INTERNATIONAL STANDARD

ISO/IEC/IEEE 29119-4

Second edition
2021-10

Software and systems engineering — Software testing —

Part 4:
Test techniques

Ingénierie du logiciel et des systèmes — Essais du logiciel —

Partie 4: Techniques d'essai

Reference number
ISO/IEC/IEEE 29119-4:2021(E)

Contents

Page

Foreword ... **v**

Introduction ... **vii**

1 Scope ... **1**

2 Normative references ... **1**

3 Terms and definitions ... **1**

4 Conformance ... **7**
4.1 Intended usage ... 7
4.2 Full conformance ... 7
4.3 Tailored conformance ... 7

5 Test design techniques ... **8**
5.1 Overview ... 8
5.2 Specification-based test design techniques ... 10
5.2.1 Equivalence partitioning ... 10
5.2.2 Classification tree method ... 12
5.2.3 Boundary value analysis ... 12
5.2.4 Syntax testing ... 14
5.2.5 Combinatorial test design techniques ... 15
5.2.6 Decision table testing ... 18
5.2.7 Cause-effect graphing ... 18
5.2.8 State transition testing ... 19
5.2.9 Scenario testing ... 20
5.2.10 Random testing ... 21
5.2.11 Metamorphic testing ... 21
5.2.12 Requirements-based testing ... 22
5.3 Structure-based test design techniques ... 23
5.3.1 Statement testing ... 23
5.3.2 Branch testing ... 23
5.3.3 Decision testing ... 24
5.3.4 Branch condition testing ... 25
5.3.5 Branch condition combination testing ... 25
5.3.6 Modified condition/decision coverage (MCDC) testing ... 26
5.3.7 Data flow testing ... 27
5.4 Experience-based test design techniques ... 29
5.4.1 Error guessing ... 29

6 Test coverage measurement ... **30**
6.1 Overview ... 30
6.2 Test measurement for specification-based test design techniques ... 30
6.2.1 Equivalence partition coverage ... 30
6.2.2 Classification tree method coverage ... 30
6.2.3 Boundary value analysis coverage ... 31
6.2.4 Syntax testing coverage ... 31
6.2.5 Combinatorial test design techniques coverage ... 31
6.2.6 Decision table testing coverage ... 32
6.2.7 Cause-effect graphing coverage ... 32
6.2.8 State transition testing coverage ... 32
6.2.9 Scenario testing coverage ... 33
6.2.10 Random testing coverage ... 33
6.2.11 Metamorphic testing coverage ... 33
6.2.12 Requirements-based testing coverage ... 33
6.3 Test measurement for structure-based test design techniques ... 33
6.3.1 Statement testing coverage ... 33
6.3.2 Branch testing coverage ... 33

6.3.3 Decision testing coverage 34
6.3.4 Branch condition testing coverage 34
6.3.5 Branch condition combination testing coverage 34
6.3.6 Modified condition/decision coverage (MCDC) 34
6.3.7 Data flow testing coverage 35
6.4 Test measurement for experience-based testing design techniques — Error guessing coverage 35
Annex A (informative) **Testing quality characteristics** **36**
Annex B (informative) **Guidelines and examples for the application of specification-based test design techniques** **49**
Annex C (informative) **Guidelines and examples for the application of structure-based test design techniques** **102**
Annex D (informative) **Guidelines and examples for the application of experience-based test design techniques** **122**
Annex E (informative) **Guidelines and examples for the application of grey-box test design techniques** **125**
Annex F (informative) **Test design technique effectiveness** **128**
Annex G (informative) **ISO/IEC/IEEE 29119-4 and BS 7925-2 test design technique alignment** **131**
Annex H (informative) **Test models** **133**
Bibliography **134**
IEEE Notices and Abstract **136**

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO/IEC documents should be noted. This document was drafted in accordance with the rules given in the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see https://patents.iec.c).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

ISO/IEC/IEEE 29119-4 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*, Subcommittee SC 7, *Software and systems engineering*, in cooperation with the Systems and Software Engineering Standards Committee of the IEEE Computer Society, under the Partner Standards Development Organization cooperation agreement between ISO and IEEE.

This second edition cancels and replaces the first edition (ISO/IEC/IEEE 29119-4:2015), which has been technically revised.

The main changes compared to the previous edition are as follows:

— The test techniques in this document are defined using a new form of the test design and implementation process from ISO/IEC/IEEE 29119-2. In the first version, this process was based on the use of test conditions. Feedback on use of the previous edition highlighted a problem with users' understanding of test conditions and their use for deriving test cases. This second edition has replaced the use of test conditions with test models. Annex H provides more detail on this change and the Introduction describes the new process.

A list of all parts in the ISO/IEC/IEEE 29119 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The purpose of this document 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 within the test design and implementation process that is defined in ISO/IEC/IEEE 29119-2. This document does not describe a process for test design and implementation. The intent is to describe a series of techniques that have wide acceptance in the software testing industry. This document is originally based on the British standard, BS 7925-2. Annex G provides a mapping from the requirements of BS 7925-2 to the clauses and subclauses of this document.

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

NOTE A "test item" is a work product that is being tested (see ISO/IEC/IEEE 29119-1).

EXAMPLE 1 "Test items" include systems, software items, objects, classes, requirements documents, design specifications, and user guides.

Each technique follows the test design and implementation process that is defined in ISO/IEC/IEEE 29119-2 and shown in Figure 1.

Of the activities in this process, this document provides guidance on how to implement the following activities in detail for each technique that is described:

— create test model (TD1);

— identify test coverage items (TD2);

— derive test cases (TD3).

A test model represents testable aspects of a test item, such as a function, transaction, feature, quality attribute, or structural element identified as a basis for testing. The test model reflects the required test completion criterion in the test strategy.

EXAMPLE 2 If a test completion criterion for state transition testing was identified that required coverage of all states then the test model would show the states the test item can be in.

Test coverage items are attributes of the test model that can be covered during testing. A single test model will typically be the basis for several test coverage items.

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 create test procedures activity (TD4) in the test design and implementation process of ISO/IEC/IEEE 29119-2 is not included in Clauses 5 or 6 because the process is the same for all techniques.

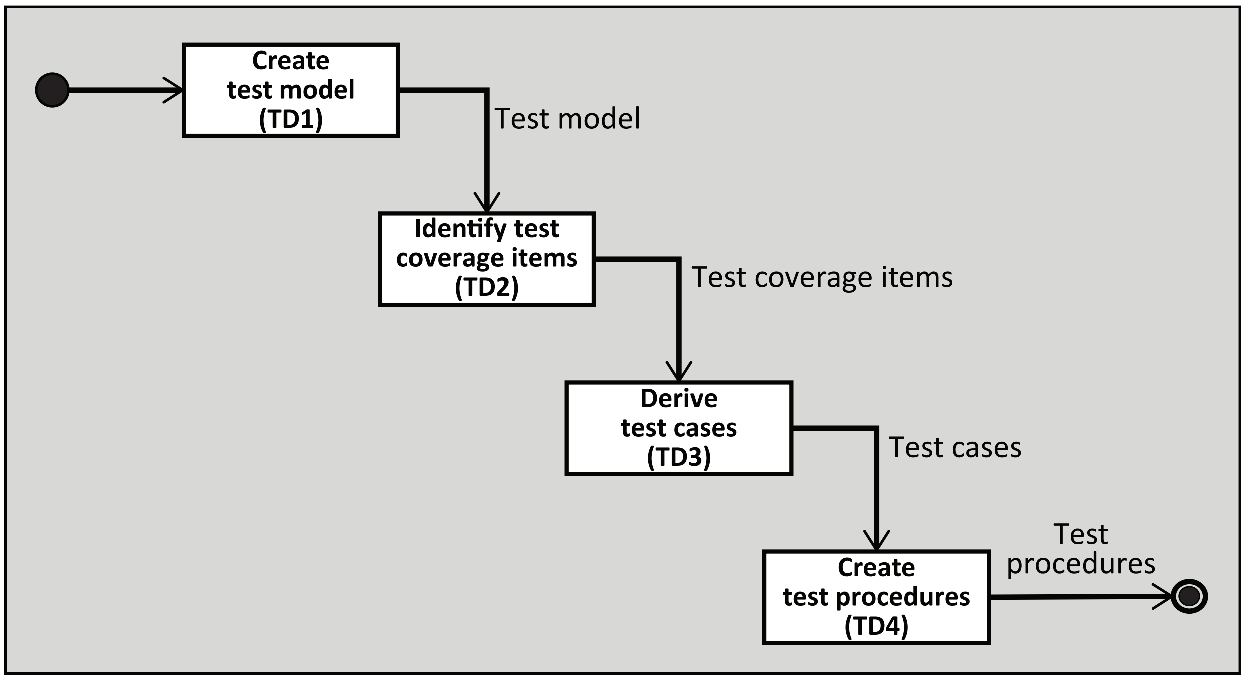


Figure 1 — ISO/IEC/IEEE 29119-2 test design and implementation process

ISO/IEC 25010 defines eight quality characteristics (including functional suitability) 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.

Experience-based testing practices like exploratory testing and other test practices such as model-based testing are not defined in this document because this document only describes techniques for designing test cases. Test practices such as exploratory testing, which can use the test techniques defined in this document, are described in ISO/IEC/IEEE 29119-1.

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

This document aims to provide stakeholders with the ability to design test cases for software testing in any organization.

Software and systems engineering — Software testing —

Part 4:
Test techniques

1 Scope

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

Each technique follows the test design and implementation process that is defined in ISO/IEC/IEEE 29119-2 and shown in Figure 1. This document is intended for, but not limited to, testers, test managers, and developers, particularly those responsible for managing and implementing software testing.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC/IEEE 29119-2, *Software and systems engineering — Software testing — Part 2: Test processes*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO, IEC and IEEE maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

— IEEE Standards Dictionary Online: available at https://ieeexplore.ieee.org/xpls/dictionary.jsp

NOTE For additional terms and definitions in the field of systems and software engineering, see ISO/IEC/IEEE 24765, which is published periodically as a “snapshot” of the SEVOCAB (Systems and software Engineering Vocabulary) database and is publicly accessible at https://www.computer.org/sevocab.

3.1
Backus-Naur Form
formal meta-language used for defining the syntax of a language in a textual format

3.2
base choice
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

3.3
boundary value analysis
specification-based *test case* (3.49) design technique based on exercising the boundaries of *equivalence partitions* (3.28)

3.4
branch condition combination testing
structure-based *test case* (3.49) design technique based on exercising combinations of Boolean values of *conditions* (3.13) within a decision

3.5
branch condition testing
structure-based *test case* (3.49) design technique based on exercising Boolean values of the *conditions* (3.13) within decisions and the *decision outcomes* (3.21)

3.6
branch testing
structure-based *test case* (3.49) design technique based on exercising branches in the *control flow* (3.14) of the *test item* (3.52)

3.7
c-use
computation data use
use of the value of a variable in any type of statement

3.8
cause-effect graph
graphical representation of *decision rules* (3.22) between causes (inputs described as Boolean conditions) and effects (outputs described as Boolean expressions)

3.9
cause-effect graphing
specification-based *test case* (3.49) design technique based on exercising *decision rules* (3.22) in a *cause-effect graph* (3.8)

3.10
classification tree
hierarchical tree model of the input data to a program in which the inputs are represented by distinct classifications (relevant test aspects) and classes (input values)

3.11
classification tree method
specification-based *test case* (3.49) design technique based on exercising classes in a *classification tree* (3.10)

3.12
combinatorial test design techniques
class of specification-based *test case* (3.49) design techniques based on exercising combinations of *P-V pairs* (3.38)

EXAMPLE All combinations *testing* (3.57), pair-wise testing, each choice testing, base choice testing

3.13
condition
Boolean expression containing no Boolean operators

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

3.14
control flow
sequence in which operations are performed during the execution of a *test item* (3.52)

3.15
control flow sub-path
sequence of *executable statements* (3.30) within a test *item* (3.52)

3.16
data definition
variable definition
statement where a variable is assigned a value

3.17
data definition c-use pair
data definition (3.16) and subsequent *computation data use* (3.7), where the *data use* (3.20) uses the value defined in the data definition

3.18
data definition p-use pair
data definition (3.16) and subsequent *predicate data use* (3.37), where the *data use* (3.20) uses the value defined in the data definition

3.19
data flow testing
class of structure-based *test case* (3.49) design techniques based on exercising *definition-use pairs* (3.26)

EXAMPLE All-definitions *testing* (3.57), all-c-uses testing, all-p-uses testing, all-uses testing, all-du-paths testing.

3.20
data use
executable statement (3.30) where the value of a variable is accessed

3.21
decision outcome
result of a decision that determines the branch to be executed

3.22
decision rule
combination of *conditions* (3.13) (also known as causes) and actions (also known as effects) that produce a specific outcome in *decision table testing* (3.24) and *cause-effect graphing* (3.9)

3.23
decision table
tabular representation of *decision rules* (3.22) between causes (inputs described as Boolean *conditions* (3.13)) and effects (outputs described as Boolean expressions)

3.24
decision table testing
specification-based *test case* (3.49) design technique based on exercising *decision rules* (3.22) in a *decision table* (3.23)

3.25
decision testing
structure-based *test case* (3.49) design technique based on exercising *decision outcomes* (3.21) in the *control flow* (3.14) of the *test item* (3.52)

3.26
definition-use pair
data definition-use pair
data definition (3.16) and subsequent *predicate* (3.40) or computational *data use* (3.20), where the data use uses the value defined in the data definition

3.27
entry point
point in a *test item* (3.52) at which execution of the test item can begin

Note 1 to entry: An entry point is an *executable statement* (3.30) within a test item that can be selected by an external process as the starting point for one or more *paths* (3.39) through the test item. It is most commonly the first executable statement within the test item.

