coverage items derived above:

$$Coverage_{(boundary\_value\_analysis)} = \frac{49}{49} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for boundary value analysis has been achieved.

B.2.4 Syntax Testing

B.2.4.1 Introduction

The aim of syntax testing is to derive a set of test cases that cover the input syntax of the test item according to the chosen level of input syntax coverage. This technique is based upon an analysis of the test basis of the test item 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.

B.2.4.2 Example

Consider a test item that simply checks whether an input *float_in* conforms to the syntax of a floating point number, `float` (defined below). The test item 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 test item. `|` 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.2.4.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature needs to be defined:

FS1: `float_in`

B.2.4.4 Step 2: Derive Test Conditions (TD2)

The first step is to derive the test conditions from the syntax. Test conditions may be defined as the input parameters in the syntax, as follows:

float	=	int "e" int	TCOND1
int	=	["+"|"-"] nat	TCOND2
nat	=	{dig}	TCOND3
dig	=	"0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"	TCOND4

B.2.4.5 Step 3: Derive Test Coverage Items (TD3)

The test coverage items for syntax testing are the "options" (valid test coverage items) and "mutations" (invalid test coverage items) of the defined syntax (see clause 5.2.4.2 for definitions of "options" and "mutations").

Valid test coverage items can be derived for the elements on the right hand side of the BNF definition. There are three test coverage items that can be derived for the "+" and "-" signs of TCOND2:

TCOVER1:	there is no "+" or "-" sign	(for TCOND2 & TCOND1)
TCOVER2:	there is a "+" sign	(for TCOND2 & TCOND1)
TCOVER3:	there is a `"-"` sign	(for TCOND2 & TCOND1)

NOTE 1 Separate test coverage items could be derived for the first and second instances of "+" and "-"if required.

`nat` has two test coverage items:

TCOVER4:	`nat` is a single digit number	(for TCOND3 & TCOND2)
TCOVER5:	`nat` is a multiple digit number	(for TCOND3 & TCOND2)

NOTE 2 Separate test coverage items could be derived for the first and second instances of `nat` if required.

`dig` has ten options:

TCOVER6:	integer is a "0"	(for TCOND4 & TCOND3)
TCOVER7:	integer is a "1"	(for TCOND4 & TCOND3)
TCOVER8:	integer is a "2"	(for TCOND4 & TCOND3)
TCOVER9:	integer is a "3"	(for TCOND4 & TCOND3)
TCOVER10:	integer is a "4"	(for TCOND4 & TCOND3)
TCOVER11:	integer is a "5"	(for TCOND4 & TCOND3)
TCOVER12:	integer is a "6"	(for TCOND4 & TCOND3)
TCOVER13:	integer is a "7"	(for TCOND4 & TCOND3)
TCOVER14:	integer is an "8"	(for TCOND4 & TCOND3)
TCOVER15:	integer is a "9"	(for TCOND4 & TCOND3)

There are thus fifteen valid test coverage items that can be defined.

The first step in deriving invalid test coverage items is to construct a checklist of generic mutations that can be applied to the test conditions. 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.

NOTE 3 Other types of syntax mutation could be used, depending on the types of defects testing is intending to target.

These generic mutations are applied to the individual elements of the syntax to yield specific mutations.

TCOVER16:	apply m1 to first "`int`"	(for TCOND1)
TCOVER17:	apply m1 to "`e`"	(for TCOND1)
TCOVER18:	apply m1 to second "`int`"	(for TCOND1)
TCOVER19:	apply m1 to "`["+"|"-"]`"	(for TCOND2)
TCOVER20:	apply m1 to "`nat`"	(for TCOND2)
TCOVER21:	apply m2 to substitute "`e`" for first "`int`"	(for TCOND1)
TCOVER22:	apply m2 to substitute "`["+"|"-"]`" first "`int`"	(for TCOND 1 & TCOND2)
TCOVER23:	apply m2 to substitute first "`int`" for "e"	(for TCOND1)

TCOVER24:	apply m2 to substitute “`["+"|"-"]`” for “`e`”	(for TCOND1 & TCOND2)
TCOVER25:	apply m2 to substitute “`e`” for second “`int`”	(for TCOND1)
TCOVER26:	apply m2 to substitute “`["+"|"-"]`” for second “`int`”	(for TCOND1)
TCOVER27:	apply m2 to substitute “`e`” for “`["+"|"-"]`”	(for TCOND1 & TCOND2)
TCOVER28:	apply m2 to substitute “`e`” for “`nat`”	(for TCOND1 & TCOND2)
TCOVER29:	apply m2 to substitute “`["+"|"-"]`” for “`nat`”	(for TCOND1 & TCOND2)
TCOVER30:	apply m3 to first “`int`”	(for TCOND1)
TCOVER31:	apply m3 to “`e`”	(for TCOND1)
TCOVER32:	apply m3 to second “`int`”	(for TCOND1)
TCOVER33:	apply m4 to add element before first “`int`”	(for TCOND1)
TCOVER34:	apply m4 to add element before “`e`”	(for TCOND1)
TCOVER35:	apply m4 to add element before second “`int`”	(for TCOND1)
TCOVER36:	apply m4 to add element after second “`int`”	(for TCOND1)
TCOVER37:	apply m4 to add element before first “`int`” and “`["+"|"-"]`”	(for TCOND1 & TCOND2)
TCOVER38:	apply m4 to add element between “`["+"|"-"]`” and first “`int`”	(for TCOND1 & TCOND2)
TCOVER39:	apply m4 to add element between first “`int`” and “`e`”	(for TCOND1)

[“+”|”-”] 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).

B.2.4.6 Step 4: Derive Test Cases (TD4)

Valid test cases are derived by selecting one or more options for inclusion in the current test case, identifying inputs to exercise the option(s) and determining the expected result (in this case, ‘check_res’). The resulting valid test cases are:

Table 22 — Valid test cases for syntax testing

Test Case	Input ‘float_in’	Test Cov. Item covered	Expected Result ‘check_res’
TC 1	3e2	TCOVER1	'valid'
TC 2	+2e+5	TCOVER2	'valid'
TC 3	-6e-7	TCOVER3	'valid'
TC 4	6e-2	TCOVER4	'valid'
TC 5	1234567890e3	TCOVER5	'valid'
TC 6	0e0	TCOVER6	'valid'
TC 7	1e1	TCOVER7	'valid'
TC 8	2e2	TCOVER8	'valid'
TC 9	3e3	TCOVER9	'valid'
TC 10	4e4	TCOVER10	'valid'
TC 11	5e5	TCOVER11	'valid'
TC 12	6e6	TCOVER12	'valid'
TC 13	7e7	TCOVER13	'valid'
TC 14	8e8	TCOVER14	'valid'
TC 15	9e9	TCOVER15	'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 failures are located.

Invalid test cases are derived by selecting one or more mutations for inclusion in the current test case, identifying inputs to exercise the mutation(s) and determining the expected result (in this case, ‘check_res’). The resulting invalid test cases are:

Table 23 — Invalid test cases for syntax testing

Test Case	Input ‘float_in’	Mutation	Test Cov. Item Covered	Expected Result ‘check_res’
TC 16	xe0	m1	TCOVER16	'invalid'
TC 17	0x0	m1	TCOVER17	'invalid'
TC 18	0ex	m1	TCOVER18	'invalid'
TC 19	x0e0	m1	TCOVER19	'invalid'
TC 20	+xe0	m1	TCOVER20	'invalid'
TC 21	ee0	m2	TCOVER21	'invalid'
TC 22	+e0	m2	TCOVER22	'invalid'
TC 23	000	m2	TCOVER23	'invalid'
TC 24	0+0	m2	TCOVER24	'invalid'
TC 25	0ee	m2	TCOVER25	'invalid'
TC 26	0e+	m2	TCOVER26	'invalid'
TC 27	e0e0	m2	TCOVER27	'invalid'
TC 28	+ee0	m2	TCOVER28	'invalid'
TC 29	++e0	m2	TCOVER29	'invalid'
TC 30	e0	m3	TCOVER30	'invalid'
TC 31	00	m3	TCOVER31	'invalid'
TC 32	0e	m3	TCOVER32	'invalid'
TC 33	y0e0	m4	TCOVER33	'invalid'
TC 34	0ye0	m4	TCOVER34	'invalid'
TC 35	0ey0	m4	TCOVER35	'invalid'
TC 36	0e0y	m4	TCOVER36	'invalid'
TC 37	y+0e0	m4	TCOVER37	'invalid'
TC 38	+y0e0	m4	TCOVER38	'invalid'
TC 39	+0ye0	m4	TCOVER39	'invalid'

Some of the mutations are indistinguishable from correctly formed expansions and these have been discarded. For example, the generic mutation m2 (substitute TCOND2 for TCOND4) generates correct syntax as m2 is “substitute an element with another defined element” and TCOND2 and TCOND4 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 TCOND4, 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.2.4.7 Step 5: Assemble Test Sets (TD5)

A decision may be made to assemble one test set for valid test cases and one for invalid test cases:

TS1: TEST CASE 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
TS2: TEST CASE 16, 17, 18, 19, 20, 21, 22, 23, 24, 24, 25, 26, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39

B.2.4.8 Step 6: Derive Test Procedures (TD6)

All test cases could be assembled into the one test procedure, starting with the valid test cases, and ending with the invalid test cases.

TP1: covering all test cases in TS1, followed by all test cases in TS2, in the order specified in the test sets.

B.2.4.9 Syntax Testing Coverage

As stated in clause 6.1.5, there is no approach for calculating test coverage item coverage for syntax testing.

B.2.5 Combinatorial Test Design Techniques

B.2.5.1 Introduction

The aim of combinatorial testing is to reduce the cost of testing by deriving a small (possibly minimal) number of test cases that cover the chosen set of parameters and input values of the test item. Combinatorial test design techniques provide the ability to derive test cases from input values that have previously been selected, such as through the application of other specification-based test design techniques like equivalence partitioning or boundary value analysis. Each technique will be demonstrated through the application of one example. Since each technique shares common steps in identifying feature sets and deriving test conditions, these steps are demonstrated once below for all techniques, and this is then followed by the steps of deriving test coverage items and test cases that are unique to each combinatorial technique.

B.2.5.2 Example

Consider the test basis for a test item *travel_preference*, which records the travel preferences of staff members of an organisation that travel to major capital cities for work purposes. Each set of travel preferences is chosen through three sets of radio buttons, which consist of the following input value choices:

Destination = Paris, London, Sydney

Class = First, Business, Economy

Seat = Aisle, Window

If a valid input combination is provided to the program it will output "Accept", otherwise it will output "Reject".

B.2.5.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set to be defined:

FS1: travel_preference function

B.2.5.4 Step 2: Derive Test Conditions (TD2)

Every combinatorial technique shares a common approach to deriving test conditions. That is, test conditions correspond to each parameter (P) of the test item taking on one specific value (V), resulting in one P-V pair. This is repeated until all parameters are paired with their corresponding values. For the example above, this results in the following P-V pairs:

TCOND1:	Destination – Paris	(for FS1)
TCOND2:	Destination – London	(for FS1)
TCOND3:	Destination – Sydney	(for FS1)
TCOND4:	Class – First	(for FS1)
TCOND5:	Class – Business	(for FS1)
TCOND6:	Class – Economy	(for FS1)
TCOND7:	Seat – Aisle	(for FS1)
TCOND8:	Seat – Window	(for FS1)

B.2.5.5 All Combinations

B.2.5.5.1 Step 3: Derive Test Coverage Items (TD3)

In all combinations testing, the test coverage items are the unique combinations of P-V pairs, made up of one P-V pair for each test item parameter. These P-V pairs were earlier identified as test conditions.

TCOVER1:	Destination – Paris,	Class – First,	Seat – Aisle	(for TCOND 1, 4, 7)
TCOVER2:	Destination – Paris,	Class – First,	Seat – Window	(for TCOND 1, 4, 8)
TCOVER3:	Destination – Paris,	Class – Business,	Seat – Aisle	(for TCOND 1, 5, 7)
TCOVER4:	Destination – Paris,	Class – Business,	Seat – Window	(for TCOND 1, 5, 8)
TCOVER5:	Destination – Paris,	Class – Economy,	Seat – Aisle	(for TCOND 1, 6, 7)
TCOVER6:	Destination – Paris,	Class – Economy,	Seat – Window	(for TCOND 1, 6, 8)
TCOVER7:	Destination – London,	Class – First,	Seat – Aisle	(for TCOND 2, 4, 7)
TCOVER8:	Destination – London,	Class – First,	Seat – Window	(for TCOND 2, 4, 8)
TCOVER9:	Destination – London,	Class – Business,	Seat – Aisle	(for TCOND 2, 5, 7)
TCOVER10:	Destination – London,	Class – Business,	Seat – Window	(for TCOND 2, 5, 8)
TCOVER11:	Destination – London,	Class – Economy,	Seat – Aisle	(for TCOND 2, 6, 7)
TCOVER12:	Destination – London,	Class – Economy,	Seat – Window	(for TCOND 2, 6, 8)
TCOVER13:	Destination – Sydney,	Class – First,	Seat – Aisle	(for TCOND 3, 4, 7)
TCOVER14:	Destination – Sydney,	Class – First,	Seat – Window	(for TCOND 3, 4, 8)
TCOVER15:	Destination – Sydney,	Class – Business,	Seat – Aisle	(for TCOND 3, 5, 7)
TCOVER16:	Destination – Sydney,	Class – Business,	Seat – Window	(for TCOND 3, 5, 8)
TCOVER17:	Destination – Sydney,	Class – Economy,	Seat – Aisle	(for TCOND 3, 6, 7)
TCOVER18:	Destination – Sydney,	Class – Economy,	Seat – Window	(for TCOND 3, 6, 8)

B.2.5.5.2 Step 4: Derive Test Cases (TD4)

Test cases are derived by selecting one P-V pair and combining it with every other P-V pair from all other parameters (where each combination creates exactly one test case), identifying arbitrary valid values to exercise any other input variable required by the test case, determining the expected result and repeating until the required coverage is achieved. In this example, this results in the following test cases:

Table 24 — Test cases for all combinations testing

Test Case #	Input Values			Expected Result	Test Coverage Item(s) Covered
	Destination	Class	Seat		
1	Paris	First	Aisle	Accept	TCOVER1
2	Paris	First	Window	Accept	TCOVER2
3	Paris	Business	Aisle	Accept	TCOVER3
4	Paris	Business	Window	Accept	TCOVER4
5	Paris	Economy	Aisle	Accept	TCOVER5
6	Paris	Economy	Window	Accept	TCOVER6
7	London	First	Aisle	Accept	TCOVER7
8	London	First	Window	Accept	TCOVER8
9	London	Business	Aisle	Accept	TCOVER9
10	London	Business	Window	Accept	TCOVER10
11	London	Economy	Aisle	Accept	TCOVER11
12	London	Economy	Window	Accept	TCOVER12
13	Sydney	First	Aisle	Accept	TCOVER13
14	Sydney	First	Window	Accept	TCOVER14
15	Sydney	Business	Aisle	Accept	TCOVER15
16	Sydney	Business	Window	Accept	TCOVER16
17	Sydney	Economy	Aisle	Accept	TCOVER17
18	Sydney	Economy	Window	Accept	TCOVER18

B.2.5.5.3 Step 5: Assemble Test Sets (TD5)

It may be decided that all test cases will be divided into those that cover aisle seats and window seats. This would result in the following test sets.

TS1: TEST CASES 1, 3, 5, 7, 9, 11, 13, 15, 17
TS2: TEST CASES 2, 4, 6, 8, 10, 12, 14, 16, 18

B.2.5.5.4 Step 6: Derive Test Procedures (TD6)

Since each test set is going to be executed by a different tester, they can be divided into two test procedures.