3.28
equivalence partition
equivalence class
class of inputs or outputs that are expected to be treated similarly by the *test item* (3.52)

3.29
equivalence partitioning
specification-based *test case* (3.49) design technique based on exercising *equivalence partitions* (3.28)

3.30
executable statement
statement which, when compiled, is translated into object code, which will be executed procedurally when the *test item* (3.52) is running and may perform an action on program data

3.31
exit point
last *executable statement* (3.30) within a *test item* (3.52)

Note 1 to entry: An exit point is a terminal point of a *path* (3.39) through a test item, being an executable statement within the test item which either terminates the test item or returns control to an external process. This is most commonly the last executable statement within the test item.

3.32
expected result
observable predicted behaviour of the *test item* (3.52) under specified *conditions* (3.13) based on its specification or another source

3.33
experience-based testing
class of *test case* (3.49) design techniques based on using the experience of testers to generate test cases

EXAMPLE Error guessing.

Note 1 to entry: Experience based *testing* (3.57) can include concepts such as test attacks, tours, and error taxonomies which target potential problems such as security, performance, and other quality areas.

3.34
metamorphic relation
description of how changes to the test inputs for a *test case* (3.49) affect the expected outputs based on the required behaviour of a *test item* (3.52)

3.35
metamorphic testing
specification-based *test case* (3.49) design technique based on generating test cases on the basis of existing test cases and *metamorphic relations* (3.34)

3.36
modified condition/decision coverage testing
MCDC testing
structure-based *test case* (3.49) design technique based on demonstrating that a single Boolean *condition* (3.13) within a decision can independently affect the outcome of the decision

3.37
p-use
predicate data use
data use (3.20) associated with the *decision outcome* (3.21) of the *predicate* (3.40) portion of a decision statement

3.38
P-V pair
parameter-value pair
combination of a *test item* (3.52) parameter with a value assigned to that parameter, used as a test coverage item in *combinatorial test design techniques* (3.12)

3.39
path
sequence of *executable statements* (3.30) of a test *item* (3.52)

3.40
predicate
logical expression which evaluates to TRUE or FALSE to direct the execution *path* (3.39) in code

3.41
random testing
specification-based *test case* (3.49) design technique based on generating test cases to exercise randomly selected *test item* (3.52) inputs

3.42
requirements-based testing
specification-based *test case* (3.49) design technique based on exercising atomic requirements

EXAMPLE An atomic requirement can be 'The system shall collect and store the date of birth of all registered users.'

3.43
scenario testing
specification-based *test case* (3.49) design technique based on exercising sequences of interactions between the *test item* (3.52) and other systems

Note 1 to entry: Users are considered to be other systems in this context.

3.44
state transition testing
specification-based *test case* (3.49) design technique based on exercising transitions in a state model

EXAMPLE Example state models are state transition diagram and state table.

3.45
statement testing
structure-based *test case* (3.49) design technique based on exercising *executable statements* (3.30) in the source code of the *test item* (3.52)

3.47
sub-path
path (3.39) that is part of a larger path

3.46
syntax testing
specification-based *test case* (3.49) design technique based on exercising elements of a formal definition of the *test item* (3.52) inputs

EXAMPLE *Backus-Naur Form* (3.1) is commonly used for defining the syntax of test item inputs.

3.48
test
activity in which a system or component is executed under specified *conditions* (3.13), the results are observed or recorded, and an evaluation is made of some aspect of the system or component

3.49
test case
set of preconditions, inputs and *expected results* (3.32), developed to drive the execution of a *test item* (3.52) to meet *test objectives* (3.55)

Note 1 to entry: A test case is the lowest level of test implementation documentation (i.e. test cases are not made up of test cases) for the *test level* (3.53) or *test type* (3.56) for which it is intended.

Note 2 to entry: Test case preconditions include the required state of the test environment, data (e.g. databases) used by the test item, and the test item itself.

Note 3 to entry: Inputs are the data information and actions, where applicable, used to drive *test execution* (3.51).

3.50
test condition
testable aspect of a component or system, such as a function, transaction, feature, quality attribute, or structural element identified as a basis for *testing* (3.57)

Note 1 to entry: ISO/IEC/IEEE 29119 (all parts) does not use the concept of test conditions, but instead uses the concept of a *test model* (3.54) for test design. See Annex H for an explanation.

3.51
test execution
process of running a *test* (3.48) on the *test item* (3.52), producing actual results

3.52
test item
test object
work product to be tested

EXAMPLE Software component, system, requirements document, design specification, user guide.

3.53
test level
one of a sequence of test stages, each of which is typically associated with the achievement of particular objectives and used to treat particular risks

EXAMPLE The following are common test levels, listed sequentially: unit/component *testing* (3.57), integration testing, system testing, system integration testing, acceptance testing.

Note 1 to entry: It is not always necessary for a *test item* (3.52) to be tested at all test levels, but the sequence of test levels generally stays the same.

Note 2 to entry: Typical objectives can include consideration of basic functionality for unit/component testing, interaction between integrated components for integration testing, acceptability to end users for acceptance testing

3.54
test model
representation of the *test item* (3.52), which allows the *testing* (3.57) to be focused on particular characteristics or qualities

EXAMPLE Requirements statements, *equivalence partitions* (3.28), state transition diagram, use case description, *decision table* (3.23), input syntax description, source code, *control flow* (3.14) graph, parameters and values, *classification tree* (3.10), natural language.

Note 1 to entry: The test model and the required test coverage are used to identify test coverage items.

Note 2 to entry: A separate test model can be required for each different type of required test coverage included in the test completion criteria.

Note 3 to entry: A test model can include one or more *test conditions* (3.50).

Note 4 to entry: Test models are commonly used to support test design (e.g. they are used to support test design in this document, and they are used in model-based testing). Other types of models exist to support other aspects of testing, such as test environment models, test maturity models and test architecture models.

3.55
test objective
reason for performing *testing* (3.57)

EXAMPLE Checking for correct implementation, identification of defects, measuring quality.

3.56
test type
testing (3.57) that is focused on specific quality characteristics

EXAMPLE Security testing, functional testing, usability testing, and performance testing.

Note 1 to entry: A test type can be performed at a single *test level* (3.53) or across several test levels (e.g. performance testing performed at a unit test level and at a system test level).

3.57
testing
set of activities conducted to facilitate discovery and evaluation of properties of *test items* (3.52)

Note 1 to entry: Testing activities include planning, preparation, execution, reporting, and management activities, insofar as they are directed towards testing.

4 Conformance

4.1 Intended usage

The requirements in this document are contained in Clauses 5 and 6. It is recognized that particular projects or organizations will not need to use all of the techniques defined by this document. Implementation of ISO/IEC/IEEE 29119-2 involves using a risk-based approach to select a subset of techniques suitable for a given project or organization. There are two ways that an organization or individual can claim conformance to the provisions of this document – full conformance and tailored conformance.

The organization shall assert whether it is claiming full or tailored conformance to this document.

4.2 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 in Clause 5 and the corresponding test coverage measurement approaches in Clause 6 have been satisfied.

EXAMPLE An organization can 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 this document.

4.3 Tailored conformance

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

5 Test design techniques

5.1 Overview

This document defines test design techniques for specification-based testing (5.2), structure-based testing (5.3) and experience-based testing (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 this document are classified as specification-based, structure-based or experience-based, in practice some of them can be used interchangeably (e.g. branch testing can be applied to the specification of the graphical user interface of a web-based system). This is demonstrated in Annex E. In addition, although each technique is defined independently of all others, in practice some can be used in combination with other techniques.

EXAMPLE The test coverage items derived by applying equivalence partitioning can be used to identify the input parameters of test cases derived for scenario testing.

This document uses the terms specification-based testing and structure-based testing; 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".

This document defines how the generic test design and implementation process steps TD1(create test model), TD2 (identify test coverage items), and TD3 (derive test cases) from ISO/IEC/IEEE 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. Annex B, Annex C, Annex D and Annex E provide detailed examples that demonstrate how to apply the techniques.

The techniques that are defined this document are shown in Figure 2. This set of techniques is not exhaustive. There are useful techniques that are used by testing practitioners or researchers that are not included in this document.

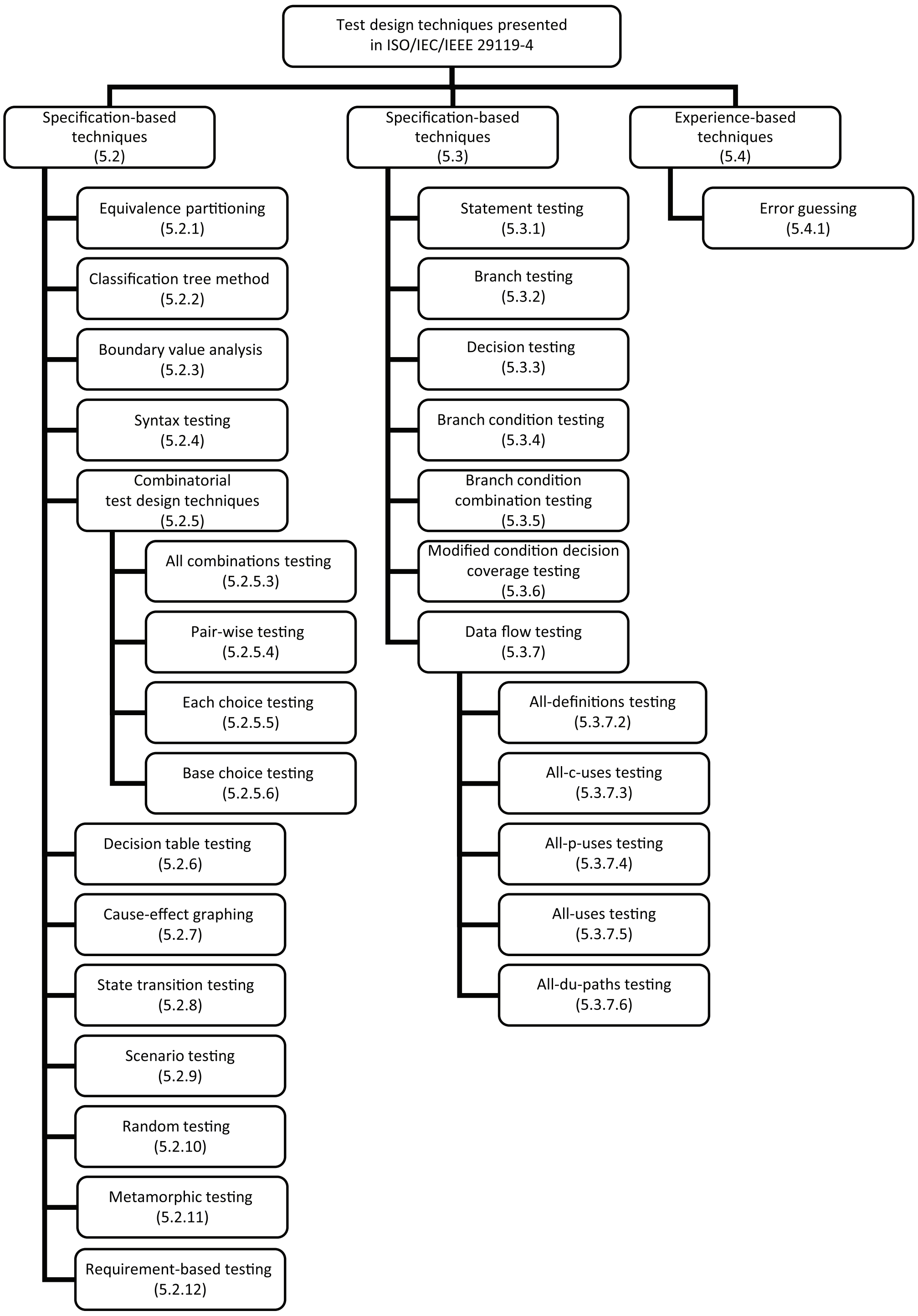

Test design techniques presented in ISO/IEC/IEEE 29119-4
Specification-based techniques (5.2)
Specification-based techniques (5.3)
Experience-based techniques (5.4)
Equivalence partitioning (5.2.1)
Classification tree method (5.2.2)
Boundary value analysis (5.2.3)
Syntax testing (5.2.4)
Combinatorial test design techniques (5.2.5)
All combinations testing (5.2.5.3)
Pair-wise testing (5.2.5.4)
Each choice testing (5.2.5.5)
Base choice testing (5.2.5.6)
Decision table testing (5.2.6)
Cause-effect graphing (5.2.7)
State transition testing (5.2.8)
Scenario testing (5.2.9)
Random testing (5.2.10)
Metamorphic testing (5.2.11)
Requirement-based testing (5.2.12)
Statement testing (5.3.1)
Branch testing (5.3.2)
Decision testing (5.3.3)
Branch condition testing (5.3.4)
Branch condition combination testing (5.3.5)
Modified condition decision coverage testing (5.3.6)
Data flow testing (5.3.7)
All-definitions testing (5.3.7.2)
All-c-uses testing (5.3.7.3)
All-p-uses testing (5.3.7.4)
All-uses testing (5.3.7.5)
All-du-paths testing (5.3.7.6)
Error guessing (5.4.1)

Figure 2 — The set of test design techniques presented in this document

Of the four activities in the test design and implementation process (see Figure 1), test design techniques provide unique and specific guidance on the derivation of test models (TD1), identification of test coverage items (TD2) and the derivation of test cases (TD3). Therefore, each technique is defined in terms of these three activities.

There are varying levels of granularity within steps TD1 (create test model), TD2 (identify test coverage items) and TD3 (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 coverage items in step TD2 (e.g. a control flow model is normally used as a test model for identifying test coverage items for all structural techniques).

EXAMPLE 1 In state transition testing, if there is a requirement to cover all states then the state model for the test item would form the test model. If there is a requirement to cover all transitions between states, then each transition would be a test coverage item.

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 activity (TD3) 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 5.2.1.4 and 5.2.3.3).

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

Although the techniques defined in this document are each described in a separate clause (as if they were mutually exclusive), in practice they can be applied in a blended way.

EXAMPLE 3 Boundary value analysis can be used to select test input values, after which pair-wise testing can be used to design complete test cases from the test input values. Similarly, equivalence partitioning can be used to select the classifications and classes for the classification tree method and then each choice testing can be used to construct test cases from the classes.

The techniques presented in this document can also be used in conjunction with the test types that are presented in Annex A. For example, equivalence partitioning can be used to identify user groups and then representative users (test coverage items) from those groups to create test cases for usability testing.

The definitions of the techniques are provided in Clause 5. The corresponding coverage measures for each technique are presented in Clause 6. These are supported by examples of each technique in Annexes B, C, D and E. 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.

EXAMPLE 4 Statement coverage analysers can be used to support structure-based testing.

Annex A provides examples of how the test design techniques defined in this document can be applied to testing the non-functional quality characteristics defined in ISO/IEC 25010.

5.2 Specification-based test design techniques

5.2.1 Equivalence partitioning

5.2.1.1 General

Equivalence partitioning (BS 7925-2; Myers 1979[19]) is based on exercising equivalence partitions. These equivalence partitions are derived based on the expectation that values within a partition should be treated similarly by the test item.

5.2.1.2 Create test model (TD1)

A test model shall be created that identifies the equivalence partitions for the test item. A suitable test model shows the various functions expected to be performed by the test item from which equivalence partitions can be derived. These functions can be considered as a set of processes, which can be decomposed into input processes, internal processes and output processes.

The functions described in the test item's specification can typically be used to identify its internal processes. Consideration of the inputs and outputs of the test item can be used to help determine its probable input and output processes.

Equivalence partitions should be identified for both valid and invalid functions (leading to both valid and invalid outputs) and valid and invalid inputs to the test item. In this context, invalid functions are those that are not included in the specification and which cannot reasonably be expected to be included in a correct implementation of the test item. Similarly, invalid inputs are those that should either be ignored by the test item or cause the test item to output an error message.

Equivalence partitions for a test item can overlap, and in some situations, one partition can be a subset of another partition. By adding detail to the test model, further equivalence partitions can often be derived, often as subsets of existing partitions. As more detail and partitions are added, the number of test cases required to fully exercise the test model is likely to increase.

EXAMPLE For a test item expecting lowercase alphabetical characters as (valid) inputs, invalid input partitions can be derived containing integers, reals, uppercase alphabetical characters, symbols and control characters, depending on the level of rigour required during testing. If each of these partitions were included separately in the model, each would be expected to be exercised to achieve full coverage, whereas if a single partition was created to cover all invalid inputs, then only a single partition would need to be exercised to achieve full partition coverage.

NOTE 1 Invalid equivalence partitions that correspond to inputs, functions or outputs that have not been specified (as is often the case) are typically based on the experience and imagination of the tester, which means different testers will often derive different invalid partitions. This subjective form of test design can also occur when applying experience-based techniques like error guessing.

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

5.2.1.3 Identify test coverage items (TD2)

Each equivalence partition shall be identified as a test coverage item.

5.2.1.4 Derive test cases (TD3)

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:

a) Decide on an approach for selecting combinations of test coverage items to be exercised by test cases. Two common approaches are (BS 7925-2; Myers 1979[19]):

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

   2) minimized, in which the minimum number of test cases is derived to cover each equivalence partition at least once.

   NOTE Other approaches to selecting combinations of test coverage items to be exercised by test cases are described in 5.2.5.

b) Select test coverage items for inclusion in the current test case based on the approach chosen in step a).

c) 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.

d) Determine the expected result of the test case by applying the inputs to the test basis.

e) Repeat steps b) to d) until the required level of test coverage is achieved.

5.2.2 Classification tree method

5.2.2.1 Create test model (TD1)

The classification tree method shall use a test model that partitions the input domain of the test item in terms of "classifications" and represents them graphically in the form of a classification tree. See Grochtmann and Grimm 1993[11] for more details. Each classification consists of a complete and disjoint (non-overlapping) set of "classes" (values the classifications can take). These classes may be further classified (even recursively) into sub-classifications and sub-classes, depending on the level of rigour required in the testing.

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) are completely disjoint, whereas in equivalence partitioning, they can overlap depending on how the technique is applied.

5.2.2.2 Identify test coverage items (TD2)

There are two approaches for identifying test coverage items for classification trees:

— When the required test coverage is based on "minimality", then the test coverage items shall be identified as the classes represented by leaf nodes in the classification tree.

— When the required test coverage is based on "maximality", then the test coverage items shall be identified as unique combinations of the classes represented by leaf nodes in the classification tree.

5.2.2.3 Derive test cases (TD3)

Test cases shall be derived to exercise the test coverage items. When deriving test cases, combinations of classes are included in each test case.

NOTE The combinations of classes are often illustrated in a combination table (see Figure B.1).

The following steps shall be followed during test case derivation:

a) Based on the selected coverage criterion, select a combination of classes for inclusion in the current test case that has not already been covered by a test case.

b) Identify input values for any classes that do not already have an assigned value in the current test case.

c) Determine the expected result of the test case by applying the inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.2.3 Boundary value analysis

5.2.3.1 Create test model (TD1)

Boundary value analysis shall use a model of the test item that partitions the inputs and outputs of the test item into a number of ordered sets and subsets (i.e. partitions and sub-partitions) with identifiable boundaries, where each boundary is then used to identify test coverage items. See BS 7925-2 and Myers 1979[19] for more details. These partitions correspond to those equivalence partitions defining ordered sets that would form the test model for equivalence partitioning (see 5.2.1.2).

EXAMPLE For a partition defined as integers from 1 to 10 inclusive, there are two boundaries, where the lower boundary is 1 and the upper boundary is 10, and these two input values would be covered by testing.

NOTE For output boundaries, corresponding input partitions are derived based on the processing described in the test item's specification. Test inputs are then selected from the input partitions that cause the output partitions to occur during testing.

5.2.3.2 Identify test coverage items (TD2)

One of the following two options for the identification of test coverage items shall be used:

— two-value boundary testing; or

— three-value boundary testing.

For two-value boundary testing, two test coverage items shall be identified for each boundary, 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 identified for each boundary, 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 can have only a single boundary identified in the test basis. For example, the numerical partition “age ≥ 70 years” has a specified 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 are ideally 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 can be required for certain circumstances (e.g. for rigorous testing to check that 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 B.2.3.5.

5.2.3.3 Derive test cases (TD3)

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

a) Decide on an approach for selecting combinations of test coverage items to be exercised by test cases. Two common approaches are (BS 7925-2; Myers 1979[19]):

   1) one-to-one, in which each test case is derived to exercise a specific test coverage item;

   2) minimized, in which test coverage items are included in test cases such that the minimum number of test cases are derived to cover all test coverage items at least once.

   NOTE Other approaches to selecting combinations of test coverage items to be exercised by test cases are described in 5.2.5.

b) Select test coverage items for inclusion in the current test case based on the approach chosen in step a).

c) Identify arbitrary valid values for any other input variables required by the test case that were not already selected in step b).

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

e) Repeat steps b) to d) until the required level of test coverage is achieved.

5.2.4 Syntax testing

5.2.4.1 Create test model (TD1)

Syntax testing shall use a formal model of the inputs to a test item as the basis for test design. See Beizer 1995[2] and Burnstein 2003[5] for more details. This syntax model is represented as a number of rules, where each rule defines the format of an input parameter in terms of "sequences of", "iterations of", or "selections between" elements in the syntax. The syntax may be represented in a textual or diagrammatic format.

EXAMPLE 1 Backus-Naur Form is a formal meta-language that can be used to define the syntax for the inputs to a test item in a textual format.

EXAMPLE 2 An abstract syntax tree can be used to represent formal syntax diagrammatically.

5.2.4.2 Identify test coverage items (TD2)

In syntax testing, test coverage items shall be identified based on two goals: positive testing, in which test coverage items are identified to cover the valid syntax in various ways, and negative testing, in which test coverage items are identified to deliberately violate the rules of the syntax. Test coverage items for positive testing are known as "options" of the defined syntax, whereas test coverage items for negative testing are known as "mutations" of the defined syntax.

The following guidelines may be used to identify "options" (although alternative guidelines may be used where appropriate):

a) Whenever selection is mandated by the syntax, an "option" is identified 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.

b) Whenever an iteration is mandated by the syntax, at least two "options" are identified 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 identified as test coverage items.

c) Whenever iteration is mandated with a maximum number of repetitions, at least two "options" are identified for the iteration; one with the maximum number of repetitions and the other with less 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 "less than 100 letters" will be identified as test coverage items.

To identify "mutations", the defined syntax may be mutated in various ways, often dependent on the experience and creativity of the tester.

EXAMPLE 4 For the input parameter "colour = Blue | Red | Green", one "mutation" can 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 B.2.4.5.

5.2.4.3 Derive test cases (TD3)

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

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

   1) one-to-one, in which each test case is derived to exercise a specific option or mutation; or

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

   NOTE 1 Other approaches to selecting combinations of test coverage items to be exercised by test cases are described in 5.2.5.

   NOTE 2 Using a minimized approach for mutations can lead to such unrealistic test inputs that it can be unlikely that the test item will fail to identify the test input as invalid.

b) Select test coverage items for inclusion in the current test case.

c) 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.

d) Determine the expected result of the test case by applying the inputs to the test basis.

e) Repeat steps b) to d) until all identified options and mutations are exercised.

5.2.5 Combinatorial test design techniques

5.2.5.1 Overview

Combinatorial test design techniques (Grindal, Offutt and Andler 2005[10]) are used to systematically derive a meaningful and manageable subset of test cases that cover the test model and the associated 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. Where numerous parameters (each with numerous discrete values) must interact, this technique enables a significant reduction in the number of test cases required without compromising functional coverage.

5.2.5.2 Create test model (TD1)

All combinatorial test design techniques shall use a test model comprising a set of P-V pairs that combines each of the test item parameters (P) taking on each of their possible values (V).

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 can 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 first be necessary to apply other test design techniques, such as equivalence partitioning or boundary value analysis, to reduce the large set of possible values for a parameter to a manageable subset.

5.2.5.3 All combinations testing

5.2.5.3.1 Identify test coverage items (TD2)

In ‘all combinations’ testing, the test coverage items shall be members of the set of all unique combinations of P-V pairs, such that each parameter is included at least once in the set. See Grindal, Offutt and Andler 2005[10] for more details.

5.2.5.3.2 Derive test cases (TD3)

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:

a) Select test coverage items for inclusion in the current test case that have not already been covered by a test case.

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

c) Repeat steps a) and b) until the required level of test coverage is achieved.

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

EXAMPLE A test item with two parameters for digits (10 possible values) and lower-case alphabetic characters (26 possible values) would result in 260 test cases.

5.2.5.4 Pair-wise testing

5.2.5.4.1 Identify test coverage items (TD2)

In pair-wise 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. See Grindal, Offutt and Andler 2005[10] for more details.

EXAMPLE A test item with two parameters for months (12 values) and gender (two values) results in 24 possible pairs of P-V pairs (e.g. January-female, January-male, February-female, February-male, March-female).

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 efficiently exercising the test item with fewer test cases. Pair-wise testing is also known as “all pairs” testing.

5.2.5.4.2 Derive test cases (TD3)

Having identified the P-V pairs, test cases shall be derived steps to exercise pairs of P-V pairs, where each test case exercises one or more unique pairs. The following shall be used during test case derivation:

a) Select test coverage items for inclusion in the current test case, where pairs of P-V pairs cover one or more pairs of parameter values that have not yet been included in a test case.

b) Identify arbitrary valid values for any other parameters present in the test case.

c) Determine the expected result of the test case by applying the inputs to the test basis.

d) Repeat steps a) to c) 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[18]) to determine a near-optimal set.

5.2.5.5 Each choice testing

5.2.5.5.1 Identify test coverage items (TD2)

In each choice (or 1-wise) testing, the test coverage items shall be members of the set of P-V pairs such that each parameter value is included at least once in the set. See Grindal, Offutt and Andler 2005[10] for more details.

5.2.5.5.2 Derive test cases (TD3)

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:

a) Select test coverage items for inclusion in the current test case, where at least one selected test coverage item has not been included in a prior test case.

b) Identify arbitrary valid values for any other parameters present in the test case.

c) Determine the expected result of the test case by applying the inputs to the test basis.

d) Repeat steps a) to c) 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.