TP1: covering all test cases in TS1, in the order specified in the test set.
TP2: covering all test cases in TS2, in the order specified in the test set.

B.2.5.5.5 All Combinations Test Coverage

Using the formula provided in clause 6.1.6.1 and the test coverage items derived above:

$$Coverage_{(all-combinations)} = \frac{18}{18} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for all-combinations testing has been achieved.

B.2.5.6 Pair-wise Testing

B.2.5.6.1 Step 3: Derive Test Coverage Items (TD3)

In pair-wise testing, test coverage items are identified as the unique pairs of P-V pairs for different parameters. For the *travel_preference* example, the following test coverage items can be defined:

TCOVER1:	Paris, First	(for TCOND1, TCOND4)
TCOVER2:	Paris, Business	(for TCOND1, TCOND5)
TCOVER3:	Paris, Economy	(for TCOND1, TCOND6)
TCOVER4:	London, First	(for TCOND2, TCOND4)
TCOVER5:	London, Business	(for TCOND2, TCOND5)
TCOVER6:	London, Economy	(for TCOND2, TCOND6)
TCOVER7:	Sydney, First	(for TCOND3, TCOND4)
TCOVER8:	Sydney, Business	(for TCOND3, TCOND5)
TCOVER9:	Sydney, Economy	(for TCOND3, TCOND6)
TCOVER10:	Paris, Aisle	(for TCOND1, TCOND7)
TCOVER11:	Paris, Window	(for TCOND1, TCOND8)
TCOVER12:	London, Aisle	(for TCOND2, TCOND7)
TCOVER13:	London, Window	(for TCOND2, TCOND8)
TCOVER14:	Sydney, Aisle	(for TCOND3, TCOND7)
TCOVER15:	Sydney, Window	(for TCOND3, TCOND8)
TCOVER16:	First, Aisle	(for TCOND4, TCOND7)
TCOVER17:	First, Window	(for TCOND4, TCOND8)
TCOVER18:	Business, Aisle	(for TCOND5, TCOND7)
TCOVER19:	Business, Window	(for TCOND5, TCOND8)
TCOVER20:	Economy, Aisle	(for TCOND6, TCOND7)
TCOVER21:	Economy, Window	(for TCOND6, TCOND8)

B.2.5.6.2 Step 4: Derive Test Cases (TD4)

Test cases are derived by selecting one or more unique pairs of P-V pairs (test coverage items) for inclusion in the current test case, selecting arbitrary valid values for any other input variable required by the test case, determining the expected result of the test and repeating until all P-V pairs with different parameters are included in at least one test case. In this example, three P-V pairs can be included in all test cases.

Table 25 — Test cases pair-wise testing

Test Case #	Input Values			Expected Result	Test Coverage Item(s) Covered
	Destination	Class	Seat		
1	Paris	First	Aisle	Accept	TCOVER1, TCOVER10, TCOVER16
2	Paris	Business	Window	Accept	TCOVER2, TCOVER11, TCOVER19
3	Paris	Economy	Aisle	Accept	TCOVER3, TCOVER10, TCOVER20
4	London	First	Aisle	Accept	TCOVER4, TCOVER12, TCOVER16
5	London	Business	Window	Accept	TCOVER5, TCOVER13, TCOVER19
6	London	Economy	Aisle	Accept	TCOVER6, TCOVER12, TCOVER20
7	Sydney	First	Window	Accept	TCOVER7, TCOVER15, TCOVER17
8	Sydney	Business	Aisle	Accept	TCOVER8, TCOVER14, TCOVER18
9	Sydney	Economy	Window	Accept	TCOVER9, TCOVER15, TCOVER21

B.2.5.6.3 Step 5: Assemble Test Sets (TD5)

It may be decided since there are a relatively small number of test cases, that they can be combined into the one test set, as follows.

TS1: TEST CASES 1, 2, 3, 4, 5, 6, 7, 8, 9

B.2.5.6.4 Step 6: Derive Test Procedures (TD6)

Since there is only one test set, it can be combined into the one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

B.2.5.6.5 Pair-wise Test Coverage

Using the formula provided in clause 6.1.6.2 and the test coverage items derived above:

$$Coverage_{(pairwise)} = \frac{21}{21} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for pair-wise testing has been achieved.

B.2.5.7 Each Choice Testing

B.2.5.7.1 Step 3: Derive Test Coverage Items (TD3)

In each choice (or 1-wise) testing, the test coverage items are the set of P-V pairs. Thus, for the *travel_preference* example, the following test coverage items can be defined:

TCOVER1:	Destination – Paris	(for TCOND1)
TCOVER2:	Destination – London	(for TCOND2)
TCOVER3:	Destination – Sydney	(for TCOND3)
TCOVER4:	Class – First	(for TCOND4)
TCOVER5:	Class – Business	(for TCOND5)
TCOVER6:	Class – Economy	(for TCOND6)
TCOVER7:	Seat – Aisle	(for TCOND7)
TCOVER8:	Seat – Window	(for TCOND8)

B.2.5.7.2 Step 4: Derive Test Cases (TD4)

Each choice test cases are derived by selecting one or more P-V pairs for inclusion in the current test case, selecting arbitrary valid values for any other input variables required by the test case, determining the expected result and repeating until all P-V pairs are included in at least one test case. For this example, only three test cases are required:

Table 26 — Test cases for each choice testing

Test Case #	Input Values			Expected Result	Test Coverage Item(s) Covered
	Destination	Class	Seat		
1	Paris	First	Aisle	Accept	TCOVER1, TCOVER4, TCOVER7
2	London	Business	Window	Accept	TCOVER2, TCOVER5, TCOVER8
3	Sydney	Economy	Aisle	Accept	TCOVER3, TCOVER6, TCOVER7

Note that other test cases could be derived that would also achieve the required level of coverage.

B.2.5.7.3 Step 5: Assemble Test Sets (TD5)

Since a very small number of test cases derived in this example, it may be decided that they be combined into the one test set.

TS1: TEST CASES 1, 2, 3

B.2.5.7.4 Step 6: Derive Test Procedures (TD6)

Since all test cases are in the one test set, we can derive one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

B.2.5.7.5 Each Choice Test Coverage

Using the formula provided in clause 6.1.6.3 and the test coverage items derived above:

$$Coverage_{(each\_choice)} = \frac{8}{8} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for each choice testing has been achieved.

B.2.5.8 Base Choice Testing

B.2.5.8.1 Step 3: Derive Test Coverage Items (TD3)

Test coverage items for base choice testing are chosen by selecting a "base choice" value for each parameter. For example, the base choice could be chosen from the operational profile, from the main path in use case testing or from the test coverage items that are derived during equivalence partitioning. In this example, the operational profile may indicate that the following input values should be chosen as the base choice:

TCOVER1:	Destination – London,	Class – Economy,	Seat – Window	(covers TCOND2, TCOND6 & TCOND8)

The remaining test coverage items are derived by identifying all remaining P-V pairs:

TCOVER2:	Destination – Paris,	Class – Economy,	Seat – Window	(covers TCOND1, TCOND6 & TCOND8)

TCOVER3:	Destination – Sydney,	Class – Economy,	Seat – Window	(covers TCOND3, TCOND6 & TCOND8)
TCOVER4:	Destination – London,	Class – First,	Seat – Window	(covers TCOND2, TCOND4 & TCOND8)
TCOVER5:	Destination – London,	Class – Business,	Seat – Window	(covers TCOND2, TCOND5 & TCOND8)
TCOVER6:	Destination – London,	Class – Economy,	Seat – Aisle	(covers TCOND2, TCOND6 & TCOND7)

B.2.5.8.2 Step 4: Derive Test Cases (TD4)

A base-choice test case can now be derived by combining the test coverage items:

Base Choice: London, Economy, Window

This is shown as the first test case the table below. The remaining test cases can now be derived by substituting one P-V pair into the base-choice test case per test and repeating until all P-V pairs are covered:

Table 27 — Test cases for base choice testing

Test Case #	Input Values			Expected Result	Test Coverage Item(s) Covered
	Destination	Class	Seat		
1	London	Economy	Window	Accept	TCOVER1
2	Paris	Economy	Window	Accept	TCOVER2
3	Sydney	Economy	Window	Accept	TCOVER3
4	London	First	Window	Accept	TCOVER4
5	London	Business	Window	Accept	TCOVER5
6	London	Economy	Aisle	Accept	TCOVER6

B.2.5.8.3 Step 5: Assemble Test Sets (TD5)

Since a small number of test cases derived in this example, it may be decided that they be combined into the one test set.

TS1: TEST CASES 1, 2, 3, 4, 5, 6

B.2.5.8.4 Step 6: Derive Test Procedures (TD6)

Since all test cases are in the one test set, we can derive one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

B.2.5.8.5 Base Choice Test Coverage

Using the formula provided in clause 6.1.6.4 and the test coverage items derived above:

$$Coverage_{(base\_choice)} = \frac{6}{6} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for base choice testing has been achieved.

B.2.6 Decision Table Testing

B.2.6.1 Introduction

The aim of decision table testing is to derive a set of test cases that cover the logical associations between inputs and outputs (which is represented as a series of conditions and actions) associated by decision rules according to the chosen level of condition and action coverage.

B.2.6.2 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 test basis:

> *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).*

B.2.6.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set to be defined:

FS1: cheque debit function

B.2.6.4 Step 2: Derive Test Conditions (TD2)

The test conditions are the conditions and actions that can be derived from the test basis.

The conditions are:

TCOND1 (C1):	New balance in credit	(for FS1)
TCOND2 (C2):	New balance overdraft, but within authorised limit	(for FS1)
TCOND3 (C3):	Account is postal	(for FS1)

The actions are:

TCOND4 (A1):	Process debit	(for FS1)
TCOND5 (A2):	Suspend account	(for FS1)
TCOND6 (A3):	Send out letter	(for FS1)

B.2.6.5 Step 3: Derive Test Coverage Items (TD3)

The decision table enables identification of test coverage items as decision rules in the decision table. Each column of the decision table is a decision rule. Decision tables may be also be represented with the decision rules in rows rather than columns. The table comprises two parts. In the first part each decision rule is tabulated against the conditions. A 'T' indicates that the condition must be TRUE for the decision rule to apply and an 'F' indicates that the condition must be FALSE for the decision rule to apply. In the second part, each decision 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 decision rule. Two or more columns may be combined if they contain a Boolean condition that does not affect the outcome regardless of its value.

The example has the following decision table, which identifies 8 test coverage items:

Table 28 — Decision table of the cheque debit function

Decision 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	*	*

NOTE 1 Although “T” and “F” have been used in the above decision table to denote “True” and “False”, other notations could be used (e.g. the words “true” and “false” could be used instead).

NOTE 2 In the table above, both conditions and actions are binary (T or F) conditions, which results in a “limited-entry” decision table. In “extended entry” decision tables, conditions and/or actions can assume multiple values.

B.2.6.6 Step 4: Derive Test Cases (TD4)

Test cases are derived by selecting one or more feasible decision rules from the decision table at a time that have not yet been covered by a test case, identifying inputs to exercise the condition(s) and actions(s) of the decision rule and arbitrary valid values for any other input variables required by the test case, determining the expected result and repeating these steps until the required level of test coverage is achieved. The following test cases would be required to achieve 100% decision table coverage, and correspond to the decision rules in the decision table above (no test cases are derived for decision rules 7 and 8 since they are infeasible):

Table 29 — Test case table of the cheque debit function

Test Case	CAUSES/INPUTS				EFFECTS/RESULTS		Test Coverage Item Covered
	account type	overdraft limit	current balance	debit amount	New balance	action code	
1	'c'	£100	-£70	£50	-£70	'L'	1
2	'p'	£1500	£420	£2000	£420	'S&L'	2
3	'c'	£250	£650	£800	-£150	'D&L'	3
4	'p'	£750	-£500	£200	-£700	'D&L'	4
5	'c'	£1000	£2100	£1200	£900	'D'	5
6	'p'	£500	£250	£150	£100	'D&L'	6

B.2.6.7 Step 5: Assemble Test Sets (TD5)

Since there are only six test cases required to cover all decision rules, it may be decided that all test cases will be manual and will all be placed in the one test set.

TS1: TEST CASE 1, 2, 3, 4, 5, 6

B.2.6.8 Step 6: Derive Test Procedures (TD6)

Since all test cases are in the one test set, we can derive one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

B.2.6.9 Decision Table Test Coverage

Using the formula provided in clause 6.1.7 and the test coverage items derived above:

$$Coverage_{(decision\_table\_testing)} = \frac{6}{6} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for decision table testing has been achieved.

B.2.7 Cause-Effect Graphing

B.2.7.1 Introduction

The aim of cause-effect graphing is to derive test cases that cover the logical relationships between causes (e.g. inputs) and effects (e.g. outputs) of a test item according to a chosen level of coverage. The technique utilises a notation that allows a cause-effect graph of the test item to be designed that illustrates relationships between causes and effects as well as explicit constraints placed on causes and effects. This differs from decision table testing in which constraints are not explicitly stated. Naturally, the technique is only effective to the extent that the model captures the test basis of the test item.

B.2.7.2 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 test basis:

> *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).*

B.2.7.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set to be defined:

FS1: cheque debit function

B.2.7.4 Step 2: Derive Test Conditions (TD2)

The test conditions are the causes and effects that can be derived from the test basis.

The causes are:

TCOND1 (C1):	New balance in credit	(for FS1)
TCOND2 (C2):	New balance overdraft, but within authorised limit	(for FS1)
TCOND3 (C3):	Account is postal	(for FS1)

The effects are:

TCOND4 (A1):	Process debit	(for FS1)
TCOND5 (A2):	Suspend account	(for FS1)
TCOND6 (A3):	Send out letter	(for FS1)

A cause-effect graph shows the relationship between the causes and effects in a notation similar to that used by designers of hardware logic circuits. The test basis is modelled by the graph shown below.

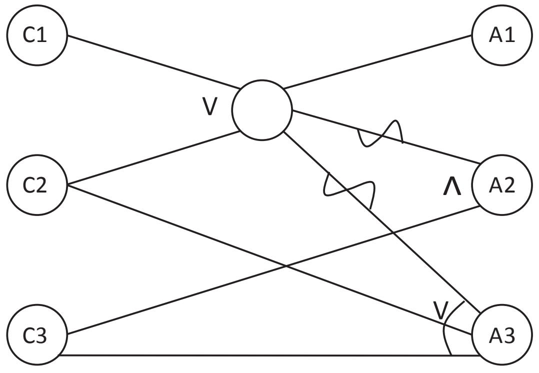
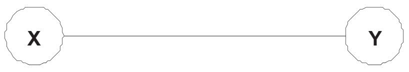


Figure 13 — Cause-effect graph of the cheque debit function (see below for notation)

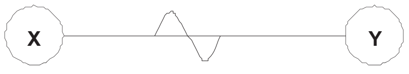
NOTE 1 The "empty" node that connects C1/C2 to A1/A2/A3 is a connector node that is used to group together two or more causes.

Identity



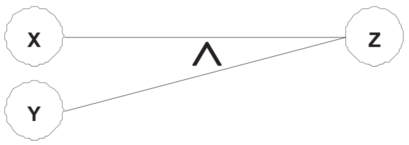
Node Y is true only if X is true
If X = T then Y = T else Y = F

Not



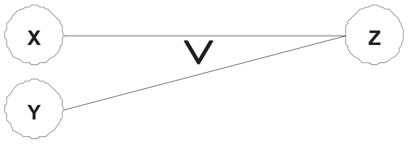
Node Y is true only if X is false
If X = F then Y = T else Y = F

And



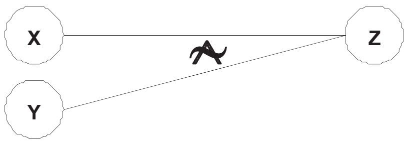
Node Z is true only if both X and Y are true
If X = T and Y = T then Z = T else Z = F

Or



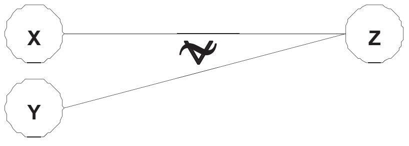
Node Z is true only if either X or Y are true
If X = T or Y = T then Z = T else Z = F

Nand



Node Z is true only if either X or Y or both are false
If X = F or Y = F then Z = F else Z = T

Nor



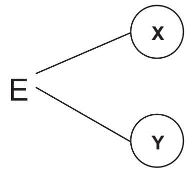
Node Z is true only if neither X nor Y are true
If X = T or Y = T then Z = F else Z = T

Figure 14 — Basic notation for cause-effect graphing

NOTE 2 Although the following “constraint” notations are not required for the example demonstrated in this clause, they are included here as they confer advantages in identifying required, permitted and forbidden relationships between causes and relationships between effects. Such constraint relationships are not explicitly stated in decision tables and are often implicit in specifications. These notations provide means for verifying both the integrity of the cause-effect graph and the decision table and test cases that are derived from it.

Cause Constraints

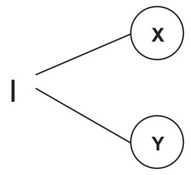
Exclusive



Cause X and cause Y cannot be simultaneously true

If X = 1 then Y = 0, if Y = 1 then X = 0, can be simultaneouslv false

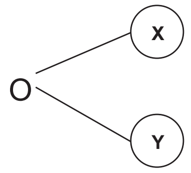
Inclusive



Cause X and cause Y cannot be simultaneously false

If X = 0 then Y = 1, if Y = 0 then X = 1, can be simultaneouslv true

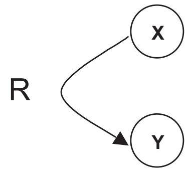
One and onlv one



One and only one of cause X and cause Y must be true

If X = 1 then Y = 0, if Y = 1 then X = 0, cannot be simultaneouslv true or false

Requires

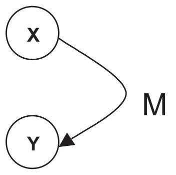


Cause Y must be true whenever cause X is true

If X = 1 then Y = 1, if X = 0 then Y = 1 or Y = 0

Effect Constraints

Masks



Effect Y will be forced false whenever effect X is true

If X = 1 then Y = 0

Figure 15 — Notation for representing cause and effect constraints in cause-effect graphing

B.2.7.5 Step 3: Derive Test Coverage Items (TD3)

The cause-effect graph is then recast in terms of a decision table (e.g. see for example (Myers 1979) and (Nursimulu and Probert 1995)), which enables identification of the test coverage items (i.e. the feasible decision rules in the decision table). Each column of the decision table is a decision rule. The table comprises two parts. In the first part each decision rule is tabulated against the causes. A 'T' indicates that the cause must be TRUE for the decision rule to apply and an 'F' indicates that the cause must be FALSE for the decision rule to apply. In the second part, each decision rule is tabulated against the effects. A 'T' indicates that the effect will occur; an 'F' indicates that the effect will not occur; an asterisk (*) indicates that the combination of causes is infeasible and so no effects are defined for the decision rule.

The example has the following decision table, which identifies six test coverage items (decision rules 7 and 8 are not test coverage items since they are infeasible):

Table 30 — Decision table of the cheque debit function

Decision 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	*	*

B.2.7.6 Step 4: Derive Test Cases (TD4)

Test cases are derived by selecting one or more feasible decision rules from the decision table that have not been included in a test case, identifying inputs to exercise the causes(s) and effects(s) of the decision rule and arbitrary valid values for any other input variable required by the test case, determining the expected result of the test case, and repeating these steps until all feasible decision rules are covered. The following test cases achieve 100% cause-effect coverage and correspond to the decision rules in the decision table above (no test cases are generated for decision rules 7 and 8 as they are infeasible):

Table 31 — Test case table of the cheque debit function

	CAUSES/INPUTS				EFFECTS/RESULTS		
Test Case	account type	overdraft limit	current balance	debit amount	New balance	action code	Test Coverage Item Covered
1	'c'	£100	-£70	£50	-£70	'L'	1
2	'p'	£1500	£420	£2000	£420	'S&L'	2
3	'c'	£250	£650	£800	-£150	'D&L'	3
4	'p'	£750	-£500	£200	-£700	'D&L'	4
5	'c'	£1000	£2100	£1200	£900	'D'	5
6	'p'	£500	£250	£150	£100	'D&L'	6

B.2.7.7 Step 5: Assemble Test Sets (TD5)

Since there are only six test cases required to cover all decision rules, it may be decided that all test cases will be manual and will all be placed in the one test set.

TS1: TEST CASE 1, 2, 3, 4, 5, 6

B.2.7.8 Step 6: Derive Test Procedures (TD6)

Since all test cases are in the one test set, we can derive one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

B.2.7.9 Cause-Effect Graphing Test Coverage

Using the formula provided in clause 6.1.8 and the test coverage items derived above:

$$Coverage_{(cause-effect-graphing)} = \frac{6}{6} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for cause-effect graphing has been achieved.

B.2.8 State Transition Testing

B.2.8.1 Introduction

The aim of state transition testing is to derive a set of test cases that cover transitions and/or states of the test item according to the chosen level of coverage. The technique is based upon an analysis of the test basis of the test item to model its behaviour by state transitions.

B.2.8.2 Example

Consider a test item, *manage_display_changes*, with the following test basis:

The test item 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'.

B.2.8.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature needs to be defined:

FS1: *manage_display_changes*

B.2.8.4 Step 2: Derive Test Conditions (TD2)

A state model is produced as the test condition. State transition diagrams (STD) are commonly used as state models and their notation is illustrated below. A STD consists of states, transition, events and actions (see Figure 16). 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 test item. A transition is determined by the current state and an event and is normally labelled simply with the event and action.

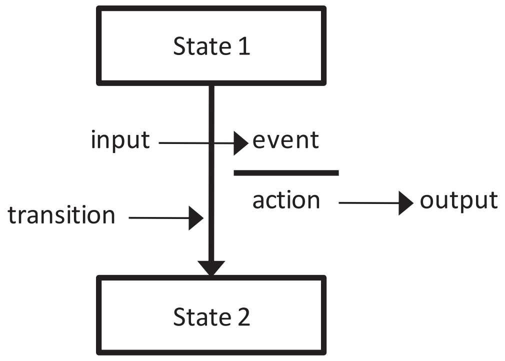


Figure 16 — Generic state model

The STD for the test item *manage_display_changes* is as follows (which represents test condition TCOND1):

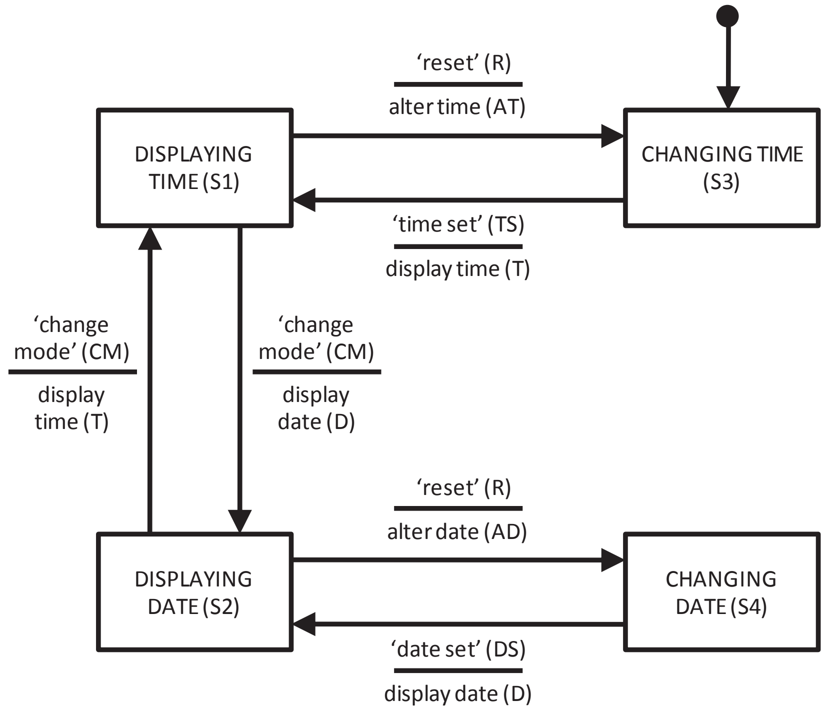


Figure 17 — State transition diagram for manage_display_changes

B.2.8.5 Step 3: Derive Test Coverage Items - 0-Switch and "All Transitions" Testing (TD3)

Assuming the chosen level of coverage is "all transitions", a state table can be drawn to represent all valid and invalid transitions (the required test coverage items). To achieve full 0-switch coverage only the valid transitions need to be exercised.

A limitation of 0-switch coverage is that the tests are derived to exercise only the valid transitions in the test item. A more thorough test of the test item will *also* attempt to cause invalid transitions to occur ("all transitions"). The STD only explicitly shows the valid transitions (all transitions not shown are considered invalid). One example of a state model that explicitly shows both valid and invalid transitions is a state table, while an alternate representation is a state transition diagram that includes an "anomalous" state at which all invalid transitions terminate. One notation used for state tables is briefly described below:

Table 32 — State table notation

	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 or Action for the given start state and input.

The state table for *manage_display_changes* is shown below:

Table 33 — State table for manage_display_changes

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

Any entry where the state remains the same and the action is shown as null (–) 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 coverage of valid test coverage items (0-switch). Thus a more complete test set ("all transitions") will test both possible transitions and null transitions, which means testing the response of the test item to all inputs specified in the test basis in all possible states. The state table provides an ideal means of directly deriving test coverage items to cover null transitions (for "all transition" coverage).

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 coverage items for "all transitions" coverage, which can be read from the state table as shown below:

Table 34 — State table to test case table mapping for manage_display_changes

	CM	R	TS	DS
S1	S2/D (TCOVER1)	S3/AT (TCOVER2)	S1/– (TCOVER3)	S1/– (TCOVER4)
S2	S1/T (TCOVER5)	S4/AD (TCOVER6)	S2/– (TCOVER7)	S2/– (TCOVER8)
S3	S3/– (TCOVER9)	S3/– (TCOVER10)	S1/T (TCOVER11)	S3/– (TCOVER12)
S4	S4/– (TCOVER13)	S4/– (TCOVER14)	S4/– (TCOVER15)	S2/D (TCOVER16)

Thus, the following (valid and invalid) test coverage items were identified from the state table above for "all transitions" coverage:

TCOVER1:	S1 to S2 with input CM	(for FS1, valid transition)
TCOVER2:	S1 to S3 with input R	(for FS1, valid transition)
TCOVER3:	S1 to S1 with input TS	(for FS1, invalid transition)
TCOVER4:	S1 to S1 with input DS	(for FS1, invalid transition)
TCOVER5:	S2 to S1 with input T	(for FS1, valid transition)
TCOVER6:	S2 to S4 with input R	(for FS1, valid transition)
TCOVER7:	S2 to S2 with input TS	(for FS1, invalid transition)
TCOVER8:	S2 to S2 with input DS	(for FS1, invalid transition)
TCOVER9:	S3 to S3 with input CM	(for FS1, invalid transition)
TCOVER10:	S3 to S3 with input R	(for FS1, invalid transition)
TCOVER11:	S3 to S1 with input TS	(for FS1, valid transition)
TCOVER12:	S3 to S3 with input DS	(for FS1, invalid transition)
TCOVER13:	S4 to S4 with input CM	(for FS1, invalid transition)
TCOVER14:	S4 to S4 with input R	(for FS1, invalid transition)
TCOVER15:	S4 to S4 with input TS	(for FS1, invalid transition)
TCOVER16:	S4 to S2 with input DS	(for FS1, valid transition)

B.2.8.6 Step 4: Derive Valid Test Cases (TD4)

B.2.8.6.1 Options

Test cases can now be derived to exercise each of the possible transitions (using the abbreviated STD labels). Input(s) to exercise the transition(s) that are to be covered by each test case can be determined from the STD, as can the expected result, which can be determined by combining the expected output and final state of the transition in the STD. Test cases may be derived to cover one to *n* transitions per test case, where *n* is the maximum number of transitions possible. For example, test cases could be derived for 0-switch or 1-switch coverage (although in practice it would not be necessary to derive both 0-switch and 1-switch test cases, this is demonstrated below simply to explain the approach). Test cases can also be derived to cover invalid transitions. These three scenarios are demonstrated below.

B.2.8.6.2 Step 4a: Derive 0-Switch Test Cases (Valid Transitions)

The following six test cases provide 0-switch test coverage. Each test case is derived by selecting a transition and identifying the inputs, expected output and final state from the STD until all transitions are covered by one test case.

Table 35 — 0-switch test cases for manage_display_changes

Test Case	1	2	3	4	5	6
Start State	S1	S1	S2	S2	S3	S4
Input	CM	R	CM	R	TS	DS
Expected Output	D	AT	T	AD	T	D
Finish State	S2	S3	S1	S4	S1	S2
Test Coverage Items Covered	1	2	5	6	11	16

NOTE A test procedure could be written for the six text cases in the table above that would allow them to be executed sequentially so that the "Finish State" for one test case is the start state of the next (e.g. execution order 5, 1, 4, 6, 3, 2). This is elaborated on in step 5.

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).

These six test cases exercise each of the "valid" transitions and so achieves 0-switch coverage (Cho 1987). 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.

B.2.8.6.2.1 Step 4b: Derive Test Cases for Invalid Transitions

Test cases to cover invalid transitions can now be defined as follows:

Table 36 — invalid test cases for manage_display_changes

Test Case	7	8	9	10	11	12	13	14	15	16
Start State	S1	S1	S2	S2	S3	S3	S3	S4	S4	S4
Input	TS	DS	TS	DS	CM	R	DS	CM	R	TS
Exp. Output	N	N	N	N	N	N	N	N	N	N
Finish State	S1	S1	S2	S2	S3	S3	S3	S4	S4	S4
Test Cov. Items Covered	3	4	7	8	9	10	12	13	14	15

As the table above shows, test cases that cover invalid test coverage items should not cause transition away from the starting state. The test cases in the above two tables combined will achieve "all transitions" coverage.

B.2.8.6.2.2 Step 4c: Derive Test Coverage Items - 1-Switch Testing (TD3)

The following test coverage items can be derived from the STD to achieve 1-switch coverage:

TCOVER17:	S1 to S2 to S1 with inputs CM and CM	(for FS1)
TCOVER18:	S1 to S2 to S4 with inputs CM and R	(for FS1)
TCOVER19:	S1 to S3 to S1 with inputs R and TS	(for FS1)
TCOVER20:	S3 to S1 to S2 with inputs TS and CM	(for FS1)
TCOVER21:	S3 to S1 to S3 with inputs TS and R	(for FS1)
TCOVER22:	S2 to S1 to S2 with inputs CM and CM	(for FS1)
TCOVER23:	S2 to S1 to S3 with inputs CM and R	(for FS1)
TCOVER24:	S2 to S4 to S2 with inputs R and DS	(for FS1)
TCOVER25:	S4 to S2 to S1 with inputs DS and CM	(for FS1)
TCOVER26:	S4 to S2 to S4 with inputs DS and R	(for FS1)

B.2.8.6.2.3 Step 4d: Derive 1-Switch Test Cases (TD4)

If the test coverage chosen in step TD3 was to cover all 1-switch transitions, then test cases could be written to exercise all possible sequential pairs of transitions. In this example, there are ten, as follows:

Table 37 — 1-switch test cases for manage_display_changes

Test Case	17	18	19	20	21	22	23	24	25	26
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
Test Cov. Items Covered	17	18	19	20	21	22	23	24	25	26

This indicates that test case 17 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 the 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.