EXAMPLE A test item with three parameters for digits (10 possible values), months (12 possible values) and lower-case alphabetic characters (26 possible values) would result in 26 test cases.

5.2.5.6 Base choice testing

5.2.5.6.1 Identify test coverage items (TD2)

In base choice testing, the test coverage items shall be sets of P-V pairs for each of the input parameters, where all parameters except one are set to their “base” value and the final parameter is set to one of its other valid values. See Grindal, Offutt and Andler 2005[10] for more details.

NOTE There are a number of approaches for choosing the base values for each parameter, but generally the base choice is the most ‘important’ value that the parameter can take. For example, the base values can be chosen from the operational profile, from the typical path in scenario testing, or from the most frequently used values for the parameter.

5.2.5.6.2 Derive test cases (TD3)

Having 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 remaining parameter to each of its valid values until the required level of test coverage is achieved. The following steps shall be used during test case derivation:

a) Derive the base-choice test case by setting each parameter to its “base” value.

b) 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.

c) Determine the expected result of the new test case by applying the inputs to the test basis.

d) Repeat steps b) and c) until the required level of test coverage is achieved.

5.2.6 Decision table testing

5.2.6.1 Create test model (TD1)

Decision table testing shall use 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 cause is expressed as a condition, which is either true or false (i.e. a Boolean) for an input, or combination of inputs;

— each effect is expressed as a Boolean expression represented by an output (i.e. action), or a combination of outputs;

— a set of decision rules define the required relationships between causes and effects.

See BS 7925-2 and Myers 1979[19] for more details.

NOTE If conditions consist of multiple values rather than simple Booleans, this results in an "extended entry" decision table, the testing of which can be handled by equivalence partitioning.

5.2.6.2 Identify test coverage items (TD2)

In decision table testing, the test coverage items shall be the decision rules, which define the relationship between a unique combination of the test item's conditions and outputs.

5.2.6.3 Derive test cases (TD3)

Test cases shall be derived to exercise each of the decision rules (test coverage items). The following steps shall be used during test case derivation:

a) Select a decision rule from the decision table for implementation as a test case.

b) Identify input values to satisfy the input conditions of the decision rule to be covered by the test case.

c) Determine the expected result of the test case by applying the inputs to the decision table.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

NOTE If the decision table contains dependent input conditions then this can result in infeasible combinations (e.g. two separate conditions, "age less than 18" and "age greater than 65" both set to true). In this situation such infeasible decision rules need to be identified and ideally documented, and do not need to be covered during testing.

5.2.7 Cause-effect graphing

5.2.7.1 Create test model (TD1)

Cause-effect graphing shall use 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 expressed as a condition, which is either true or false (i.e. a Boolean) for an input, or combination of inputs;

— each effect is expressed as a Boolean expression represented by an output, or a combination of outputs;

— a set of decision rules define the required relationships between causes and effects.

See BS 7925-2, Myers 1979[19], and Nursimulu and Probert 1995[20] for more details.

The cause-effect graph models logical relationships between causes and effects as a Boolean logic network with Boolean operators, and can optionally model syntactic and semantic constraints over relationships between causes and relationships between effects (see Figures B.5 and B.6).

5.2.7.2 Identify test coverage items (TD2)

In cause-effect graphing, the test coverage items shall be the decision rules, which define the relationship between a unique combination of the test item's causes and effects.

5.2.7.3 Derive test cases (TD3)

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:

a) Select a decision rule to be implemented in the current test case.

b) Identify input values to exercise the decision rule to be covered by the test case.

c) Determine the expected result of the test case by applying the inputs to the cause-effect graph (or corresponding decision table).

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.2.8 State transition testing

5.2.8.1 Create test model (TD1)

State transition testing shall use a model of the states the test item can occupy, the transitions between states, the events which cause transitions and the actions that can result from the transitions. See BS 7925-2 and Chow 1978[7] for more details. The states of the model shall be discrete, identifiable and finite in number. An individual transition may be constrained by an event guard, which defines a set of conditions that must be true when the event occurs, in order for the transition to occur. In state transition testing, the test coverage typically applies to states or transitions of the state model, depending on the required coverage. The model may take the form of a state transition diagram, a state table or another suitable representation.

5.2.8.2 Identify test coverage items (TD2)

In state transition testing, test coverage items will change depending on the chosen test completion criterion. Possible test completion criteria 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 valid sequences of N + 1 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 (TD3)

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:

a) Select test coverage items for inclusion in the current test case.

b) Identify input values to exercise the test coverage items to be covered by the test case.

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

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.2.9 Scenario testing

5.2.9.1 Create test model (TD1)

Scenario testing shall use a model of the sequences of possible 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. See Desikan and Ramesh 2007[9] for more details. The test model typically includes a set of scenarios related to the test item, where each scenario covers one sequence of interactions.

In scenario testing, the test model shall identify:

— 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

— "alternative" scenarios that represent alternative (non-main) scenarios that may be taken through the test item.

NOTE 1 Alternative scenarios can include abnormal use, extreme or stress conditions, exceptions and error handling.

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

NOTE 3 In many situations, it is good practice to have identified scenarios reviewed and agreed by relevant stakeholders, such as business users.

One common form of scenario testing is use case testing (Hass 2008[12]), which utilises 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) between the test item and actors.

NOTE 4 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 can be a user or another system.

NOTE 5 Transaction flow testing (Beizer 1995[2]) is often classified as a type of scenario testing.

5.2.9.2 Identify test coverage items (TD2)

The test coverage items shall be the main and alternative scenarios identified from the test model.

5.2.9.3 Derive test cases (TD3)

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:

a) Select a scenario to exercise in the current test case.

b) Identify input values to exercise the scenario covered by the test case.

c) Determine the expected result of the test case by applying the inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.2.10 Random testing

5.2.10.1 Create test model (TD1)

Random testing shall use a representation of the input domain of the test item that defines the set of all possible input values. See BS 7925-2, Craig and Jaskiel 2002[8], and Kaner 1988[16] for more details.

5.2.10.2 Identify test coverage items (TD2)

There are no recognized test coverage items for random testing.

5.2.10.3 Derive test cases (TD3)

Test cases for random testing shall be chosen by randomly selecting input values from the input domain of the test item according to the chosen input distribution. The following steps shall be used during test case derivation:

a) Select an input distribution for the selection of test inputs.

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

b) Generate random values for the test inputs based on the input distribution chosen in step a).

c) Determine the expected result of the test case by applying the inputs to the test basis.

d) Repeat steps b) and c) 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 b) is typically automated.

5.2.11 Metamorphic testing

5.2.11.1 Create test model (TD1)

The test model for metamorphic testing shall be a set of metamorphic relations, which define properties of the test item in relation to multiple inputs and their expected outputs. See Chen 2018[23] for more details. The properties defined by the metamorphic relation allow follow-up test cases to be generated from a source test case. A metamorphic relation for a test item describes how a change in the test inputs from the source test case to the follow-up test case affects a change (or not) in the expected outputs from the source test case to the follow-up test case.

EXAMPLE 1 A test item measures the distance between a start and end point. The source test case has test inputs A (start point) and B (end point) and an expected result C (distance) from running the test case. The metamorphic relation states that if the start and end points are swapped, then the expected result remains unchanged. Thus, a follow-up test case can be generated with B as the start point, A as the end point and C as the distance.

EXAMPLE 2 A test item predicts the age of death for an individual based on a set of lifestyle parameters. A source test case has various test inputs, including 10 cigarettes smoked per day, and an expected result of age 58 years from running the test case. The metamorphic relation states that if a person smokes more cigarettes, then their expected age of death will probably decrease (and not increase). Thus, a follow-up test case can be generated with the same input set of lifestyle parameters, except with the number of cigarettes smoked increased to 20 per day. The expected result (the predicted age of death) for this follow-up test case can now be set to less than or equal to 58 years.

NOTE In many situations, it is good practice to have identified metamorphic relations reviewed and agreed by relevant stakeholders, such as business users.

5.2.11.2 Identify test coverage items (TD2)

The test coverage items shall be the individual metamorphic relations, which comprise the test model.

5.2.11.3 Derive test cases (TD3)

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

a) Select a test coverage item (metamorphic relation) for inclusion in the current test case.

b) Create a source test case (or identify an existing source test case), execute it and validate the result.

c) Use the metamorphic relation to derive one or more follow-up test cases.

   NOTE 1 The expected result for the follow-up test case will not always be an exact value but will often be described as a function of the actual result achieved by executing the source test case (e.g. expected result for follow-up test case > actual result for source test case).

   NOTE 2 A single metamorphic relation can often be used to derive multiple follow-up test cases (e.g. a metamorphic relation for a function that translates speech into text can be used to generate multiple follow-up test cases with the same speech at different input volume levels but with the same text as the expected result).

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.2.12 Requirements-based testing

5.2.12.1 Create test model (TD1)

The test model for requirements-based testing shall be a set of atomic requirements. See Pinkster et al, 2006[24] for more details.

NOTE Requirements-based testing is typically performed when the requirements are specified as a list of atomic textual requirements, such as 'The system shall collect and store the date of birth of all registered users.'

5.2.12.2 Identify test coverage items (TD2)

The test coverage items shall be the individual atomic requirements, which comprise the test model.

5.2.12.3 Derive test cases (TD3)

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

a) Select a test coverage item (atomic requirement) for inclusion in the current test case.

b) Create a test case that exercises the requirement.

NOTE Test cases for requirements-based testing are typically positive test cases that confirm whether stated requirements are met. Negative test cases are normally derived using other, complementary, test techniques.

c) Repeat steps a) and b) until the required level of test coverage is achieved.

5.3 Structure-based test design techniques

5.3.1 Statement testing

5.3.1.1 Create test model (TD1)

A model of the source code of the test item, which identifies statements as either executable or non-executable, shall be identified. See BS 7925-2 and Myers 1979[19] for more details.

NOTE The developer's source code is typically used as the test model for statement testing.

5.3.1.2 Identify test coverage items (TD2)

Each executable statement shall be a test coverage item.

5.3.1.3 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path that reaches one or more executable statements that have not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the corresponding test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.3.2 Branch testing

5.3.2.1 Create test model (TD1)

A control flow model that identifies branches in the control flow of the test item shall be created. See BS 7925-2 for more details.

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

5.3.2.2 Identify test coverage items (TD2)

Each branch in the control flow model shall be a test coverage item.

NOTE Function and method calls are not identified as separate test coverage items in branch testing.

5.3.2.3 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path that reaches one or more branches that have not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

If there are no branches in the test item then a single test case is still required to achieve full branch coverage.

If there are multiple entry points to the test item, sufficient test cases will be required to cover each entry point.

5.3.3 Decision testing

5.3.3.1 Create test model (TD1)

A control flow model of the test item that identifies decisions shall be created. 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; Myers 1979[19]). 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).

5.3.3.2 Identify test coverage items (TD2)

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

5.3.3.3 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path that reaches one or more decision outcomes that have not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

If there are no decisions in the test model then a single test case is still required to achieve full decision coverage.

5.3.4 Branch condition testing

5.3.4.1 Create test model (TD1)

A control flow model of the test item that identifies decisions and conditions within decisions shall be created. See BS 7925-2 for more details.

5.3.4.2 Identify test coverage items (TD2)

In branch condition testing, the possible Boolean values (i.e. true or false) of the conditions within decisions and the decision outcomes from each decision shall be identified as test coverage items.

EXAMPLE The decision statement "if A OR B AND C then" contains six test coverage items related to the three logical operators and two test coverage items related to the two decision outcomes.

5.3.4.3 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify Boolean values of conditions within a decision that have not yet been executed during testing.

b) Determine the set of test inputs that will cause the decision to be exercised with the selected Boolean values of conditions.

c) Determine the expected result by applying the selected test inputs to the test basis.

d) Repeat steps a) to c) until all Boolean values of conditions within decisions have been exercised.

e) Identify a control flow sub-path that reaches one or more decision outcomes, if all of them have not yet been executed during testing.

f) Determine the test inputs that will cause the control flow sub-path to be exercised.

g) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

h) Repeat steps e) to g) until the required level of test coverage is achieved.

If there are no decisions in the test item then a single test case is still required to achieve full branch condition coverage.

5.3.5 Branch condition combination testing

5.3.5.1 Create test model (TD1)

A control flow model of the test item that identifies decisions and conditions within decisions shall be created. See BS 7925-2 for more details.

5.3.5.2 Identify test coverage items (TD2)

Each unique feasible combination of Boolean values of conditions within each decision shall be identified as a test coverage item. See BS 7925-2 and Myers 1979[19] for more details.

NOTE Where a decision includes a single condition, two possible "combinations" (true and false) are considered to exist.

EXAMPLE The decision statement “if A OR B AND C then” contains 8 test coverage items corresponding to all possible true and false combinations of the three conditions A, B and C. In general, a minimum of 2^N test coverage items are required when there are N conditions within a decision.

5.3.5.3 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a combination of Boolean values of conditions within a decision that have not yet been executed during testing.

b) Determine the set of test inputs that will cause the decision to be exercised with the selected combination of Boolean values of conditions.

c) Determine the expected result by applying the selected test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

If there are no decisions in the test item then a single test case is still required to achieve full branch condition combination coverage.

5.3.6 Modified condition/decision coverage (MCDC) testing

5.3.6.1 Create test model (TD1)

A control flow model of the test item that identifies decisions and conditions within decisions shall be created. See BS 7925-2 for more details.