B.2.8.6.3 Step 5: Assemble Test Sets (TD5)

If may be decided that all 0-switch test cases (covering valid transitions) will be assembled into one test set, the "all transitions" test cases covering invalid transitions in another test set, as follows:

TS1: 0-switch test cases	– TEST CASES 1, 2, 3, 4, 5, 6.
TS2: "all transitions" INVALID test cases	– TEST CASES 7, 8, 9, 10, 11, 12, 13, 14, 15, 16.
TS3: 1-switch test cases	– TEST CASES 17, 18, 19, 20, 21, 22, 23, 24, 25, 26.

NOTE 1 In some instances, it is possible to order individual test cases such that the "Finish State" for one test case is the starting state of the next. This can improve efficiency during test execution. The possibility of doing this depends on the specific state model being tested. In the example above, the following ordering of test cases would achieve this objective: TS1: 0 switch test cases – TEST CASES 5, 1, 4, 6, 3, 2.

NOTE 2 Since the 0-switch test cases defined above cover all paths in the 1-switch test cases, it is unnecessary to define test sets and test procedures for the 1-switch test cases. However, they are included here for completeness.

B.2.8.6.4 Step 6: Derive Test Procedures (TD6)

One test procedure could be defined to execute all test cases in the order that they are defined in step 5:

TP1: covering all test cases in TS1, TS2 and TS3, in the order they are defined in the test sets.

B.2.8.6.5 State Transition Testing Coverage

Using the formula provided in clause 6.1.9 and the test coverage items derived above:

$$Coverage_{(0-switch\_coverage)} = \frac{6}{6} \times 100\% = 100\%$$

$$Coverage_{(all\_transitions\_coverage)} = \frac{16}{16} \times 100\% = 100\%$$

$$Coverage_{(1-switch\_coverage)} = \frac{10}{10} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for 0-switch testing and all-transitions testing has been achieved.

B.2.9 Scenario Testing

B.2.9.1 Introduction

The aim of scenario testing is to derive test cases that cover the scenarios of the test item according to the chosen level of coverage. Scenario testing is based upon an analysis of the test basis to produce a model of its behaviour in terms of sequences of actions that constitute workflows through the test item.

B.2.9.2 Example

Consider a test item *withdraw_cash* that forms part of the system that drives an Automated Teller Machine (ATM), and which has the following test basis:

The withdraw_cash function allows customers with bank accounts to withdraw funds from their account via an ATM. A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM. After the withdrawal is complete, the account balance is debited by the withdrawn amount, a receipt for the withdrawal is printed, and the ATM is available and ready for the next user.

The following scenarios have been specified as being required by the customer:

Typical Scenario

— *Successful withdrawal of funds from account.*

Alternate Scenarios

Disapproval of a withdrawal, because:

— *the user's bank card is rejected as it is unrecognised by the ATM*

— *the user enters their PIN incorrectly up to 2 times*

— *the user enters their PIN incorrectly three or more times, with the ATM retaining the card*

— *the user selects deposit or transfer instead of withdrawal*

— *the user selects an incorrect account that does not exist on the entered card*

— *the withdrawal amount entered by the user is invalid*
— *there is insufficient cash in the ATM*
— *the user enters a non-dispensable amount*
— *the user enters an amount that exceeds their daily allowance*
— *there are insufficient funds in the user's bank account*

B.2.9.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: *withdraw_cash* function

B.2.9.4 Step 2: Derive Test Conditions (TD2)

To enable the identification of test conditions, a model of the test item must be produced that identifies the activities present in each scenario. The example model below is a flow-of-events diagram. In this notation, the "main" path is represented as a thick black line, the start and end points of the workflow are labelled, and each action is tagged with a unique identifier that designates it as a user (U) or system (S) (i.e. test item) action.

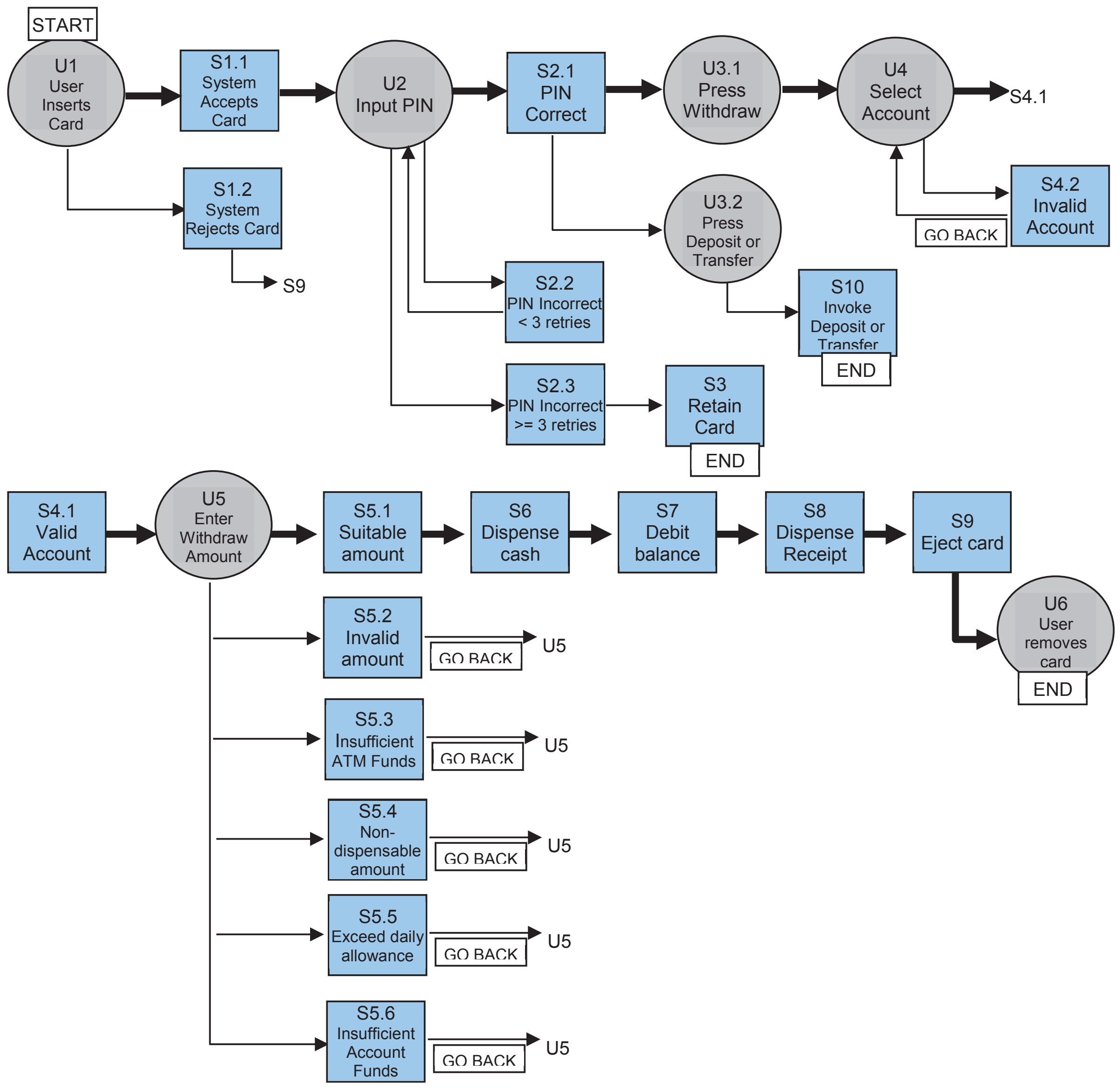


Figure 18 — Flow of events diagram for *withdraw_cash* function

In scenario testing, the test conditions are the main and alternate scenarios that are to be covered during testing (i.e. they are the sequences of user and system interactions through the flow of events diagram that constitutes one scenario). There were 11 scenarios described in the example, including one main and ten alternate. These can be described as test conditions (covering FS1) as follows:

TCOND1: Successful withdrawal of funds	(covers U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.1, S6, S7, S8, S9, U6)
TCOND2: User's card is unrecognised by ATM	(covers U1, S1.2, S9)
TCOND3: User enters PIN incorrectly < 3 times	(covers U1, S1.1, U2, S2.2)
TCOND4: User enters pin incorrectly 3 times	(covers U1, S1.1, U2, S2.2, U 2, S2.3, S3)
TCOND5: Insufficient cash in the ATM	(covers U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.3)
TCOND6: Insufficient funds in user's account	(covers U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.6)
TCOND7: Withdrawal amount is invalid	(covers U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.2)

There are also a number of other alternate scenarios that were identified by building the model of the expected behaviour of the test item. These can be defined as test conditions as follows.

TCOND8:	User enters PIN incorrectly > 3 times	(covers U1, S1.1, U2, S2.2, U2, S2.2, U2, S2.3, S3)
TCOND9:	User enters non-dispensable amount	(e.g. requests $20 when ATM only contains $50 notes) (covers U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.4)
TCOND10:	User enters amount exceeding daily allowance	(covers U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.5)
TCOND11:	User chooses deposit or transfer	(covers U1, S1.1, U2, S2.1, U3.2, S10)

B.2.9.5 Step 3: Derive Test Coverage Items (TD3)

In scenario testing, the test conditions are the typical and alternate scenarios, which are the same as the test coverage items.

TCOVER1 = TCOND1
TCOVER2 = TCOND2
TCOVER3 = TCOND3
TCOVER4 = TCOND4
TCOVER5 = TCOND5
TCOVER6 = TCOND6
TCOVER7 = TCOND7
TCOVER8 = TCOND8
TCOVER9 = TCOND9
TCOVER10 = TCOND10
TCOVER11 = TCOND11

B.2.9.6 Step 4: Derive Test Cases (TD4)

In scenario testing, test cases are derived by selecting a scenario to cover, identifying inputs to exercise the path covered by the test case, determining the expected result of the test and repeating until all scenarios are covered as required. The steps of the test case are typically worded in natural language format. If we assume that one test case is required to cover each test coverage item that was identified, the following test cases could be derived.

NOTE 1 Within each test case, there are a wide variety of input values that could be chosen to populate each input field. Equivalence Partitioning may be used to derive a set of valid and invalid values for populating each input field.

NOTE 2 The test cases below are each contained in a separate table for readability. In practice, they could be contained in one table.

Table 38 — Test cases for scenario testing

Test Case #	1
Test Case Name	Successful withdrawal of funds from account
Scenario Path Exercised	U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.1, S6, S7, S8, S9, U6
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	ATM Balance – $50,000
	Customer Account Balance – $100
	Withdrawal amount – $50
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	Withdrawal has successfully been made from customer account
	ATM balance is $49,950
	Customer account balance is $50
	ATM is open, operational and awaiting a customer card as input
Test Cov. Items Covered	TCOVER1

Table 39 — Test cases for scenario testing continued

Test Case #	2
Test Case Name	User's bank card is unrecognised by the ATM
Scenario Path Exercised	U1, S1.2, S9
Input	Invalid card
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	Card is rejected by the ATM
Test Cov. Items Covered	TCOVER2

Table 40 — Test cases for scenario testing continued

Test Case #	3
Test Case Name	User enters PIN incorrectly < 3 times
Scenario Path Exercised	U1, S1.1, U2, S2,2
Input	Valid card with valid customer account – assume 293910982246 is valid
	Invalid PIN entered twice – assume 0000 is invalid and does not match card
	ATM Balance – $100
	Customer Account Balance – $500
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	PIN is rejected by the ATM
Test Cov. Items Covered	TCOVER3

Table 41 — Test cases for scenario testing continued

Test Case #	4
Test Case Name	User enters PIN incorrectly 3 times
Scenario Path Exercised	U1, S1.1, U2, S2,2, U2, U3
Input	Valid card with valid customer account – assume 293910982246 is valid
	Invalid PIN entered three times – assume 0000 is invalid and does not match card
	ATM Balance – $100
	Customer Account Balance – $500
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	PIN is rejected by the ATM and ATM retains card
Test Cov. Items Covered	TCOVER4

Table 42 — Test cases for scenario testing continued

Test Case #	5
Test Case Name	Insufficient cash in the ATM
Scenario Path Exercised	U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.3
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	ATM Balance – $100
	Customer Account Balance – $500
	Withdrawal amount – $200
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	System displays message indicating that there are insufficient funds available in the ATM, and prompts the user to enter a new amount
Test Cov. Items Covered	TCOVER5

Table 43 — Test cases for scenario testing continued

Test Case #	6
Test Case Name	Insufficient funds in the user's bank account
Scenario Path Exercised	U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.6
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	ATM Balance – $50,000
	Customer Account Balance – $20
	Withdrawal amount – $50
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	System displays message indicating that there are insufficient funds available in user's bank account, and prompts user to enter a new amount
Test Cov. Items Covered	TCOVER6

Table 44 — Test cases for scenario testing continued

Test Case #	7
Test Case Name	Withdrawal amount entered by user is invalid
Scenario Path Exercised	U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.2
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	ATM Balance – $100
	Customer Account Balance – $20
	Withdrawal amount – $17
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	System displays message indicating that the amount entered is invalid, and prompts the user to enter a new amount
Test Cove Items Covered	TCOVER7

Table 45 — Test cases for scenario testing continued

Test Case #	8
Test Case Name	User enters PIN incorrectly > 3 times
Scenario Path Exercised	U1, S1.1, U2, S2.2, U2, S2.2, U2, S2.3, S3
Input	Valid card with valid customer account – assume 293910982246 is valid
	Invalid PIN – assume 1234 is invalid and does not match card
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	Each time invalid PIN is entered, system displays a message indicating that PIN is incorrect and prompts user to enter PIN again. On the third try, system retains the card and displays message indicating that card has been retained and that user should contact their bank to recover it.
Test Cov. Items Covered	TCOVER8

Table 46 — Test cases for scenario testing continued

Test Case #	9
Test Case Name	User enters non-dispensable amount
Scenario Path Exercised	U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.4
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	ATM Balance – $100, and ATM only contains $50 notes
	Customer Account Balance – $1,000
	Withdrawal amount – $20
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	System displays message indicating that denomination entered cannot be dispense by the ATM, and prompts the user to enter a new amount
Test Cov. Items Covered	TCOVER9

Table 47 — Test cases for scenario testing continued

Test Case #	10
Test Case Name	User enters amount exceeding their daily allowance
Scenario Path Exercised	U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.5
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	ATM Balance – $100, and ATM only contains $50 notes
	Customer Account Balance – $3,000
	Customer Maximum Daily Allowance – $1,000
	Withdrawal amount – $2,000
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	System displays message indicating that the amount entered exceeds the user’s daily allowance, and prompts the user to enter a new amount
Test Cov. Items Covered	TCOVER10

Table 48 — Test cases for scenario testing continued

Test Case #	11
Test Case Name	User chooses deposit or transfer
Scenario Path Exercised	U1, S1.1, U2, S2.1, U3.2, S10
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	Transaction Type – Deposit
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected Result	System displays message prompting the user to enter deposit details
Test Cov. Items Covered	TCOVER11

B.2.9.7 Step 5: Assemble Test Sets (TD5)

Tests could be grouped according to whether they cover the main or alternate scenarios:

TS1: TEST CASE 1.
TS2: TEST CASES 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

B.2.9.8 Step 6: Derive Test Procedures (TD6)

Only one procedure is required as follows:

TP1: covering the test case in TS1, in the order specified in the test set.
TP2: covering the test cases in TS2, in the order specified in the test set.