5.3.6.2 Identify test coverage items (TD2)

Each unique feasible combination of individual Boolean values of conditions within a decision that allows a single Boolean condition to independently affect the outcome of the decision shall be identified as a test coverage item. Whether a condition independently affects the outcome of a decision is shown by varying only that condition while holding all other possible conditions within the decision to fixed states.

NOTE Where a decision includes a single condition, two possible “combinations” (true and false) are considered to exist and two test coverage items are identified.

EXAMPLE The decision statement “if A OR B AND C then” requires a minimum of four test coverage items to demonstrate that the three conditions can each independently affect the outcome of the decision. In general, a minimum of $N + 1$ test coverage items are required when there are N conditions within a decision.

5.3.6.3 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a combination of Boolean values of conditions defined by a test coverage item within a decision that have not yet been executed during testing.

b) Determine the set of test inputs that will cause the decision to be exercised with the selected combination of Boolean values of conditions.

c) Determine the expected result by applying the selected test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

If there are no decisions in the test item then a single test case is still required to achieve full MCDC coverage.

5.3.7 Data flow testing

5.3.7.1 Create test model (TD1)

In data flow testing, a model of the test item shall be derived that identifies control flow sub-paths through the test item, within which each definition of a given variable is linked to subsequent uses of the same variable and within which there has been no intervening redefinition of the variable's value. See Burnstein 2003[5] for more details.

"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). 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 or if-then-else. 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.

There are a number of forms of data flow testing, which are all based on the same test model. The five forms defined in this document 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 Identify test coverage items (TD2)

A set of control flow sub-paths from each variable definition to some use (either p-use or c-use) of that definition (with no intervening definitions) shall be identified as test coverage items.

5.3.7.2.2 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path from a variable definition to a subsequent use of that definition that has not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path from the identified definition to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

NOTE In practice, these steps can require automation.

5.3.7.3 All-c-uses testing

5.3.7.3.1 Identify test coverage items (TD2)

A set of control flow sub-paths from each variable definition to each c-use of that definition (with no intervening definitions) shall be identified as test coverage items.

5.3.7.3.2 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path from a variable definition to a subsequent c-use of that definition that has not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path from the identified definition to the subsequent c-use to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.3.7.4 All-p-uses testing

5.3.7.4.1 Identify test coverage items (TD2)

A set of control flow sub-paths from each variable definition to each p-use of that definition (with no intervening definitions) shall be identified as test coverage items.

5.3.7.4.2 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path from a variable definition to a subsequent p-use of that definition that has not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path from the identified definition to the subsequent p-use to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.3.7.5 All-uses testing

5.3.7.5.1 Identify test coverage items (TD2)

A set of control flow sub-paths from each variable definition to every use (both p-use and c-use) of that definition (with no intervening definitions) shall be identified as test coverage items.

5.3.7.5.2 Derive test cases (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path from a variable definition to a subsequent p-use or c-use of that definition that has not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path from the identified definition to the subsequent p-use or c-use to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.3.7.6 All-du-paths testing

5.3.7.6.1 Identify test coverage items (TD2)

The set of all loop-free control flow sub-paths from each variable definition to every use (both p-use and c-use) of that definition (with no intervening definitions) shall be identified as test coverage items.

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 (TD3)

The following steps shall be followed during test case derivation:

a) Identify a control flow sub-path from a variable definition to a subsequent p-use or c-use of that definition that has not yet been executed during testing.

b) Determine the test inputs that will cause the control flow sub-path from the identified definition to the subsequent p-use or c-use to be exercised.

c) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required level of test coverage is achieved.

5.4 Experience-based test design techniques

5.4.1 Error guessing

5.4.1.1 Create test model (TD1)

Error guessing shall use a test model in the form of a checklist of potential defects to focus the testing. See BS 7925-2 and Myers 1979[19] for more details. The checklist is used to identify test inputs that may cause failures, if those defects exist in the test item. The checklist constitutes the test model.

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

5.4.1.2 Identify test coverage items (TD2)

There are no recognized test coverage items for error guessing.

5.4.1.3 Derive test cases (TD3)

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

a) Select a defect type for coverage by the test case.

b) Identify test input values that can be expected to cause a failure corresponding to the selected defect type.

c) Determine the expected result of the test case by applying the test inputs to the test basis.

d) Repeat steps a) to c) until the required testing has been completed (normally this would be covering an agreed number of checklist entries).

6 Test coverage measurement

6.1 Overview

The coverage measures defined in this document 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/IEEE 29119-3). If a test coverage item is discounted, justification for its infeasibility will typically be recorded in a test report.

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

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

where

C	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 by the test design technique.

Specific values for C, N and T for each technique are defined in 6.2 and 6.3. The set of coverage measures presented in 6.2 and 6.3 are designed to be used with the test design techniques presented in this document. They are not designed to be an exhaustive list; there may be other useful coverage measures that are used by organizations that are not mentioned in this clause.

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

6.2 Test measurement for specification-based test design techniques

6.2.1 Equivalence partition coverage

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

— $C_{\text{equivalence\_partitions}}$ is the equivalence partition coverage;

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

— T is the total number of partitions identified.

6.2.2 Classification tree method coverage

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

For minimality coverage, the following definitions shall be used:

— $C_{\text{classification\_tree\_minimality}}$ is the minimality classification tree method coverage;

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

— T is the total number of classes.

For maximality combination coverage, the following definitions shall be used:

— $C_{\text{classification\_tree\_maximality}}$ is the maximality classification tree method coverage;

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

— T is the total number of unique combinations of classes.

6.2.3 Boundary value analysis coverage

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

— $C_{\text{boundary\_values}}$ 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.

6.2.4 Syntax testing coverage

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

NOTE Calculating coverage as a percentage is not possible for syntax testing due to the potentially extremely large number of options and infinite number of mutations possible.

6.2.5 Combinatorial test design techniques coverage

6.2.5.1 All combinations testing coverage

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

— $C_{\text{all\_combinations}}$ 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).

6.2.5.2 Pair-wise testing coverage

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

— $C_{pair\text{-}wise}$ 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 pairs of P-V pairs.

6.2.5.3 Each choice testing coverage

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

— $C_{\text{each\_choice}}$ 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.

6.2.5.4 Base choice testing coverage

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

— $C_{\text{base\_choice}}$ 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).

6.2.6 Decision table testing coverage

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

— $C_{\text{decision\_table}}$ 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.

6.2.7 Cause-effect graphing coverage

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

— $C_{\text{cause-effect}}$ is the cause-effect coverage;

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

— T is the total number of feasible decision rules.

6.2.8 State transition testing coverage

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

— $C_{\text{all\_states}}$ 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:

— $C_{\text{0-switch}}$ 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:

— $C_{\text{all\_transitions}}$ is all transitions 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:

— $C_{\text{N-switch}}$ 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.

6.2.9 Scenario testing coverage

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

— $C_{\text{scenarios}}$ is scenario coverage;

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

— T is the total number of scenarios.

6.2.10 Random testing coverage

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

6.2.11 Metamorphic testing coverage

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

NOTE Calculating coverage as a percentage is often not possible for metamorphic testing due to the potentially extremely large number of follow-up test cases that can be derived from a single metamorphic relation.

6.2.12 Requirements-based testing coverage

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

— $C_{\text{requirements}}$ is requirement coverage;

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

— T is the total number of atomic requirements.

6.3 Test measurement for structure-based test design techniques

6.3.1 Statement testing coverage

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

— $C_{\text{statement}}$ is statement coverage;

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

— T is the total number of executable statements.

6.3.2 Branch testing coverage

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

— C_{branch} is branch coverage;

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

— T is the total number of branches.

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

6.3.3 Decision testing coverage

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

— C_{decision} is decision coverage;

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

— T is the total number of decision outcomes.

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

6.3.4 Branch condition testing coverage

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

— $C_{\text{branch\_condition}}$ is branch condition coverage;

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

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

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

6.3.5 Branch condition combination testing coverage

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

— $C_{\text{branch\_condition\_combination}}$ 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.

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

6.3.6 Modified condition/decision coverage (MCDC)

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

— C_{MCDC} 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 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 decision outcome.

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

6.3.7 Data flow testing coverage

6.3.7.1 All-definitions testing coverage

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

— $C_{\text{all-definitions}}$ is all-definitions coverage;

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

— T is the total number of definition-use pairs from distinct variable definitions.

6.3.7.2 All-c-uses testing coverage

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

— $C_{\text{all-c-uses}}$ is all-c-uses coverage;

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

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

6.3.7.3 All-p-uses testing coverage

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

— $C_{\text{all-p-uses}}$ is all-p-uses coverage;

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

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

6.3.7.4 All-uses testing coverage

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

— $C_{\text{all-uses}}$ is all-uses coverage;

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

— T is the total number of definition-use pairs from each definition to all uses of that definition.

6.3.7.5 All-du-paths testing coverage

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

— $C_{\text{all-du-paths}}$ is all-du-paths coverage;

— N is the number of loop-free sub-paths from each variable definition to each of its uses that can be reached (without an intervening definition of the variable) covered by executed test cases;

— T is the total number of loop-free control flow sub-paths from each variable definition to every use (both p-use and c-use) of that definition (with no intervening definitions).

6.4 Test measurement for experience-based testing design techniques — Error guessing coverage

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

Annex A
(informative)

Testing quality characteristics

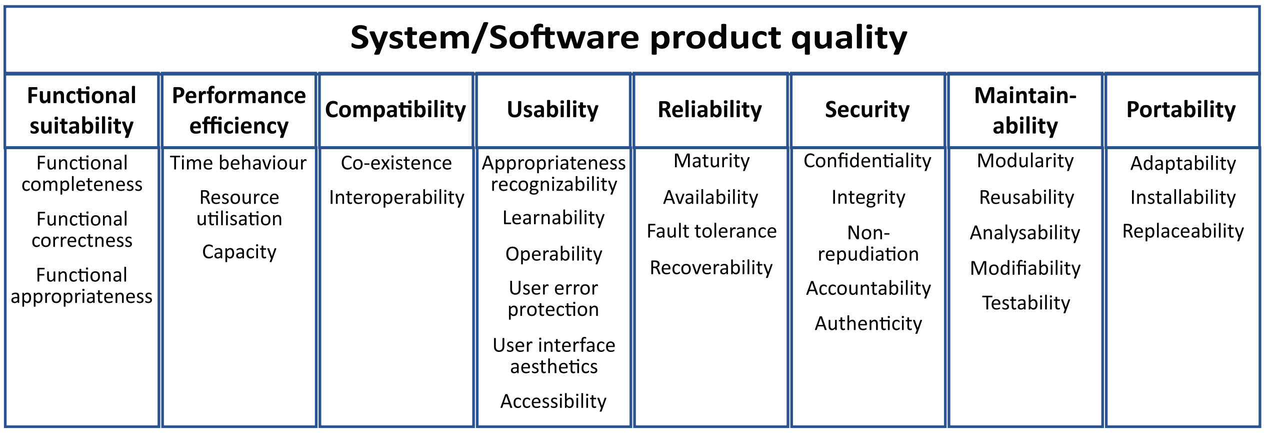
A.1 Quality characteristics

Software testing is carried out to collect evidence on how well required quality criteria have been satisfied by a test item. This annex contains examples of how quality characteristics defined in ISO/IEC 25010 can be mapped to the test design techniques defined in this document. The test design techniques defined in this document can be used for testing a variety of these characteristics. There may be other quality characteristics that can be considered (e.g. privacy).

ISO/IEC 25010 defines the product quality model shown below in Figure A.1, which categorizes 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 can 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 A.3 and A.4).

A.2 provides an explanation of types of testing that can be used to test the quality characteristics presented in Figure A.1. A mapping of each quality characteristic to applicable types of testing is presented in A.3. The relationship between the quality characteristics and the specification-based and structure-based test design techniques covered in this document is explained in A.4.

NOTE ISO/IEC 25030 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 and corresponding types of testing that apply to testing each quality requirement.



System/Software product quality							
Functional suitability	**Performance efficiency**	**Compatibility**	**Usability**	**Reliability**	**Security**	**Maintain-ability**	**Portability**
Functional completeness	Time behaviour	Co-existence	Appropriateness recognizability	Maturity	Confidentiality	Modularity	Adaptability
Functional correctness	Resource utilisation	Interoperability	Learnability	Availability	Integrity	Reusability	Installability
Functional appropriateness	Capacity		Operability	Fault tolerance	Non-repudiation	Analysability	Replaceability
			User error protection	Recoverability	Accountability	Modifiability	
			User interface aesthetics		Authenticity	Testability	
			Accessibility				

Figure A.1 — 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), 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 can 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 https://www.w3.org/standards/ for more information.

A.2.2 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.2.3 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. One common subtype of conversion testing is data migration testing. 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.2.4 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.

Failover/recovery testing is a form of disaster/recovery testing that is limited to moving to a back-up system in the event of failure, without transfer to a different operating site.

Back-up/recovery testing is a form of disaster/recovery testing that is limited to restoring from back-up memory in the event of failure, without transfer to a different operating site or back-up system.

A.2.5 Functional testing

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

A.2.6 Installability testing

The purpose of installability testing is to determine if a test item(s) can be installed, uninstalled/removed and/or upgraded 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, uninstallation or upgrade processes (as described in the installation manual or guidelines), the people who will carry out the installation, uninstallation or upgrade, the target platform(s) and the test item(s) to be installed, uninstalled or upgraded.

A.2.7 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 can include assessing whether a test item running in one environment can interact accurately with another test item or system in another separate environment.

A.2.8 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.2.9 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).

Maintainability can be indirectly measured by applying static analysis.

A.2.10 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 and reliability) 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, random access memory (RAM) and read only memory (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.2.11 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 can include assessing whether the test item can be operated from a variety of different browsers.

A.2.12 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;

— reconfigure the test item.

A.2.13 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 failures). 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.2.14 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 unauthorised persons or systems cannot use, read or modify them and authorised 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 can 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.2.15 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 9241-11 defines requirements for human-systems interaction.

A.3 Mapping quality characteristics to types of testing

In Table A.1, the types of testing presented in A.2 are mapped to the quality characteristics presented in Figure A.1.

Table A.1 — Mapping of ISO/IEC 25010 product quality characteristics to types of testing

Type of testing	Quality characteristic	Sub-characteristics
Accessibility testing	Usability	Accessibility
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
Localization testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Usability	Appropriateness recognizability
		Learnability
		Operability
		User error protection
		User interface aesthetics
		Accessibility
	Portability	Adaptability
Maintainability testing	Maintainability	Modularity
		Reusability
		Analysability
		Modifiability
		Testability
Performance-related testing	Performance efficiency	Time-behaviour
		Resource utilisation
		Capacity
Portability testing	Portability	Adaptability
		Installability
		Replaceability
Procedure testing	None	None
Reliability testing	Reliability	Maturity
		Availability
		Fault tolerance
		Recoverability

Table A.1 *(continued)*

Type of testing	Quality characteristic	Sub-characteristics
Security testing	Security	Confidentiality
		Integrity
		Non-repudiation
		Accountability
		Authenticity
Usability testing	Usability	Appropriateness recognizability
		Learnability
		Operability
		User error protection
		User interface aesthetics
		Accessibility

A.4 Mapping quality characteristics to test design techniques

The test design techniques described in this document can be used to test a variety of the quality characteristics listed in Figure A.1. Table A.2 provides an example mapping between them.

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

Test design technique	ISO/IEC 25010 quality characteristic	ISO/IEC 25010 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

Table A.2 *(continued)*

Test design technique	ISO/IEC 25010 quality characteristic	ISO/IEC 25010 sub-characteristics
Combinatorial test design techniques	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Compatibility	Co-existence
	Performance efficiency	Time-behaviour
	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

Table A.2 *(continued)*

Test design technique	ISO/IEC 25010 quality characteristic	ISO/IEC 25010 sub-characteristics
Metamorphic testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Performance	Time behaviour
		Resource utilisation
		Capacity
	Compatibility	Co-existence
		Interoperability
	Usability	Appropriateness recognizability
		Learnability
		Operability
		User error protection
		User interface aesthetics
		Accessibility
	Reliability	Maturity
		Availability
		Fault tolerance
		Recoverability
	Security	Confidentiality
		Integrity
		Non-repudiation
		Accountability
		Authenticity
	Maintainability	Modularity
		Reusability
		Analysability
		Modifiability
		Testability
	Portability	Adaptability
		Installability
		Replaceability

Table A.2 *(continued)*

Test design technique	ISO/IEC 25010 quality characteristic	ISO/IEC 25010 sub-characteristics
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

Table A.2 *(continued)*

Test design technique	ISO/IEC 25010 quality characteristic	ISO/IEC 25010 sub-characteristics
Requirements-based testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Performance	Time behaviour
		Resource utilisation
		Capacity
	Compatibility	Co-existence
		Interoperability
	Usability	Appropriateness recognizability
		Learnability
		Operability
		User error protection
		User interface aesthetics
		Accessibility
	Reliability	Maturity
		Availability
		Fault tolerance
		Recoverability
	Security	Confidentiality
		Integrity
		Non-repudiation
		Accountability
		Authenticity
	Maintainability	Modularity
		Reusability
		Analysability
		Modifiability
		Testability
	Portability	Adaptability
		Installability
		Replaceability
Scenario testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
	Usability	Learnability
		Operability
		User error protection
		User interface aesthetics
		Accessibility
		Appropriateness recognizability

Table A.2 *(continued)*

Test design technique	ISO/IEC 25010 quality characteristic	ISO/IEC 25010 sub-characteristics
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
		Operability
		User error protection
		User interface aesthetics
		Accessibility
		Appropriateness recognizability
Structure-based test design techniques		
Branch condition combination testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Branch condition testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Branch testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Data flow testing	Functional suitability	Functional completeness
		Functional correctness
		Functional appropriateness
Decision 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		

Table A.2 *(continued)*

Test design technique	ISO/IEC 25010 quality characteristic	ISO/IEC 25010 sub-characteristics
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 5.2 and 6.2 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/IEEE 29119-2. Although each example is applied in a specification-based testing context, as stated in 5.1, in practice most of the techniques defined in this document can be used interchangeably (e.g. boundary value analysis can 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 Test basis

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

The component receives an exam mark (out of 75) and a coursework (c/w) mark (out of 25) as input, from which it outputs a grade for the course in the range 'A' to 'D'. The grade is generated by calculating 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 any invalid inputs are detected (e.g. a mark is outside its expected range) then a fault message ('FM') is generated. All inputs are passed as integers.

B.2.1.2 Determine required test coverage

Full coverage of all equivalence partitions.

B.2.1.3 Create test model (TD1)

The diagram in Figure B.1 shows the equivalence partitions for the 'generate_grading' program as the grey-filled boxes with black outlines.

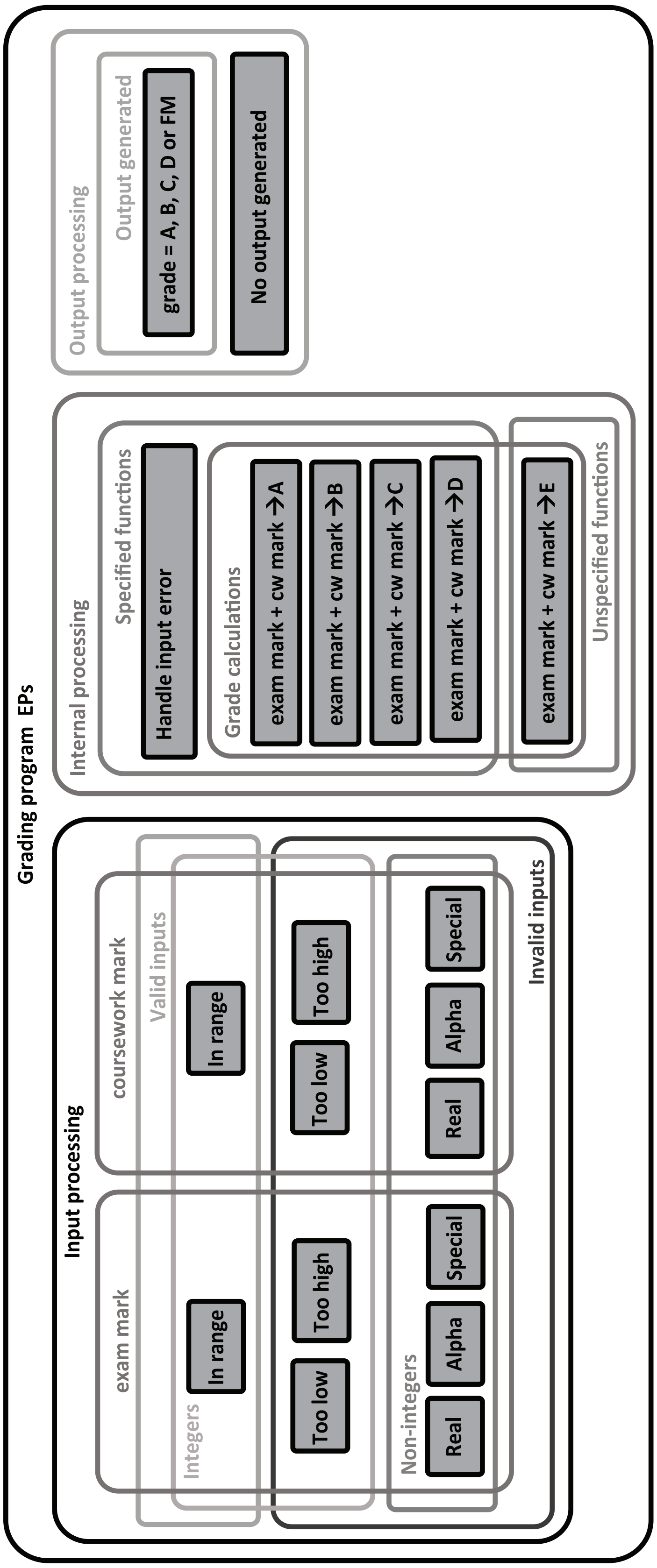


Figure B.1 — Equivalence partitions for the 'generate_grading' program

B.2.1.4 Identify test coverage items (TD2)

The test coverage items are the equivalence partitions shown in the test model:

From the input processing:

TESTCOVER1: $0 \leq$ exam mark ≤ 75

TESTCOVER2: $0 \leq$ coursework mark ≤ 25

TESTCOVER3: exam mark < 0

TESTCOVER4: exam mark > 75

TESTCOVER5: coursework mark < 0

TESTCOVER6: coursework mark > 25

TESTCOVER7: exam mark = real

TESTCOVER8: exam mark = alphabetic

TESTCOVER9: exam mark = special character (e.g. @, #, {, null)

TESTCOVER10: coursework mark = real

TESTCOVER11: coursework mark = alphabetic

TESTCOVER12: coursework mark = special character (e.g. @, #, {, null)

From the internal processing:

TESTCOVER13:	‘A’ is induced by	$70 \leq$ total mark ≤ 100
TESTCOVER14:	‘B’ is induced by	$50 \leq$ total mark < 70
TESTCOVER15:	‘C’ is induced by	$30 \leq$ total mark < 50
TESTCOVER16:	‘D’ is induced by	$0 \leq$ total mark < 30

TESTCOVER17: output = ‘E’

TESTCOVER18: ‘Fault Message’ (FM) is induced

NOTE It has been assumed that the generation of a grading of E is due to an extraneous unnecessary function that has been mistakenly included in the program.

From the output processing:

TESTCOVER19: output generated (A, B, C, D or FM)

TESTCOVER20: no output generated

B.2.1.5 Derive test cases (TD3)

B.2.1.5.1 General

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 (other approaches to selecting combinations of test coverage items to be exercised by test cases that may be used are described in Reference [11]). In the first a test case is generated for each identified partition on a one-to-one basis (see option a specified in B.2.1.5.2), while in the second a minimized set of test cases is generated to cover all the identified partitions (see option b specified in B.2.1.5.3). 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 mark and coursework mark.

B.2.1.5.2 Option a: derive test cases for one-to-one equivalence partitioning

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 test coverage items were identified leading to twenty test cases.

The test cases corresponding to partitions derived from the input exam mark are shown in Table B.1. Note that the input coursework mark in the following test case table has been set to an arbitrary valid value.

Table B.1 — Test cases for valid input processing of exam mark

Test case	1	2	3
Input (exam mark)	60	-10	93
Input (coursework mark)	15	15	15
total mark (as calculated)	75	-	-
Test coverage item	TESTCOVER1	TESTCOVER3	TESTCOVER4
Partition tested (*e* = exam mark)	$0 \leq e \leq 75$	$e < 0$	$e > 75$
Expected output	'A'	'FM'	'FM'

The test cases corresponding to partitions derived from the input coursework mark are shown in Table B.2.

Table B.2 — Test cases for valid input processing of coursework mark

Test case	4	5	6
Input (exam mark)	40	40	40
Input (coursework mark)	20	-15	47
total mark (as calculated)	60	-	-
Test coverage item	TESTCOVER2	TESTCOVER5	TESTCOVER6
Partition tested (*c* = coursework mark)	$0 \leq c \leq 25$	$c < 0$	$c > 25$
Expected output	'B'	'FM'	'FM'

The test cases corresponding to partitions derived from possible invalid inputs are shown in Table B.3 and Table B.4.

Table B.3 — Test cases for invalid input processing for exam mark

Test case	7	8	9
Input (exam mark)	60.5	Q	$
Input (coursework mark)	15	15	15
Test coverage item	TESTCOVER7	TESTCOVER8	TESTCOVER9
Partition tested (*e* = exam mark)	*e* = real number	*e* = alphabetic	*e* = special char
Expected output	'FM'	'FM'	'FM'

Table B.4 — Test cases for invalid input processing for coursework mark

Test case	10	11	12
Input (exam mark)	40	40	40
Input (coursework mark)	20.23	G	@

Table B.4 *(continued)*

Test case	10	11	12
Test coverage item	TESTCOVER10	TESTCOVER11	TESTCOVER12
Partition tested (c = coursework mark)	c = real number	c = alphabetic	c = special char
Expected output	'FM'	'FM'	'FM'

The test cases corresponding to partitions derived from the valid outputs are shown in Table B.5 and Table B.6.