B.2.9.9 Scenario Test Coverage

Using the formula provided in clause 6.1.10 and the test coverage items derived above:

$$Coverage_{(scenario)} = \frac{11}{11} \times 100\% = 100\%$$

B.2.10 Use Case Testing

B.2.10.1 Introduction

Use case testing is a form of scenario testing, in which test case derivation is based on a use case model of the test item. It is demonstrated here using a separate example to provide users of this standard with a fully worked example of this technique.

B.2.10.2 Example

Consider the following example use case for a test item *change_password*:

Table 49 — Example use case for *change_password*

Use Case ID	UC001	
Use Case	change_password	
Purpose	To allow a user to change their existing password to a new password	
Actors	User	
Description	This user case allows users to change their current password to a new password.	
Trigger	User clicks Change Password button on the Main Menu screen	
Preconditions	User must already be logged into the system	
Scenario Name	**Step**	**Action**
Main Flow	1	User clicks Change Password button
	2	System displays Change Password screen
	3	User enters their current password correctly
	4	User enters their new password correctly
	5	User re-enters their new password correctly
	6	User clicks OK
	7	System displays message “Password changed successfully”
	8	System returns the user to the screen they were viewing before step 1
Alternate Flow – Existing Password Incorrect	3.1	User enters their current password incorrectly
	3.2	User clicks OK
	3.3	System displays an error message “Current password entered incorrectly. Please try again.” and highlights all text in the Current Password field
Alternate Flow – New Password Less Than 8 Characters	4.1.1	User enters a new password that is less than 8 characters long
	4.1.2	User clicks OK
	4.1.3	System displays an error message “New password must be at least 8 characters long. Please try again.”
Alternate Flow – New Password Same as Current Password	4.2.1	User enters a new password that is the same is their current password
	4.2.2	User clicks OK
	4.2.3	System displays error message “New password must not be the same as current password. Please try again.”
Alternate Flow – New Passwords Do Not Match	5.1	User re-enters new password that does not match the new password they entered at step 4
	5.2	User clicks OK
	5.3	System displays error message “New passwords do not match. Please try again.”
Variants and Exceptions	None	
Rules	New password must be different from current password New password must be at least 8 characters long System will mask all current and new password characters with an asterisk (*)	
Safely	None	
Frequency	Used the first time a new user logs into the system Typically used twice per user per year on average	

B.2.10.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: *change_password* function

B.2.10.4 Step 2: Derive Test Conditions (TD2)

Test conditions may be the typical and alternate scenarios present in the use case. In this example they are:

TCOND1:	Typical Flow	(for FS1)
TCOND2:	Alternate Flow – Existing Password Incorrect	(for FS1)
TCOND3:	Alternate Flow – New Password Less Than 8 Characters	(for FS1)
TCOND4:	Alternate Flow – New Password Same as Current Password	(for FS1)
TCOND5:	Alternate Flow – New Passwords Do Not Match	(for FS1)

B.2.10.5 Step 3: Derive Test Coverage Items (TD3)

The test coverage items in use case testing are the typical and alternate scenarios, as follows.

TCOVER1:	Typical Flow	(for TCOND1)
TCOVER2:	Alternate Flow – Existing Password Incorrect	(for TCOND2)
TCOVER3:	Alternate Flow – New Password Less Than 8 Characters	(for TCOND3)
TCOVER4:	Alternate Flow – New Password Same as Current Password	(for TCOND4)
TCOVER5:	Alternate Flow – New Passwords Do Not Match	(for TCOND5)

B.2.10.6 Step 4: Derive Test Cases (TD4)

Test cases are derived by selecting a scenario to cover, identifying inputs to exercise the path covered by the test case, determining the expected result and repeating until all use case scenarios are covered as required.

Table 50 — Test cases for use case testing

Use Case Name	change_password
Test Case Name	Main Flow
Description	User successfully changes their password
Actors	User
Test Cov. Item Covered	TCOVER1
Use Case Steps Covered	1, 2, 3, 4, 5, 6, 7, 8
Preconditions	User is already logged into the system

#	**Step**	**Expected Result**
1	User clicks Change Password button	System displays Change Password screen
2	User enters their current password correctly	Current password is masked with asterisk (*) symbols
3	User enters their new password correctly	New password is masked with asterisk (*) symbols
4	User re-enters their new password correctly	New re-entered password is masked with asterisk (*) symbols
5	User clicks OK	System displays message “Password changed successfully” and returns user to screen they were viewing before step 1

Table 51 — Test cases for use case testing continued

Use Case Name	change_password
Test Case Name	Alternate Flow – Existing Password Incorrect
Description	User attempts to change password but enters their current password incorrectly
Actors	User
Test Cov. Item Covered	TCOND2
Use Case Steps Covered	1, 2, 3.1, 3.2, 3.3
Preconditions	User is already logged into the system

#	**Step**	**Expected Result**
1	User clicks Change Password button	System displays Change Password screen
2	User enters current password incorrectly	Current password is masked with asterisk (*) symbols
3	User clicks OK	System displays error message “Current password entered incorrectly. Please try again.”

Table 52 — Test cases for use case testing continued

Use Case Name	change_password
Test Case Name	Alternate Flow – New Password Less Than 8 Characters
Description	User attempts to change password but enters less than 8 characters for password
Actors	User
Test Cov. Item Covered	TCOND3
Use Case Steps Covered	1, 2, 3, 4.1.1, 4.1.2
Preconditions	User is already logged into the system

#	**Step**	**Expected Result**
1	User clicks Change Password button	System displays Change Password screen
2	User enters their current password correctly	Current password is masked with asterisk (*) symbols
3	User enters a new password that is less than 8 characters long	New password is masked with asterisk (*) symbols
4	User clicks OK	System displays an error message “New password must be at least 8 characters long. Please try again.”

Table 53 — Test cases for use case testing continued

Use Case Name	change_password
Test Case Name	Alternate Flow – New Password Same as Current Password
Description	User attempts to change password but enters new password matching old password
Actors	User
Test Cov. Item Covered	TCOND4
Use Case Steps Covered	1, 2, 3, 4.2.1, 4.2.2, 4.2.3
Preconditions	User is already logged into the system

#	**Step**	**Expected Result**
1	User clicks Change Password button	System displays Change Password screen
2	User enters their current password correctly	Current password is masked with asterisk (*) symbols
3	User enters a new password that is the same as their current password	New password is masked with asterisk (*) symbols
4	User clicks OK	System displays error message “New password must not be the same as current password. Please try again.”

Table 54 — Test cases for use case testing continued

Use Case Name	change_password
Test Case Name	Alternate Flow – New Passwords Do Not Match
Description	User attempts to change password but their new passwords do not match
Actors	User
Test Cov. Item Covered	TCOND5
Use Case Steps Covered	1, 2, 3, 4, 5.1, 5.2, 5.3
Preconditions	User is already logged into the system

#	**Step**	**Expected Result**
1	User clicks Change Password button	System displays Change Password screen
2	User enters their current password correctly	Current password is masked with asterisk (*) symbols
3	User enters their new password correctly	New password is masked with asterisk (*) symbols
4	User re-enters new password that does not match new password entered at step 3	Re-entered password is masked with asterisk (*) symbols
5	User clicks OK	System displays error message “New passwords do not match. Please try again.”

B.2.10.7 Step 5: Assemble Test Sets (TD5)

Tests could be grouped according to whether they cover the typical or alternate scenarios:

TS1: TEST CASE 1.
TS2: TEST CASES 2, 3, 4, 5.

B.2.10.8 Step 6: Derive Test Procedures (TD6)

Only one procedure is required as follows:

TP1: covering the test case in TS1, in the order specified in the test set.
TP2: covering the test cases in TS2, in the order specified in the test set.

B.2.10.9 Use Case Test Coverage

Using the formula for calculating scenario test coverage provided in clause 6.1.10 and the test coverage items derived above:

$$Coverage_{(u\sec case)} = \frac{5}{5} \times 100\% = 100\%$$

B.2.11 Random Testing

B.2.11.1 Introduction

The aim of random testing is to derive a set of test cases that cover the input parameters of a test item using values that are selected according to a chosen input distribution. This technique requires no partitioning of the input domain of the test item, but simply requires input values to be chosen from this input domain at random.

B.2.11.2 Example

Consider a test item that transforms coordinates, with the following test basis:

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*

B.2.11.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: transforms coordinates function

B.2.11.4 Step 2: Derive Test Conditions (TD2)

The test conditions in random testing are the domain of all possible inputs from which test input values can be selected for each input parameter. They are:

TCOND1:	x - range -320..+320, in increments of $1/2^6$	(for FS1)
TCOND2:	y - range -240..+240, in increments of $1/2^7$	(for FS1)

B.2.11.5 Step 3: Derive Test Coverage Items (TD3)

There are no recognised test coverage items from random testing.

B.2.11.6 Step 4: Derive Test Cases (TD4)

Test cases can now be constructed by first choosing an input distribution and then applying that input distribution to each test condition and determining the expected result of each test case (shown as ‘output’ for the two output parameters ‘r’ and ‘H’ in the table below). Since no information is available about the operational distribution of the input parameters to the test item in this example, a uniform distribution is chosen. From the definitions we can see that in any one randomly chosen input for x can take one of 41,024 values (641 x 2^6), while y can take one of 61,568 values (481 x 2^7). 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.

Since each test case must include selection of a random test input value from the test conditions for both x and y, each test case will cover both test conditions. In a uniform distribution, all input values within the define ranges of x and y have equal probability of being selected as inputs into the test case. For example, the following four test cases could be defined.

Table 55 — Test cases for random testing

Test Case	1	2	3	4
Input (x)	-126.125	11.015625	283.046875	-99.109375
Input (y)	238.046875	78.03125	-156.054688	-9.0625
Test Condition covered	TCOND1 TCOND2	TCOND1 TCOND2	TCOND1 TCOND2	TCOND1 TCOND2
Output r calculated as (r= sqrt (x^2+y^2))	269.3953	78.80495	323.216025	99.5228472
Output H calculated as (cos H = x/r))	117.9161	81.96469	28.8696474	174.775438

B.2.11.7 Step 5: Assemble Test Sets (TD5)

If we assume that all test cases must be executed manually and in the order that they were defined in the test case table, then we could define one test set as follows:

TS1: Manual Testing – TEST CASES 1, 2, 3, 4.

B.2.11.8 Step 6: Derive Test Procedures (TD6)

Since there is only one test set to execute, it is sensible to define one test procedure as follows:

TP1: manual testing, covering all test cases in TS1, in the order specified in the test set.

B.2.11.9 Random Testing Coverage

As stated in clause 6.1.11, there is currently no industry agreed approach for calculating coverage of test coverage items for random testing.

B.2.11.10 Automating Random Testing

Random testing may be performed either manually or using automation. 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 test basis; or

— automatically check test outputs against the test basis.

The automatic generation of random test input values is not difficult using a pseudo-random number generator as long as the test item'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 test basis, however for certain test items it is possible, such as where:

— trusted independently-produced software that performs the same function as the test item is available (presumably not meeting the same constraints such as speed of processing, implementation language, etc.);

— the test is concerned solely with whether the test item crashes or not (so the expected result is "not to crash");

— the nature of the test item'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 test item's function). An example of this is a square root function where simply squaring the output should produce the input.

In the example in clause B.2.11.2, the coordinate transformation test item can be checked automatically using the inverse function approach. In this case, rcosH=x can be obtained directly from the test basis for the test item. By some analysis rsinH=y can also be deduced. If these two equations are satisfied to a reasonable numerical tolerance then the test item 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.

For test items with larger input sets than this small example the “Symbolic Input Attribute Decomposition” (SIAD) tree (Cho 1987) is a useful method for organising the input domain for random sampling before test case design.

Annex C (informative)

Guidelines and Examples for the Application of Structure-Based Test Design Techniques

C.1 Guidelines and Examples for Structure-Based Testing

C.1.1 Overview

This annex provides guidance and examples on the structure-based test design techniques described in clauses 5.3 and 6.2. Each example follows the Test Design and Implementation Process that is defined in ISO/IEC 29119-2. A variety of applications and programming languages are used in these examples. Although each example is applied in a structure-based testing context, as stated in clause 5.1, in practice most of the techniques defined in ISO/IEC 29119-4 can be used interchangeably.

C.2 Structure-Based Test Design Technique Examples

C.2.1 Statement Testing

C.2.1.1 Introduction

The aim of statement testing is to derive a set of test cases that cover the statements of the test item according to a chosen level of statement coverage. This structural test design technique is based upon the decomposition of the test item into constituent statements.

The two principal questions to consider 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 since `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 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.

C.2.1.2 Example

Consider the following test item in Ada, which is designed to categorise positive integers into prime and non-prime, and to give factors for those which are non-prime:

```
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');
```

C.2.1.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: Identify prime and non-prime numbers function

C.2.1.4 Step 2: Derive Test Conditions (TD2)

Test conditions in statement testing are the executable statements in the code. By numbering each line of code, this illustrates test condition numbers. For example, statement 1 allows definition of one test condition:

TCOND1: `READ (Num);` Statement 1 (for FS1)

The remaining test conditions can be defined without repeating the source code each test condition relates to.

TCOND2:	Statement 2	(for FS1)
TCOND3:	Statement 3	(for FS1)
TCOND4:	Statement 4	(for FS1)
TCOND5:	Statement 5	(for FS1)
TCOND6:	Statement 6	(for FS1)
TCOND7:	Statement 7	(for FS1)
TCOND8:	Statement 10	(for FS1)
TCOND9:	Statement 11	(for FS1)
TCOND10:	Statement 13	(for FS1)
TCOND11:	Statement 15	(for FS1)

C.2.1.5 Step 3: Derive Test Coverage Items (TD3)

The test coverage items in statement testing are the same as the test conditions:

TCOVER1:	Statement 1	(for TCOND1)
TCOVER2:	Statement 2	(for TCOND2)
TCOVER3:	Statement 3	(for TCOND3)
TCOVER4:	Statement 4	(for TCOND4)
TCOVER5:	Statement 5	(for TCOND5)
TCOVER6:	Statement 6	(for TCOND6)
TCOVER7:	Statement 7	(for TCOND7)
TCOVER8:	Statement 10	(for TCOND8)
TCOVER9:	Statement 11	(for TCOND9)
TCOVER10:	Statement 13	(for TCOND13)
TCOVER11:	Statement 15	(for TCOND11)

C.2.1.6 Step 4: Derive Test Cases (TD4)

In statement testing, each statement must be covered by at least one test case. Test cases are derived by first identifying sub-paths in the control flow graph that execute one or more executable statements that have not yet been covered by a test case. The inputs to execute the sub-path are then identified, along with the expected result. This process is repeated until the required level of test coverage is achieved. In this example, only one test case is required to cover all statements in the code (i.e. to achieve 100% statement coverage).

Table 56 — Test cases for statement testing

Test Case	Input	Expected Result	Test Coverage Items Covered
1	2	2 is prime	1, 2, 3, 4, 5, 8, 9, 10
	4	2 is a factor of 4	2, 3, 4, 5, 6, 7, 8, 10, 11
		End of prime number program	

C.2.1.7 Step 5: Assemble Test Sets (TD5)

Since there is only one test case, it can be placed in the one test set as follows:

TS1: TEST CASE 1.

C.2.1.8 Step 6: Derive Test Procedures (TD6)

Only one procedure is required as follows:

TP1: covering the test case in TS1.