Table B.5 — Test cases for grade processing

Test case	13	14	15
Input (exam mark)	60	44	32
Input (coursework mark)	20	22	13
total mark (as calculated)	80	66	45
Test coverage item	TESTCOVER13	TESTCOVER14	TESTCOVER15
Partition tested (t = total mark)	$70 \le t \le 100$	$50 \le t < 70$	$30 \le t < 50$
Expected output	'A'	'B'	'C'

Table B.6 — Test cases for grade processing

Test case	16	17	18
Input (exam mark)	12	5	80
Input (coursework mark)	5	5	60
total mark (as calculated)	17	10	-
Test coverage item	TESTCOVER16	TESTCOVER17	TESTCOVER18
Partition tested (t = total mark)	$0 \le t < 30$	‘E’	‘FM’ induced
Expected output	‘D’	‘D’	'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 shown in Table B.7.

Table B.7 — Test cases for invalid output processing

Test case	19	20
Input (exam mark)	47,3	Null
Input (coursework mark)	@@@	Null
Test coverage item	TESTCOVER19	TCESTOVER20
Partition tested	output generated	no output generated
Expected output	‘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 to 12, and 18 to 20 in the example above). For instance, in some programming languages, 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.2.1.5.3 Option b: derive test cases for minimized equivalence partitioning

It can be seen from B.2.1.5.2 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 can potentially "hit" three partitions; two input partitions and one output partition. Thus, it is possible to generate a smaller "minimized" 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 suite of 11 test cases shown in Tables B.8 to B.10 corresponds to the minimized equivalence partitioning approach where each test case is designed to hit as many new partitions as possible rather than just one. In the following test cases, those test coverage items that are newly ‘hit’ are underlined, and those that are already ‘hit’ are shown in brackets ( ).

Table B.8 — Minimized test cases

Test case	1	2	3	4
Input (exam mark)	60	50	35	19
Input (coursework mark)	20	16	10	8
Test coverage items	TESTCOVER1, TESTCOVER2, TESTCOVER13, TESTCOVER19	(TESTCOVER1), (TESTCOVER2), TESTCOVER14, (TESTCOVER19)	(TESTCOVER1), (TESTCOVER2), TESTCOVER15, (TESTCOVER19)	(TESTCOVER1), (TESTCOVER2), TESTCOVER16, (TESTCOVER19)
Partition (*e* = exam mark)	$0 \le e \le 75$	$0 \le e \le 75$	$0 \le e \le 75$	$0 \le e \le 75$
Partition (*c* = coursework mark)	$0 \le c \le 25$	$0 \le c \le 25$	$0 \le c \le 25$	$0 \le c \le 25$
Partition (*t* = total mark)	$70 \le t \le 100$	$50 \le t < 70$	$30 \le t < 50$	$0 \le t < 30$
Partition	output generated	output generated	output generated	output generated
Expected output	'A'	'B'	'C'	'D'

Table B.9 — Minimized test cases *(continued)*

Test case	5	6	7	8
Input (exam mark)	-10	93	60.5	Q
Input (coursework mark)	-15	47	20.23	G
Test coverage items	TESTCOVER3, TESTCOVER5, TESTCOVER18, (TESTCOVER19)	TESTCOVER4, TESTCOVER6, (TESTCOVER18), (TESTCOVER19)	TESTCOVER7, TESTCOVER10, (TESTCOVER18), (TESTCOVER19)	TESTCOVER8, TESTCOVER11, (TESTCOVER18), (TESTCOVER19)
Partition (*e* = exam mark)	$e < 0$	$e > 75$	*e* = real number	*e* = alphabetic
Partition (*c* = coursework mark)	$c < 0$	$c > 25$	*c* = real number	*c* = alphabetic
Partition	‘FM' induced	‘FM' induced	‘FM' induced	‘FM' induced
Partition tested	output generated	output generated	output generated	output generated
Expected output	'FM'	'FM'	'FM'	'FM'

Table B.10 — Minimized test cases *(continued)*

Test case	9	10	11
Input (exam mark)	$	5	null
Input (coursework mark)	@	5	null
total mark (as calculated)	-	10	-

Table B.10 *(continued)*

Test case	9	10	11
Test coverage items	TESTCOVER9, TESTCOVER12, (TESTCOVER18), (TESTCOVER19)	(TESTCOVER1), (TESTCOVER2), (TESTCOVER16), TESTCOVER17, (TESTCOVER19)	(TESTCOVER9), (TESTCOVER12), (TESTCOVER18), (TESTCOVER19), TESTCOVER 20
Partition (*e* = exam mark)	*e* = special char	0 ≤ *e* ≤ 75	0 ≤ *e* ≤ 75
Partition (*c* = coursework mark)	*c* = special char	0 ≤ *c* ≤ 25	0 ≤ *c* ≤ 25
Partition (*t* = total mark)	-	0 ≤ *t* ≤ 15	0 ≤ *t* ≤ 15
Partition	‘FM' induced	output = ‘E’	no output generated
Partition tested	output generated	output generated	output generated
Expected output	‘FM’	‘D’	‘D’

The one-to-one and minimized approaches represent two different approaches that can be used for deriving test cases for equivalence partitioning. One-to-one test cases are particularly useful for testing error conditions (i.e. when trying to force specific error messages to be output), for example, to reduce the possibility that one error condition halts processing and/or masks or blocks 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 the same time. 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.2.2 Classification tree method

B.2.2.1 Test basis

Consider the test basis for a test item travel_preference, which records the travel preferences of staff of an Australian organization 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, standard

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”. Staff do not have the option of choosing no meal, thus this option is not supported in the example.

B.2.2.2 Determine required test coverage

The required level of test coverage is for ‘minimality’, where each class has to be used in at least one test case.

B.2.2.3 Create test model (TD1)

A classification tree can now be developed from the test basis as shown in Figure B.2.

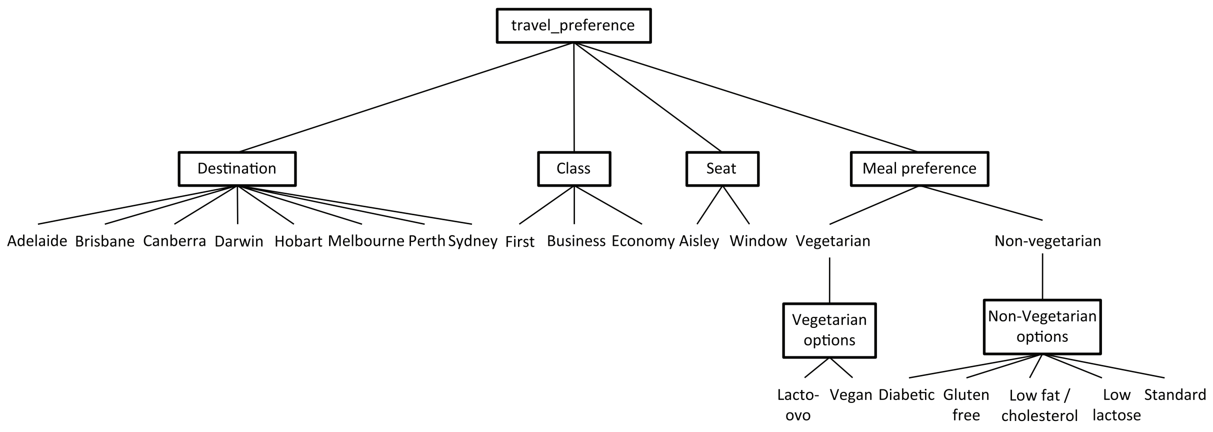


Figure B.2 — Classification tree for travel preferences

B.2.2.4 Identify test coverage items (TD2)

The test coverage items are the classes (leaf nodes) shown in the test model.

B.2.2.5 Derive test cases (TD3)

A combination table can be constructed under the classification tree to demonstrate which classes (i.e. leaf nodes) are combined to form each test case as shown in Figure B.3. The classes that are covered by each test case are marked by a series of tokens (black dots) that run horizontally underneath the classification tree.

As the required test coverage for this example is minimality, where each class has to be used in at least one test case, then the following eight combinations of classes can be used as the basis for the test cases, as shown in Figure B.3. 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".

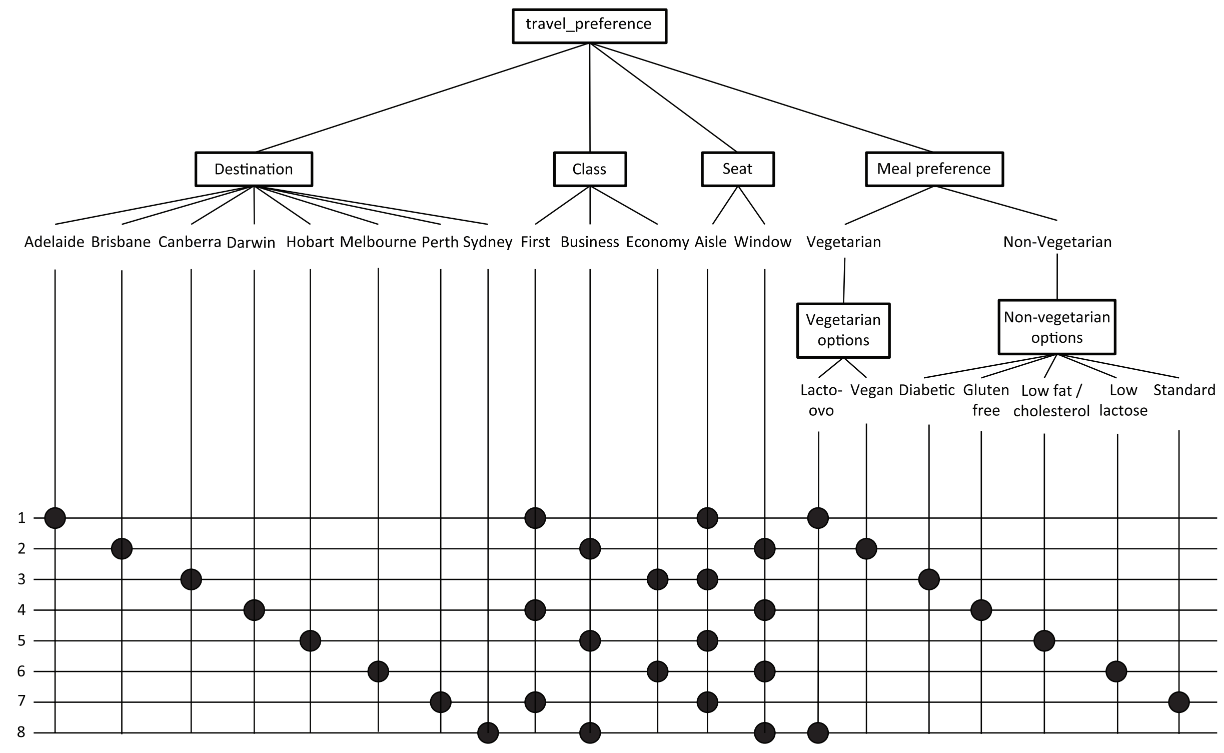


Figure B.3 — Example classification tree and corresponding combination table for travel preferences

The resultant set of test cases are shown in Table B.11.

Table B.11 — Test cases for classification tree testing

Test case	Input values				Expected result
	Destination	Class	Seat	Meal preference	
1	Adelaide	First	Aisle	Lacto-ovo	Booking accepted
2	Brisbane	Business	Window	Vegan	Booking accepted
3	Canberra	Economy	Aisle	Diabetic	Booking accepted
4	Darwin	First	Window	Gluten free	Booking accepted
5	Hobart	Business	Aisle	Low fat/choles-terol	Booking accepted
6	Melbourne	Economy	Window	Low lactose	Booking accepted
7	Perth	First	Aisle	Standard	Booking accepted
8	Sydney	Business	Window	Lacto-ovo	Booking accepted

NOTE If the required test coverage had been to meet the maximality criterion, then all combinations of classes would need to be exercised, which would result in 8 × 3 × 2 × 7 = 336 test cases.

B.2.2.6 Classification tree method coverage

Using the definitions provided in 6.2.2 and the test cases derived above:

$$C_{\text{classification\_tree\_minimality}} = \frac{20}{20} \times 100\,\% = 100\,\%$$

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

B.2.3 Boundary value analysis

B.2.3.1 General

The aim of boundary value analysis is to derive a set of test cases that cover the 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; second, that the members of some partitions can be ordered from lowest to highest with no discontinuity; and third, 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 of two-value boundary testing with one-to-one test cases (see 5.2.3.2 and 5.2.3.3 respectively).

B.2.3.2 Test basis

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

The component receives an exam mark (out of 75) and a coursework (c/w) mark (out of 25) as input, from which it outputs a grade for the course in the range 'A' to 'D'. The grade is generated by calculating 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 invalid input(s) are detected (e.g. a mark is outside its expected range) then a fault message ('FM') is generated. All inputs are passed as integers.

B.2.3.3 Determine required test coverage

Full coverage of all boundary values (using two-value boundary testing).

B.2.3.4 Create test model (TD1)

For boundary value analysis, the test model consists of the equivalence partitions for the test item that define ordered sets. The partitions are defined in terms of their boundaries.

Equivalence partitions are identified from the valid and invalid inputs and outputs of the test item.