C.2.1.9 Statement Test Coverage

Using the formula provided in clause 6.2.1 and the test coverage items derived above:

$$Coverage_{(statement)} = \frac{11}{11} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for statement testing has been achieved.

C.2.2 Branch / Decision Testing

C.2.2.1 Introduction

Branch and Decision Coverage are closely related. For test items 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 (Kernighan and Richie 1998). The three decisions are 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;
}
```

C.2.2.2 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: binsearch function

C.2.2.3 Step 2: Derive Test Conditions (TD2)

C.2.2.3.1 Options for Derivation of Test Conditions

The identification of test conditions for branch/decision testing may be demonstrated by creating a 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 within each basic block will 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 -> B2	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 19. The graph has one entry point, B1, and two exit points, B7 and B9.

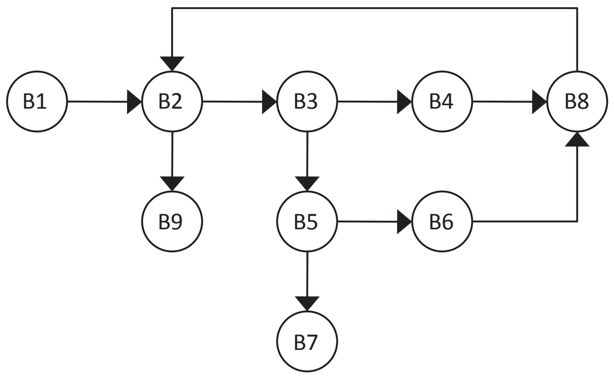


Figure 19 — Control flow graph for binsearch

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 test conditions for branch coverage will be different from those for decision coverage. This is demonstrated under steps 2a and 2b below.

C.2.2.3.2 Option 2a: Derive Test Conditions for Branch Coverage (TD2)

For branch coverage, the test conditions (BRANCH-TCOND) are the branches (arcs) that are represented by arrows in the control flow graph. There are ten in total, as follows:

BRANCH-TCOND1:	B1 -> B2	(for FS1)
BRANCH-TCOND2:	B2 -> B3	(for FS1)
BRANCH-TCOND3:	B2 -> B9	(for FS1)
BRANCH-TCOND4:	B3 -> B4	(for FS1)

]

BRANCH-TCOND5:	B3 -> B5	(for FS1)
BRANCH-TCOND6:	B4 -> B8	(for FS1)
BRANCH-TCOND7:	B5 -> B6	(for FS1)
BRANCH-TCOND8:	B5 -> B7	(for FS1)
BRANCH-TCOND9:	B6 -> B8	(for FS1)
BRANCH-TCOND10:	B8 -> B2	(for FS1)

C.2.2.3.3 Option 2b: Derive Test Conditions for Decision Coverage (TD2)

For decision coverage, the test conditions (DECISION-TCOND) are the decisions represented as nodes in the control flow graph that have more than one exit arrow. In this example, there are three test conditions:

DECISION-TCOND1:	B2	(for FS1)
DECISION-TCOND2:	B3	(for FS1)
DECISION-TCOND3:	B5	(for FS1)

C.2.2.4 Step 3: Derive Test Coverage Items (TD3)

C.2.2.4.1 Options for Derivation of Test Coverage Items

The test coverage items for branch coverage will be different from those for decision coverage. This is demonstrated under steps 3a and 3b below.

C.2.2.4.2 Option 3a: Derive Test Coverage Items for Branch Coverage (TD3)

For branch coverage, the test coverage items are the branches in the control flow graph, which are the same as the test conditions. In this example there are ten test coverage items for branch coverage, as follows:

BRANCH-TCOVER1:	B1 -> B2	(for BRANCH-TCOND1)
BRANCH-TCOVER2:	B2 -> B3	(for BRANCH-TCOND2)
BRANCH-TCOVER3:	B2 -> B9	(for BRANCH-TCOND3)
BRANCH-TCOVER4:	B3 -> B4	(for BRANCH-TCOND4)
BRANCH-TCOVER5:	B3 -> B5	(for BRANCH-TCOND5)
BRANCH-TCOVER6:	B4 -> B8	(for BRANCH-TCOND6)
BRANCH-TCOVER7:	B5 -> B6	(for BRANCH-TCOND7)
BRANCH-TCOVER8:	B5 -> B7	(for BRANCH-TCOND8)
BRANCH-TCOVER9:	B6 -> B8	(for BRANCH-TCOND9)
BRANCH-TCOVER10:	B8 -> B2	(for BRANCH-TCOND10)

C.2.2.4.3 Option 3b: Derive Test Coverage Items for Decision Coverage (TD3)

For decision coverage, the outcomes (i.e. true, false) of each decision are the test coverage items. In this example, each decision has two outcomes corresponding to the true and false values of the decisions; therefore there are six test coverage items, as follows:

DECISION-TCOVER1:	B2 = true	(for DECISION-TCOND1)
DECISION-TCOVER2:	B2 = false	(for DECISION-TCOND1)
DECISION-TCOVER3:	B3 = true	(for DECISION-TCOND2)
DECISION-TCOVER4:	B3 = false	(for DECISION-TCOND2)
DECISION-TCOVER5:	B5 = true	(for DECISION-TCOND3)
DECISION-TCOVER6:	B5 = false	(for DECISION-TCOND3)

It is generally possible for a decision to have more than two outcomes, which would increase the number of test coverage items that need to be derived.

C.2.2.5 Step 4: Derive Test Cases (TD4)

Test cases for branch testing are derived by identifying control flow sub-paths that reach one or more branches (test coverage items) that have not yet been executed during testing, determining inputs that exercise those sub-paths, determining the expected result of each test, and repeating until the required level of test coverage is achieved. Similarly, test cases for decision testing are derived by identifying control flow sub-paths that reach one or more decision that has not been exercised and determining the inputs and expected outputs for each test. For both branch coverage and decision coverage, any individual test of the test item will exercise a sub-path and hence potentially many decisions and branches.

Consider a test case which executes the sub-path B1 -> B2 -> B9. This case arises when n=0, that is, when the table being searched has no entries. This sub-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 sub-path:

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

This sub-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.

These test cases are shown below:

Table 57 — Test cases for binsearch

Test Case	Inputs			Decisions Exercised (underlined)	Test Coverage Items Covered	Expected Result
	Word	Tab	n			
1	chas	Alf bert chas dick eddy fred geoff	7	B1 » B2 » B3 » B4 » B8 » B2 » B3 » B5 » B6 » B8 » B2 » B3 » B5 » B7	BRANCH-TCOVER 1, 2, 4, 5, 6, 7, 8, 9, 10 and DECISION-TCOVER 1, 3, 4, 5, 6	2
2	chas	'empty table'	0	B1 » B2 » B9	BRANCH-TCOVER 1, 3 and DECISION-TCOVER 2	-1

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

C.2.2.6 Step 5: Assemble Test Sets (TD5)

Since there are only two test cases required, we may choose to combine them into the one test set.

TS1: TEST CASES 1 and 2.

C.2.2.7 Step 6: Derive Test Procedures (TD6)

Again, since there are only two test cases and one test set, we may choose to define only one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

C.2.2.8 Branch Test Coverage

Using the formula provided in clause 6.2.2 and the test coverage items derived above:

$$Coverage_{(branch)} = \frac{10}{10} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for branch testing has been achieved.

C.2.2.9 Decision Test Coverage

Using the formula provided in clause 6.2.3 and the test coverage items derived above:

$$Coverage_{(decision)} = \frac{6}{6} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for decision testing has been achieved.

C.2.3 Branch Condition Testing, Branch Condition Combination Testing and Modified Condition Decision Coverage (MCDC) Testing

C.2.3.1 Introduction

Branch Condition Testing, Branch Condition Combination Testing, and Modified Condition Decision Coverage Testing are closely related, as are the associated coverage measures. The aim of these three test design techniques is to derive a set of test cases that cover the conditions within decisions of the test item according to a chosen level of coverage. For convenience, these test case design and test coverage measurement approaches are demonstrated using one example.

C.2.3.2 Example

Consider the following fragment of code:

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

The Boolean conditions within the decision condition are A, B and C. These may themselves be comprised of complex expressions involving relational operators. For example, the Boolean condition 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 conditions.

C.2.3.3 Step 1: Identify Feature Sets (TD1)

Since the example for all three test design techniques is based on the same test item (the code fragment above), one feature set can be defined for demonstration in all three techniques:

FS1: condition code fragment

C.2.3.4 Step 2: Derive Test Conditions (TD2)

In these three test design techniques, each decision in the control flow graph is a test condition. In this example, there is one decision:

TCOND1: `if A or (B and C) then` (for FS1)

This test condition is applicable to Branch Condition Testing, Branch Condition Combination Testing, and Modified Condition Decision Coverage Testing.

C.2.3.5 Branch Condition Testing

C.2.3.5.1 Step 3: Derive Test Coverage Items (TD3)

Branch condition testing examines the individual conditions within multi-condition decisions, with the aim that each individual condition takes on both true and false values. Note that it is possible to have 100% branch condition coverage and yet not test every outcome for the individual decisions themselves. The test coverage items are the Boolean values (true/false) of the conditions within decisions. In this example, this technique would require Boolean condition A to be evaluated both TRUE and FALSE, Boolean condition B to be evaluated both TRUE and FALSE and Boolean condition C to be evaluated both TRUE and FALSE. Therefore, the test coverage items for this technique are:

TCOVER1:	A = TRUE	(for TCOND1)
TCOVER2:	A = FALSE	(for TCOND1)
TCOVER3:	B = TRUE	(for TCOND1)
TCOVER4:	B = FALSE	(for TCOND1)
TCOVER5:	C = TRUE	(for TCOND1)
TCOVER6:	C = FALSE	(for TCOND1)

C.2.3.5.2 Step 4: Derive Test Cases (TD4)

Branch Condition Coverage test cases are derived by identifying control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing, determining the inputs that will execute those sub-paths and the expected result of the test and repeating until all the required coverage is achieved. In this example, this can 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):

NOTE The test cases contained in the table below are not “complete” in that they do not include expected results.

Table 58 — Test cases for Branch Condition Testing

Test Case	A	B	C	A or (B and C)	Test Cov. Items Covered
1	FALSE	FALSE	FALSE	FALSE	TCOVER 2, 4, 6
2	TRUE	TRUE	TRUE	TRUE	TCOVER 1, 3, 5

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

A 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 decision. Thus Branch Condition Coverage will not necessarily subsume Branch Coverage.

Table 59 — Alternative test cases for Branch Condition Testing

Test Case	A	B	C	A or (B and C)	Test Cov. Items Covered
1	TRUE	FALSE	FALSE	TRUE	TCOVER 1, 4, 6
2	FALSE	TRUE	TRUE	TRUE	TCOVER 2, 3, 5

C.2.3.5.3 Step 5: Assemble Test Sets (TD5)

Since there are only two test cases required, we may choose to combine them into the one test set.

TS1: TEST CASES 1 and 2.

C.2.3.5.4 Step 6: Derive Test Procedures (TD6)

Again, since there are only two test cases and one test set, we may choose to define only one test procedure.

TP1: covering all test cases in TS1, in the order specified in the test set.

C.2.3.5.5 Branch Condition Test Coverage

Using the formula provided in clause 6.2.4 and the test coverage items derived above:

$$Coverage_{(branch\_condition)} = \frac{6}{6} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for branch condition testing has been achieved.

C.2.3.6 Branch Condition Combination Testing

C.2.3.6.1 Step 3: Derive Test Coverage Items (TD3)

In branch condition combination testing, the test coverage items are the unique combinations of Boolean values of conditions within decisions. In this example, this technique would require all combinations of Boolean conditions A, B and C to be evaluated. Therefore, the test coverage items for this technique are:

TCOVER1:	A = FALSE,	B = FALSE,	C = FALSE	(for TCOND1)
TCOVER2:	A = TRUE,	B = FALSE,	C = FALSE	(for TCOND1)
TCOVER3:	A = FALSE,	B = TRUE,	C = FALSE	(for TCOND1)
TCOVER4:	A = TRUE,	B = TRUE,	C = FALSE	(for TCOND1)
TCOVER5:	A = FALSE,	B = FALSE,	C = TRUE	(for TCOND1)
TCOVER6:	A = TRUE,	B = FALSE,	C = TRUE	(for TCOND1)
TCOVER7:	A = FALSE,	B = TRUE,	C = TRUE	(for TCOND1)
TCOVER8:	A = TRUE,	B = TRUE,	C = TRUE	(for TCOND1)

C.2.3.6.2 Step 4: Derive Test Cases (TD4)

Test cases are derived by identifying control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing, determining the inputs that will execute those sub-paths and the expected result of the test and repeating until the required coverage is achieved. In this example, this can be achieved by deriving the following test cases:

Table 60 — Test cases for Branch Condition Combination Testing

Test Case	A	B	C	Test Cov. Items Covered
1	FALSE	FALSE	FALSE	TCOVER1
2	TRUE	FALSE	FALSE	TCOVER2
3	FALSE	TRUE	FALSE	TCOVER3
4	TRUE	TRUE	FALSE	TCOVER4
5	FALSE	FALSE	TRUE	TCOVER5
6	TRUE	FALSE	TRUE	TCOVER6
7	ALSE	TRUE	TRUE	TCOVER7
8	TRUE	TRUE	TRUE	TCOVER8

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

C.2.3.6.3 Step 5: Assemble Test Sets (TD5)

It may be decided that all test cases for this technique are combined in to the one test set, as follows:

TS1: TEST CASES 1, 2, 3, 4, 5, 6, 7, 8.

C.2.3.6.4 Step 6: Derive Test Procedures (TD6)

Since there is only one test set, we may choose to define one corresponding test procedure, as follows:

TP1: covering all test cases in TS1, executed in the order specified in the test set.

C.2.3.6.5 Branch Condition Combination Test Coverage

Using the formula provided in clause 6.2.5 and the test coverage items derived above:

$$Coverage_{(branch\_condition\_combination)} = \frac{8}{8} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for branch condition combination testing has been achieved.

C.2.3.7 Modified Condition Decision Coverage Testing

Modified Condition Decision Coverage (MCDC) Testing 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. MCDC Testing requires test cases to show that each Boolean condition (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).

C.2.3.7.1 Step 3: Derive Test Coverage Items (TD3)

In modified condition decision coverage (MCDC) testing, the test coverage items are the unique combinations of individual Boolean values of conditions within decisions that allow a single Boolean condition to independently affect the outcome.

For the example decision condition [`A` or (`B` and `C`)], this leads to pairs of test coverage items, 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, as follows:

TCOVER1:	A = FALSE,	B = FALSE,	C = TRUE	OUTCOME = FALSE	(for TCOND1)
TCOVER2:	A = TRUE,	B = FALSE,	C = TRUE	OUTCOME = TRUE	(for TCOND1)

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:

TCOVER3:	A = FALSE,	B = FALSE,	C = TRUE	OUTCOME = FALSE	(for TCOND1)
TCOVER4:	A = FALSE,	B = TRUE,	C = TRUE	OUTCOME = TRUE	(for TCOND1)

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:

TCOVER5:	A = FALSE,	B = TRUE,	C = FALSE	OUTCOME = FALSE	(for TCOND1)
TCOVER6:	A = FALSE,	B = TRUE,	C = TRUE	OUTCOME = TRUE	(for TCOND1)

C.2.3.7.2 Step 4: Derive Test Cases (TD4)

Test cases are derived by identifying control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing, determining the inputs that will execute those sub-paths and the expected result of the test and repeating until all the required coverage is achieved. In this example, this can be achieved with the following set of test cases:

NOTE The test cases contained in the table below are not "complete" in that they do not include expected results.

Table 61 — Test cases for Modified Condition Decision Coverage

Test Case	A	B	C	Expected Result	Test Cov. Items Covered
A1	FALSE	FALSE	TRUE	FALSE	TCOVER1
A2	TRUE	FALSE	TRUE	TRUE	TCOVER2
B1	FALSE	FALSE	TRUE	FALSE	TCOVER3
B2	FALSE	TRUE	TRUE	TRUE	TCOVER4
C1	FALSE	TRUE	FALSE	FALSE	TCOVER5
C2	FALSE	TRUE	TRUE	TRUE	TCOVER6

Having created these pairs of test cases for each condition separately, it can be seen that test cases A1 and B1 are the same, and that test cases B2 and C2 are the same. The overall set of test cases to provide 100% MCDC of the example expression is consequently:

Table 62 — Overall set of test cases

Test Case	A	B	C	Expected Result	Test Cov. Items Covered
1 (A1,B1)	FALSE	FALSE	TRUE	FALSE	TCOVER 1, 3
2 (A2)	TRUE	FALSE	TRUE	TRUE	TCOVER 2
3 (B2,C2)	FALSE	TRUE	TRUE	TRUE	TCOVER 4, 5
4 (C1)	FALSE	TRUE	FALSE	FALSE	TCOVER 6

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; and

— C is shown to independently affect the outcome of the decision condition by test cases 3 and 4.

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:

Table 63 — Alternate MCDC 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 C1 (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 conditions within the decision condition. In contrast, Branch Condition Combination Coverage requires 2^ntest cases. MCDC is therefore a practical low-risk compromise with Branch Condition Combination Coverage where condition expressions involve more than just a few Boolean conditions.

C.2.3.7.3 Step 5: Assemble Test Sets (TD5)

It may be assumed that all test cases defined in Table 62 for this technique are combined into the one test set, as follows:

TS1: TEST CASES A1, A2, B1, B2, C1, C2.

Alternatively, it may be decided that only the optimised set of test cases defined in Table 63 are required and these could be placed into one test set as follows:

TS1: TEST CASES 1, 2, 3, 4.

C.2.3.7.4 Step 6: Derive Test Procedures (TD6)

Since there is only one test set, we may choose to define one corresponding test procedure, as follows:

TP1: covering all test cases in TS1, in the order specified in the test set.

C.2.3.7.5 Modified Condition Decision Coverage Test Coverage

Using the formula provided in clause 6.2.6 and the test coverage items derived above:

$$Coverage_{(modified\_condtion\_decision\_coverage)} = \frac{6}{6} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for MCDC testing has been achieved.

C.2.3.8 Other Boolean Expressions

One weakness of these three test design techniques and test coverage measurement approaches is that they are vulnerable to the placement of Boolean expressions when control decisions are 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 three test design techniques and coverage measures is to design tests for all Boolean expressions, not just those used directly in control flow decisions.

C.2.3.9 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 condition, then the second condition will not be evaluated.

The consequence is that it will be infeasible to show coverage of one value of the second condition. 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, 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 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.

C.2.3.10 Other Branches and Decisions

The above descriptions of branch condition testing, branch condition combination testing and MCDC testing and their corresponding coverage measures are 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 without any conditions) do not use Boolean conditions, and are therefore not addressed by the descriptions.

One way of handling this scenario is to use a one of these three test design techniques and a coverage measures 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, test cases that achieve 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.

C.2.4 Data Flow Testing

C.2.4.1 Introduction

The aim of data flow testing is to derive a set of test cases that cover the paths between definitions and uses of variables in a test item according to a chosen level of def-use coverage. Data flow testing is a structure-based test design technique which aims to execute sub-paths from points where each variable in a test item is defined to points where it is referenced. These sub-paths are known as definition-use pairs. The different data flow coverage criteria require different definition-use pairs and sub-paths to be executed. Test sets are generated here to achieve 100% coverage (where possible) for each of those criteria.

NOTE Data flow 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.

C.2.4.2 Example

Consider the data flow testing of the following test item 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.)

C.2.4.3 Step 1: Identify Feature Sets (TD1)

Since there is only one function under test, there is one feature set that can be defined:

FS1: Solve_Quadratic function

C.2.4.4 Step 2: Derive Test Conditions (TD2)

The first step is to list the variables used in the test item. These are: A, B, C, Discrim, Is_Complex, R1 and R2. Next, each occurrence of a variable in the test item is cross referenced against the program listing and assigned a category (definition, predicate-use or computation-use).

Table 64 — Occurrence of variables and their categories

	Category		
Line	**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	

Line	Category		
	definition	c-use	p-use
11	R2		
		A	
		B	
		Discrim	
12			
13			

The test conditions are the definition-use pairs.

The next step is to identify the definition-use pairs and their type (c-use or p-use), each of which are test coverage items, by identifying links from each entry in the definition column to each later 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 test item.

Table 65 — definition-use pairs and their type

Definition-use pair (start line -> end line)	Variable(s)		Test Conditions
	c-use	p-use	
0 ---> 1	A		TCOND1
	B		TCOND2
	C		TCOND3
0 ---> 10	A		TCOND4
	B		TCOND5
0 ---> 11	A		TCOND6
	B		TCOND7
1 ---> 4		Discrim	TCOND8
1 ---> 10	Discrim		TCOND9
1 ---> 11	Discrim		TCOND10
5 ---> 9		Is_Complex	TCOND11
7 ---> 9		Is_Complex	TCOND12

Note that it is not always necessary to derive all of the def-use pairs (as has been done here) in order to proceed with deriving the test coverage items (depending on the technique being used).

C.2.4.5 All-Definitions Testing

C.2.4.5.1 Step 3a: Derive Test Coverage Items (TD3) – All-definitions Testing

In all-definitions testing, the test coverage items are the control flow sub-paths from a variable definition to some use (either p-use or c-use) of that definition.

The following table shows one set of def-use pairs that meet this criterion.

Table 66 — All-Definitions Testing

Test Coverage Items	**All-Definitions**		
	Variable(s)	**definition-use pair / sub-path**	**Test Conditions Covered**
TCOVER1	A	0 --> 1	TCOND1
TCOVER2	B	0 --> 1	TCOND2
TCOVER3	C	0 --> 1	TCOND3
TCOVER4	Discrim	1 --> 4	TCOND8
TCOVER5	Is_Complex	5 --> 9	TCOND11
TCOVER6	Is_Complex	7 --> 9	TCOND12

C.2.4.5.2 Step 4a: Derive Test Cases (TD4) – All-definitions Testing

Test cases are derived by identifying control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing, determining the inputs that will execute those sub-paths and the expected result of the test and repeating until the required coverage is achieved. To achieve 100% All-definitions data flow coverage at least one sub-path from each variable definition to some use of that definition (either p-use or c-use) must be executed. The following test set would satisfy this requirement:

Table 67 — Test Cases for All-Definitions Testing

Test Case	All-Definitions				INPUTS			EXPECTED RESULT		
	Variable(s)	Definition-use pair	Sub-path	Test Coverage Items Covered	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	TCOVER 1, TCOVER 2, TCOVER 3	1	1	1	T	unass.	unass.
2	Discrim	1 --> 4	1-4	TCOVER 4	1	1	1	T	unass.	unass.
3	Is_Complex	5 --> 9	5- 9	TCOVER 5	1	1	1	T	unass.	unass.
4	Is_Complex	7 --> 9	7- 9	TCOVER 6	1	2	1	F	-1	-1

Note that the first test case (test case 1) satisfies the requirement for more than one variable; this may apply to each of the subsequent test sets as well. It can also be seen from test cases 1, 2 and 3 that the same test inputs satisfy the sub-path execution criteria for several definition-use pairs(again this may apply to the subsequent test sets as well).

C.2.4.5.3 All-Definitions Testing Coverage

Using the formula provided in clause 6.2.7.1 and the test coverage items derived above:

$$Coverage_{(all\_definitions)} = \frac{6}{6} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for all-definitions testing has been achieved.

C.2.4.6 All C-Uses Testing

C.2.4.6.1 Step 3b: Derive Test Coverage Items (TD3) – All-C-Uses Testing

In all-c-uses testing, the test coverage items are the control flow sub-paths from a variable definition to every c-use of that definition.

The following table shows one set of def-use pairs that meet this criterion.

Table 68 — All-C-Uses Testing

	All-c-uses			
Test Cov. Items	**Variable**	**Definition-use pair**	**Sub-path**	**Test Conditions Covered**
TCOVER1	A	0 --> 1	0-1	TCOND1
TCOVER2	B	0 --> 1	0-1	TCOND2
TCOVER3	C	0 --> 1	0-1	TCOND3
TCOVER4	A	0 --> 10	0-1-4-7-9-10	TCOND4
TCOVER5	B	0 --> 10	0-1-4-7-9-10	TCOND5
TCOVER6	A	0 --> 11	0-1-4-7-9-10-11	TCOND6
TCOVER7	B	0 --> 11	0-1-4-7-9-10-11	TCOND7
TCOVER8	Discrim	1 --> 10	1-4-7-9-10	TCOND9
TCOVER9	Discrim	1 --> 11	1-4-7-9-10-11	TCOND10

C.2.4.6.2 Step 4b: Derive Test Cases (TD4) – All-C-Uses Testing

Test cases are derived by identifying control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing, determining the inputs that will execute those sub-paths and the expected result of the test and repeating until the required coverage is achieved. To achieve 100% 'All-c-uses data flow coverage at least one sub-path from each variable definition to every c-use of that definition must be executed. The following two test cases would satisfy this requirement:

Table 69 — Test Cases for All-C-Uses Testing

	All-c-uses				INPUTS			EXPECTED RESULT		
Test Case	Variables	Definition-use pairs	Sub-path	Test Coverage Items Covered	A	B	C	Is_Complex	R1	R2
1	A, B, C, Discrim	0 --> 1 0 --> 10	0-1-4-7-9-10	TCOVER1, TCOVER2, TCOVER3, TCOVER4, TCOVER5, TCOVER8	1	2	1	F	-1	-1
2	A, B, Discrim	0 --> 11 1 --> 11	0-1-4-7-9-10-11	TCOVER6, TCOVER7, TCOVER9	1	2	1	F	-1	-1

C.2.4.6.3 All-C-Uses Testing Coverage

Using the formula provided in clause 6.2.7.2 and the test coverage items derived above:

$$Coverage_{(all-c-uses)} = \frac{9}{9} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for all-c-uses testing has been achieved.

C.2.4.7 All P-Uses Testing

C.2.4.7.1 Step 3c: Derive Test Coverage Items (TD3) – All-P-Uses Testing

In all-p-uses testing, the test coverage items are the control flow sub-paths from a variable definition to every p-use of that definition.

The following table shows one set of def-use pairs that meet this criterion.

Table 70 — All-P-Uses Testing

Test Coverage Items	All-p-uses		
	Variable(s)	Definition-use pair/ sub-path	Test Conditions Covered
TCOVER1	Discrim	1 --> 4	TCOND8
TCOVER2	Is_Complex	5 --> 9	TCOND11
TCOVER3	Is_Complex	7 --> 9	TCOND12

C.2.4.7.2 Step 4c: Derive Test Cases (TD4) – All-P-Uses Testing

Test cases are derived by identifying control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing, determining the inputs that will execute those sub-paths and the expected result of the test and repeating until the required coverage is achieved. To achieve 100% All-p-uses data flow coverage at least one sub-path from each variable definition to every p-use of that definition must be executed. The following test set would satisfy this requirement:

Table 71 —Test cases for All-P-Uses Testing

Test Case	All-p-uses				INPUTS			EXPECTED RESULT		
	Variable(s)	Definition-use pair	sub-path	Test Coverage Items Covered	A	B	C	Is_Complex	R1	R2
1	Discrim, Is_Complex	1 --> 4 5 --> 9	1-4 5-9	TCOVER1 TCOVER2	1	1	1	T	unass.	unass.
2	Is_Complex	7 --> 9	7-9	TCOVER3	1	2	1	F	-1	-1

C.2.4.7.3 All-P-Uses Testing Coverage

Using the formula provided in clause 6.2.7.3 and the test coverage items derived above:

$$Coverage_{(all-p-uses)} = \frac{3}{3} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for all-p-uses testing has been achieved.

C.2.4.8 All-Uses Testing

C.2.4.8.1 Step 3d: Derive Test Coverage Items (TD3) – All-Uses Testing

In all-uses testing, the test coverage items are control flow sub-paths from a variable definition to every use (both p-use and c-use) of that definition.

The following table shows one set of def-use pairs that meet this criterion.

Table 72 — All-Uses Testing

Test Coverage Items	All-uses / All du-paths			
	Variable(s)	d-u pair	Sub-path	Test Conditions Covered
TCOVER1	A	0 --> 1	0-1	TCOND1
TCOVER2	B	0 --> 1	0-1	TCOND2
TCOVER3	C	0 --> 1	0-1	TCOND3
TCOVER4	A	0 --> 10	0-1-4-7-9-10	TCOND4
TCOVER5	B	0 --> 10	0-1-4-7-9-10	TCOND5
TCOVER6	A	0 --> 11	0-1-4-7-9-10-11	TCOND6
TCOVER7	B	0 --> 11	0-1-4-7-9-10-11	TCOND7
TCOVER8	Discrim	1 --> 4	1-4	TCOND8
TCOVER9	Discrim	1 --> 10	1-4-7-9-10	TCOND9
TCOVER10	Discrim	1 --> 11	1-4-7-9-10-11	TCOND10
TCOVER11	Is_Complex	5 --> 9	5-9	TCOND11
TCOVER12	Is_Complex	7 --> 9	7-9	TCOND12

C.2.4.8.2 Step 4d: Derive Test Cases (TD4) – All-Uses Testing

Test cases are derived by identifying control flow sub-paths that reach one or more test coverage items that have not yet been executed during testing, determining the inputs that will execute those sub-paths and the expected result of the test and repeating until the required coverage is achieved. To achieve 100% All-uses data flow coverage at least one sub-path 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:

Table 73 — Test Cases for All-Uses Testing

Test Case	All-uses / All du-paths				INPUTS			EXPECTED RESULT		
	Variable(s)	d-u pair	Sub-path	Test Coverage Items Covered	A	B	C	Is_Complex	R1	R2
1	A, B, C, Discrim	0 --> 1 0 --> 10	0-1-4-7-9-10	TCOVER1, TCOVER2, TCOVER3, TCOVER4, TCOVER5, TCOVER9	1	1	1	T	-1	-1
2	A, B, Discrim	1 --> 4 0 --> 11 7 --> 9	0-1-4-7-9-10-11	TCOVER6, TCOVER7, TCOVER8, TCOVER10, TCOVER12	1	2	1	F	-1	-1
3	Is_Complex	5 --> 9	5-9	TCOVER11	1	1	1	F	unass.	unass.

C.2.4.8.3 All-Uses Testing Coverage

Using the formula provided in clause 6.2.7.4 and the test coverage items derived above:

$$Coverage_{(all\_uses)} = \frac{12}{12} \times 100\% = 100\%$$

Thus, 100% coverage of test coverage items for all-uses testing has been achieved.

C.2.4.9 All-DU-Paths Testing

C.2.4.9.1 Step 3e: Derive Test Coverage Items (TD3) – All-DU-Paths Testing

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

Table 74 — All-Uses Testing

	All du-paths			
Test Coverage Items	Variable(s)	d-u pair	Sub-path	Test Conditions Covered
TCOVER1	A	0 --> 1	0-1	TCOND1
TCOVER2	B	0 --> 1	0-1	TCOND2
TCOVER3	C	0 --> 1	0-1	TCOND3
TCOVER4	A	0 --> 10	0-1-4-7-9-10	TCOND4
TCOVER5	B	0 --> 10	0-1-4-7-9-10	TCOND5
TCOVER6	A	0 --> 11	0-1-4-7-9-10-11	TCOND6
TCOVER7	B	0 --> 11	0-1-4-7-9-10-11	TCOND7
TCOVER8	Discrim	1 --> 4	1-4	TCOND8
TCOVER9	Discrim	1 --> 10	1-4-7-9-10	TCOND9
TCOVER10	Discrim	1 --> 11	1-4-7-9-10-11	TCOND10
TCOVER11	Is_Complex	5 --> 9	5-9	TCOND11
TCOVER12	Is_Complex	7 --> 9	7-9	TCOND12
TCOVER 13	Is_Complex	0 --> 10	0-1-4-5-9-10	TCOND13 (infeasible)
TCOVER 14	Is_Complex	1 --> 10	1-4-5-9-10	TCOND14 (infeasible)

C.2.4.9.2 Step 4e: Derive Test Cases (TD4) – All-DU-Paths Testing

Test cases for all-du-paths can now be derived. The same set of test cases that were derived for all-uses also achieves the maximum level of test coverage item coverage possible for all-du-paths testing in this example.

Table 75 — Test Cases for All-DU-Paths Testing

	All-uses / All du-paths				INPUTS			EXPECTED RESULT		
Test Case	Variable(s)	d-u pair	Sub-path	Test Coverage Items Covered	A	B	C	Is_Complex	R1	R2
1	A, B, C, Discrim	0 --> 1 0 --> 10	0-1-4-7-9-10	TCOVER1, TCOVER2, TCOVER3, TCOVER4, TCOVER5, TCOVER9	1	1	1	T	-1	-1
2	A, B, Discrim	1 --> 4 0 --> 11 7 --> 9	0-1-4-7-9-10-11	TCOVER6, TCOVER7, TCOVER8, TCOVER10, TCOVER12	1	2	1	F	-1	-1
3	Is_Complex	5 --> 9	5-9	TCOVER11	1	1	1	F	unass.	unass.

C.2.4.9.3 All-DU-Paths Testing Coverage

Using the formula provided in clause 6.2.7.5 and the test coverage items derived above:

$$Coverage_{(all-du-paths)} = \frac{12}{14} \times 100\% = 100\%$$

Thus, 86% coverage of test coverage items for all-du-paths testing has been achieved.

C.2.4.9.4 Step 5: Assemble Test Sets (TD5)

Since the test cases derived for all-uses cover all test coverage items, they will be used as an example of the identification of test sets. All test cases that cause Is_Complex to compute to True (T) could be combined into one test set, and all those that compute to False (F) in another test set, as follows:

TS1: TEST CASES 1, 4, 7.
TS2: TEST CASES 2, 3, 5, 6, 8.

C.2.4.9.5 Step 6: Derive Test Procedures (TD6)

All test sets could be combined into the one test procedure to be executed in sequential order, as follows:

TP1: covering all test cases in TS1, followed by those in TS2, in the order specified in the test sets.

Annex D
(informative)

Guidelines and Examples for the Application of Experience-Based Test Design Techniques

D.1 Guidelines and Examples for Experience-Based Testing

D.1.1 Overview

This annex provides guidance on the requirements in clauses 5.4 and 6.3. This clause demonstrates the application of an experience-based test design technique to an example problem. The example follows the Test Design and Implementation Process that is defined in ISO/IEC 29119-2.

D.2 Experience-Based Test Design Technique Examples

D.2.1 Error Guessing

D.2.1.1 Introduction

The aim of error guessing is to derive a set of test cases that cover likely errors, using a tester's knowledge and experience with previous test items. Test cases are derived to exercise each type of error that is identified by the tester as likely being present in the current test item. This technique would typically be applied after other specification-based test design techniques such as equivalence partitioning and boundary value analysis, to supplement the types of errors targeted by those techniques.

D.2.1.2 Example

Consider the example test item, *generate_grading*, which was used as the example for boundary value analysis and which had the following test basis:

The component receives as input an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it outputs a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark, which is 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.

D.2.1.3 Step 1: Identify Feature Sets (TD1)

As there is only one test item defined in the test basis, only one feature set needs to be defined:

FS1: *generate_grading* function

D.2.1.4 Step 2: Derive Test Conditions (TD2)

Test conditions are identified by deriving a list of potential types of errors that may be present in the test item, based on knowledge and experience of similar errors in other test items that were tested in the past. For the *generate_grading* function, the following test conditions may be derived:

TCOND1:	enter NULL	(for FS1)
TCOND2:	enter 0	(for FS1)
TCOND3:	enter negative number	(for FS1)
TCOND4:	enter inputs in reverse order	(for FS1)
TCOND5:	enter very large number (e.g. 10 digits)	(for FS1)
TCOND6:	enter very large string of alphas (e.g. 10 alphas)	(for FS1)

D.2.1.5 Step 3: Derive Test Coverage Items (TD3)

Each generic type of defect is a test coverage item (i.e. the test coverage items are the same as the test conditions). Therefore, the following test coverage items can be defined:

TCOVER1:	enter NULL	(for TCOND1)
TCOVER2:	enter 0	(for TCOND2)
TCOVER3:	enter negative number	(for TCOND3)
TCOVER4:	enter inputs in reverse order	(for TCOND4)
TCOVER5:	enter very large number (e.g. 10 digits)	(for TCOND5)
TCOVER6:	enter very large string of alphas (e.g. 10 alphas)	(for TCOND6)

D.2.1.6 Step 4: Derive Test Cases (TD4)

Test cases can now be derived by selecting a defect type for inclusion in the current test case, identifying input(s) to exercise the parameters of the test case according to the chosen defect type, determining the expected result and repeating until all test coverage items are included in a test case. For this example, this results in the following test cases.

Table 76 — Test cases for error guessing

Test Case	1	2	3	4
Input (exam mark)	NULL	25	NULL	0
Input (c/w mark)	20	NULL	NULL	20
total mark (as calculated)	20	25	NULL	20
Test Cov. Item covered	TCOVER1 exam mark	TCOVER1 c/work mark	TCOVER1 exam & c/work marks	TCOVER2 exam mark
Exp. Output	'FM'	'FM'	'FM'	'FM'

Table 77 — Test cases for error guessing continued

Test Case	5	6	7	8
Input (exam mark)	25	0	-25	25
Input (c/w mark)	0	0	20	-25
total mark (as calculated)	25	0	-5	0
Test Cov. Item covered	TCOVER2 c/work mark	TCOVER2 exam & c/work marks	TCOVER3 exam mark	TCOVER3 c/work mark
Exp. Output	'FM'	'FM'	'FM'	'FM'

Table 78 — Test cases for error guessing continued

Test Case	9	10	11	12
Input (exam mark)	-25	20	1234567890	25
Input (c/w mark)	-50	55	20	1234567890
total mark (as calculated)	-75	75	1234567910	1234567915
Test Cov. Item covered	TCOVER3 exam & c/work mark	TCOVER4 exam & c/work mark	TCOVER5 exam mark	TCOVER5 exam mark
Exp. Output	‘FM’	‘FM’	‘FM’	‘FM’

Table 79 — Test cases for error guessing continued

Test Case	13	14	15	16
Input (exam mark)	1234567890	abcdefghij	25	abcdefghij
Input (c/w mark)	1234567890	20	abcdefghij	abcdefghij
total mark (as calculated)	2469135780	NULL	NULL	NULL
Test Cov. Item covered	TCOVER5 exam & c/work mark	TCOVER6 exam mark	TCOVER6 c/work mark	TCOVER6 exam & c/work mark
Exp. Output	‘FM’	‘FM’	‘FM’	‘FM’

D.2.1.7 Step 5: Assemble Test Sets (TD5)

Since all test cases are invalid in that they should cause a fault message to appear, they can all be placed into the one test set as follows:

TS1: TEST CASES 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16.

D.2.1.8 Step 6: Derive Test Procedures (TD6)

Only one procedure is required as follows:

TP1: covering the test case in TS1, in the order specified in the test set.

D.2.1.9 Error Guessing Test Coverage

As stated in clause 6.3.1, there is no approach for calculating coverage of test coverage items for error guessing.

Annex E (informative) Test Design Technique Coverage Effectiveness

E.1 Test Design Technique Coverage Effectiveness

E.1.1 Guidance

Up to this point this standard has provided no guidance on either the choice of test design techniques or test completion criteria (sometimes known as test adequacy criteria), other than that they should be selected from clauses 5 and 6 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 risk, 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.

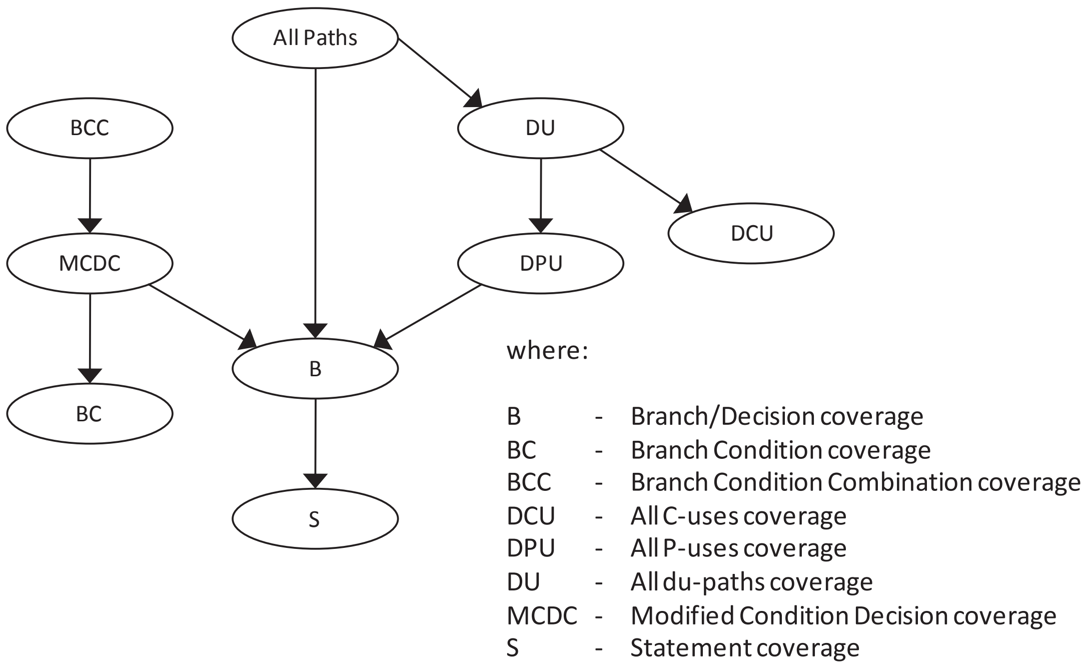
There is no requirement to choose corresponding test case design and test coverage measurement approaches. Specification-based test design techniques are effective at detecting errors of omission, while structure-based test design techniques can only detect errors of commission. So a test plan could 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 6 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 test items and their test bases, 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 to 100% will be achieved as well.

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

Not all test coverage criteria can be related by the subsumes ordering and the specification-based and structure-based criteria are not related at all. A partial ordering of criteria is possible for structure-based test design techniques, as illustrated in Figure 20 below, 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.



where:

B	-	Branch/Decision coverage
BC	-	Branch Condition coverage
BCC	-	Branch Condition Combination coverage
DCU	-	All C-uses coverage
DPU	-	All P-uses coverage
DU	-	All du-paths coverage
MCDC	-	Modified Condition Decision coverage
S	-	Statement coverage

Figure 20 — Partial Ordering of Structural Test Coverage Criteria (Reid 1996)

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 the figure above 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 test items, 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 test item that will only fail if that variable takes a negative value). Satisfying data flow criteria can concentrate the testing on these aspects of a test item'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 the figure above is correct for the definitions in clause 5 it may not apply to alternative definitions used elsewhere.

Bibliography

□AT□, □. and □c□A□ □., 200□. *The Software Test Engineer's Handbook*. O'Reilly Media, Inc.

□EI□ER, □., 1□□5. *Black Box Testing. Techniques for Functional Testing of Software and Systems*, □ohn Wiley □ Sons Inc.

Non-Functional Testing. □ritish Computer Society Special Interest □roup in Software Testing, □viewed □ September 2011□. Available from http□//www.testingstandards.co.uk/non□functional□testing□techniques.htm

□RITIS□ STANDARDS INSTITUTE, 1□□□. □S □□25-2□1□□□, *Software testing – Software component testing*. Available from□http□//shop.bsigroup.com/

□URNSTEIN, I., 2003. *Practical Software Testing: A Process-Oriented Approach*. Springer-□erlag.

CO□E□AND, □., 200□. *A Practitioner's Guide to Software Test Design*. Artech □ouse, Inc.

C□O, C. □., 1□□□. *Quality Programming*. Wiley.

C□OW, T. S., 1□□□. Testing Software Design □odelled by Finite-State □achines. In□*IEEE Transactions on Software Engineering*, □ol. **SE-4**(3).

CRAI□, R. and □AS□IE□, S., 2002. *Systematic Software Testing*. Artech □ouse Inc.

□RINDA□, □., OFFUTT, □. and AND□ER, S., 2005. Combination Testing Strategies□A Survey. In□*Software Testing, Verification and Reliability*, □ohn Wiley □ Sons □td., **15**, pp. 16□-1□□.

□ROC□T□ANN, □. and □RI□□, □rimm, □., 1□□3. Classification Trees for □artition Testing. In□*Software Testing, Verification & Reliability*, Wiley, **3**(2), pp. 63 - □2.

□ASS, A. □. □onassen, 200□. *Guide to Advanced Software Testing*. Artech □ouse.

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 2005. ISO/IEC TR 1□□5□, *Software Engineering -- Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Available from□http□//www.iso.org/

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 2011. ISO/IEC 25010, *Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Available from□ http□//www.iso.org

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 200□. ISO/IEC ISO/IEC 25020, *Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Measurement reference model and guide*. Available from□http□//www.iso.org

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 2012. ISO/IEC 25021, *Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) – Quality measure elements*. Available from□ http□//www.iso.org

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 2013. ISO/IEC CD 25022, *Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) – Measurement of quality in use*. Available from□ http□//www.iso.org

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 2013. ISO/IEC CD 25023, *Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality*. Available from□http□//www.iso.org

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 2013. ISO/IEC CD 2502□, *Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) – Measurement of data quality*. Available from□ http□//www.iso.org

INTERNATIONA□ OR□ANI□ATION FOR STANDARDISATION/INTERNATIONA□ E□ECTROTEC□NICA□ CO□□ITTEE, 200□. ISO/IEC 25030, *Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Quality requirements*. Available from□http□//www.iso.org

□ANER, C., 1□□□. *Testing Computer Software.* TA□ □ooks Inc.

□ERNI□□AN, □. W. and RIC□IE, D. □., 1□□□. *The C Programming Language*. □rentice-□all Software Series.

□IT, E., 1□□5. *Software Testing in the Real World: Improving the Process*. AC□ □ress.

□AND□, R. 1□□5. Orthogonal □atin Squares□An Application of Experiment Design to Compiler Testing. In *Communications of the ACM*, **28**(10), pp. 105□-105□.

□□ERS, □., 1□□□. *The Art of Software Testing*. □ohn Wiley □ Sons Inc.

NURSI□U□U, □. and □RO□ERT, R. □., 1□□5. Cause-Effect □raphing Analysis and □alidation of Requirements. In *Proceedings of CASCON'1995*.

REID, S., 1□□6. □opular □isconceptions in □odule Testing. In *Proceedings of the Software Testing Conference (STC)*, Washington DC.