The following valid equivalence partitions (EP) can be identified for the inputs:

EP1:	$0 \leq$ exam mark ≤ 75
EP2:	$0 \leq$ coursework mark ≤ 25

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

EP3:	exam mark > 75
EP4:	exam mark < 0
EP5:	coursework mark > 25

EP6:	coursework mark < 0

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 (remembering that all marks are integers):

EP3:	76 ≤ exam mark ≤ 32767 (or 75 < exam mark ≤ 32767)
EP4:	-32768 ≤ exam mark ≤ -1 (or -32768 ≤ exam mark < 0)
EP5:	26 ≤ coursework mark ≤ 32767 (or 25 < coursework mark ≤ 32767)
EP6:	-32768 ≤ coursework mark ≤ -1 (or -32768 ≤ coursework mark < 0)

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:

EP7:	'A' is induced by		70 ≤ total mark ≤ 100
EP8:	'B' is induced by		50 ≤ total mark ≤ 69
EP9:	'C' is induced by		30 ≤ total mark ≤ 49
EP10:	'D' is induced by		0 ≤ total mark ≤ 29
EP11:	'Fault Message' (FM)	is induced by	total mark > 100
EP12:	'Fault Message' (FM)	is induced by	total mark < 0

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, EP11 and EP12 can be redefined as follows.

EP11:	101 ≤ total mark ≤ 32767 (or 100 < total mark ≤ 32767)
EP12:	-32768 ≤ total mark ≤ -1 (or -32768 ≤ total mark < 0)

'Fault Message' is considered here as it is a specified output.

Note that the above 12 equivalence classes all define ordered sets (as is required for boundary value analysis). Further equivalence classes can be identified for unordered sets, but these are not required for boundary value analysis (the full set of equivalence partitions for this example test basis can be seen in B.2.1).

B.2.3.5 Identify test coverage items (TD2)

If 2-value boundary value analysis is applied, the test coverage items are:

— the boundary values (minimum and maximum values) of each equivalence partition;

— the values an incremental distance outside the boundary of each equivalence partition (as the marks are integers, the incremental distance is one).

BOUND1	exam mark = 0	from	EP1:	0 <= exam mark <= 75
BOUND1-out	exam mark = -1	from	EP1:	0 ≤ exam mark ≤ 75

BOUND2	exam mark = 75	from	EP1:	0 ≤ exam mark ≤ 75
BOUND2-out	exam mark = 76	from	EP1:	0 ≤ exam mark ≤ 75
BOUND3	coursework mark = 0	from	EP2:	0 ≤ coursework mark ≤ 25
BOUND3-out	coursework mark = -1	from	EP2:	0 ≤ coursework mark ≤ 25
BOUND4	coursework mark = 25	from	EP2:	0 ≤ coursework mark ≤ 25
BOUND4-out	coursework mark = 26	from	EP2:	0 ≤ coursework mark ≤ 25
BOUND5	exam mark = 76	from	EP3	76 ≤ exam mark ≤ 32767
BOUND5-out	exam mark = 75	from	EP3	76 ≤ exam mark ≤ 32767
BOUND6	exam mark = 32767	from	EP3	76 ≤ exam mark ≤ 32767
BOUND6-out	exam mark = 32768	from	EP3	76 ≤ exam mark ≤ 32767
BOUND7	exam mark = -32768	from	EP4:	-32768 ≤ exam mark ≤ -1
BOUND7-out	exam mark = -32769	from	EP4:	-32768 ≤ exam mark ≤ -1
BOUND8	exam mark = -1	from	EP4:	-32768 ≤ exam mark ≤ -1
BOUND8-out	exam mark = 0	from	EP4:	-32768 ≤ exam mark ≤ -1
BOUND9	coursework mark = 26	from	EP5:	26 ≤ coursework mark ≤ 32767
BOUND9-out	coursework mark = 25	from	EP5:	26 ≤ coursework mark ≤ 32767
BOUND10	coursework mark = 32767	from	EP5:	26 ≤ coursework mark ≤ 32767
BOUND10-out	coursework mark = 32768	from	EP5:	26 ≤ coursework mark ≤ 32767
BOUND11	coursework mark = -32768	from	EP6:	-32768 ≤ coursework mark ≤ -1
BOUND11-out	coursework mark = -32769	from	EP6:	-32768 ≤ coursework mark ≤ -1
BOUND12	coursework mark = -1	from	EP6:	-32768 ≤ coursework mark ≤ -1
BOUND12-out	coursework mark = 0	from	EP6:	-32768 ≤ coursework mark ≤ -1
BOUND13	total mark = 70	from	EP7:	70 ≤ total mark ≤ 100
BOUND13-out	total mark = 69	from	EP7:	70 ≤ total mark ≤ 100
BOUND14	total mark = 100	from	EP7:	70 ≤ total mark ≤ 100
BOUND14-out	total mark = 101	from	EP7:	70 ≤ total mark ≤ 100
BOUND15	total mark = 50	from	EP8:	50 ≤ total mark ≤ 69
BOUND15-out	total mark = 49	from	EP8:	50 ≤ total mark ≤ 69
BOUND16	total mark = 69	from	EP8:	50 ≤ total mark ≤ 69
BOUND16-out	total mark = 70	from	EP8:	50 ≤ total mark ≤ 69
BOUND17	total mark = 30	from	EP9:	30 ≤ total mark ≤ 49

BOUND17-out	total mark = 29	from	EP9:	30 ≤ total mark ≤ 49
BOUND18	total mark = 49	from	EP9:	30 ≤ total mark ≤ 49
BOUND18-out	total mark = 50	from	EP9:	30 ≤ total mark ≤ 49
BOUND19	total mark = 0	from	EP10:	0 ≤ total mark ≤ 29
BOUND19-out	total mark = -1	from	EP10:	0 ≤ total mark ≤ 29
BOUND20	total mark = 29	from	EP10:	0 ≤ total mark ≤ 29
BOUND20-out	total mark = 30	from	EP10:	0 ≤ total mark ≤ 29
BOUND21	total mark = 101	from	EP11:	101 ≤ total mark ≤ 32767
BOUND21-out	total mark = 100	from	EP11:	101 ≤ total mark ≤ 32767
BOUND22	total mark = 32767	from	EP11:	101 ≤ total mark ≤ 32767
BOUND22-out	total mark = 32768	from	EP11:	101 ≤ total mark ≤ 32767
BOUND23	total mark = -32768	from	EP12:	-32768 ≤ total mark ≤ -1
BOUND23-out	total mark = -32769	from	EP12:	-32768 ≤ total mark ≤ -1
BOUND24	total mark = -1	from	EP12:	-32768 ≤ total mark ≤ -1
BOUND24-out	total mark = 0	from	EP12:	-32768 ≤ total mark ≤ -1

Because several of the equivalence partitions are contiguous (i.e. they share a common boundary), the second value identified as being an incremental distance outside the boundary of the partition is often a duplicate of the boundary value of the next partition. For instance, BOUND1-out is a duplicate of BOUND8 as EP1 and EP4 share a common boundary.

When the duplication is removed, the following test coverage items are left:

TCOVER1	exam mark = 0	BOUND1 and BOUND8-out
TCOVER2	exam mark = 75	BOUND2 and BOUND5-out
TCOVER3	coursework mark = 0	BOUND3 and BOUND12-out
TCOVER4	coursework mark = 25	BOUND4 and BOUND9-out
TCOVER5	exam mark = 76	BOUND5 and BOUND2-out
TCOVER6	exam mark = 32767	BOUND6
TCOVER7	exam mark = 32768	BOUND6-out
TCOVER8	exam mark = -32768	BOUND7
TCOVER9	exam mark = -32769	BOUND7-out
TCOVER10	exam mark = -1	BOUND8 and BOUND1-out
TCOVER11	coursework mark = 26	BOUND9 and BOUND4-out
TCOVER12	coursework mark = 32767	BOUND10

TCOVER13	coursework mark = 32768	BOUND10-out
TCOVER14	coursework mark = -32768	BOUND11
TCOVER15	coursework mark = -32769	BOUND11-out
TCOVER16	coursework mark = -1	BOUND12 and BOUND3-out
TCOVER17	total mark = 70	BOUND13 and BOUND16-out
TCOVER18	total mark = 100	BOUND14 and BOUND21-out
TCOVER19	total mark = 50	BOUND15 and BOUND18-out
TCOVER20	total mark = 69	BOUND16 and BOUND13-out
TCOVER21	total mark = 30	BOUND17 and BOUND20-out
TCOVER22	total mark = 49	BOUND18 and BOUND15-out
TCOVER23	total mark = 0	BOUND19 and BOUND24-out
TCOVER24	total mark = 29	BOUND20 and BOUND17-out
TCOVER25	total mark = 101	BOUND21 and BOUND14-out
TCOVER26	total mark = 32767	BOUND22
TCOVER27	total mark = 32768	BOUND22-out
TCOVER28	total mark = -32768	BOUND23
TCOVER29	total mark = -32769	BOUND23-out
TCOVER30	total mark = -1	BOUND24 and BOUND19-out

NOTE Alternatively, 3-value boundary value analysis can be performed, which would result in a larger number of test coverage items being derived (one on the boundary and values either side of it).

B.2.3.6 Derive test cases (TD3)

Test cases can now be derived to cover the test coverage items that were identified in the previous step. One-to-one boundary value analysis can be used to derive one test case per test coverage item; or minimized boundary value analysis can be used to derive the minimum number of test cases required to cover all test coverage items. This example uses one-to-one boundary value analysis.

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

As 100 % boundary coverage is required and one-to-one boundary value analysis is used, then 8 test cases can be derived for the input exam mark, as shown in Table B.12. Each test case is derived as follows: first, select one boundary value (test coverage item) for inclusion in each test case; second, allocate an arbitrary valid value to the other input present in the test case (e.g. coursework mark = 15); and third, determine the expected output of the test.

Table B.12 — Test cases for exam mark

Test case	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Input (exam mark)	-1	0	32767	-32768	75	76	32768	-32769

Table B.12 *(continued)*

Test case	1	2	3	4	5	6	7	8
Input (coursework mark)	15	15	15	15	15	15	-	-
total mark (as calculated)	14	15	-	-32753	90	91	-	-
Test coverage item	TCOVER10	TCOVER1	TCOVER6	TCOVER8	TCOVER2	TCOVER5	TCOVER7	TCOVER9
Expected output	'FM'	'D'	'FM'	'FM'	'A'	'FM'	-	-

The exam mark inputs for TCOVER7 and TCOVER9 cannot be represented using 16 bits and so test cases 7 and 8 are considered to be infeasible.

Using the same approach, and setting the input exam mark to an arbitrary valid value of 40, the test cases derived for the input coursework mark are shown in Table B.13.

Table B.13 — Test cases for coursework mark

Test case	9	10	11	12	13	14	15	16
Input (exam mark)	40	40	40	40	40	40	-	-
Input (coursework mark)	-1	0	25	26	32767	-32768	32768	-32769
total mark (as calculated)	39	40	65	66	-	-32728	-	-
Test coverage item	TCOVER16	TCOVER3	TCOVER4	TCOVER11	TCOVER12	TCOVER14	TCOVER13	TCOVER15
Expected output	'FM'	'C'	'B'	'FM'	'FM'	‘FM’	-	-

The coursework mark inputs for TCOVER13 and TCOVER15 cannot be represented using 16 bits and so test cases 15 and 16 are considered to be infeasible.

Using the same approach, and given that the input values of exam mark and coursework mark have been derived from total mark (which is their sum), the test cases derived for the total mark are shown in Table B.14 and Table B.15.

Table B.14 — Test cases for total mark

Test case	17	18	19	20	21	22	23
Input (exam mark)	-1	0	24	50	29	15	49
Input (coursework mark)	0	0	25	0	0	15	20
total mark (as calculated)	-1	0	49	50	29	30	69
Test coverage item	TCOVER 30	TCOVER 23	TCOVER 22	TCOVER 19	TCOVER 24	TCOVER 21	TCOVER 20
Expected output	'FM'	'D'	'C'	'B'	'D'	'FM'	'B'

Table B.15 — Test cases for total mark *(continued)*

Test case	24	25	26	27	28	29	30
Input (exam mark)	45	75	75	32767	-16384	-	-
Input (coursework mark)	25	25	26	0	-16384	-	-
total mark (as calculated)	70	100	101	32767	-32768	32768	-32769

Table B.15 *(continued)*

Test case	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Test coverage item	TCOVER 17	TCOVER 18	TCOVER 25	TCOVER 26	TCOVER 28	TCOVER 27	TCOVER 29
Expected output	'A'	'A'	'FM'	'FM'	'FM'	-	-

The total mark values for TCOVER27 and TCOVER29 cannot be represented using 16 bits and so test cases 29 and 30 are considered to be infeasible.

It should be noted that when invalid input values are used that can be represented in 16 bits (as above, in test cases 1, 3, 4, 6, 9, 12, 13, 14, 17, 22, 26, 27 and 28) it may, depending on the implementation, be impossible to actually execute the test case. For instance, in the Ada programming language, 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 suite achieves 100 % boundary value coverage for 2-value boundary value analysis as it enables all feasible identified test coverage items to be exercised by at least one test case. Lower levels of coverage would be achieved if some of the feasible identified boundaries were not exercised.

B.2.3.7 Boundary Value Analysis Coverage

Using the definitions provided in 6.2.3 and the feasible test cases identified above:

$$C_{\text{boundary\_values}}=\frac{24}{24}\times 100\,\%=100\,\%$$

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

B.2.4 Syntax testing

B.2.4.1 General

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. The technique is illustrated 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 Test basis

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 Determine required test coverage

The required level of test coverage is to cover each of the valid options and each of the generic mutations using the following checklist:

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 Other mutations can be used, depending on which types of defects are the focus of testing.

B.2.4.4 Create test model (TD1)

The first step is to derive the test model from the syntax with the input parameters in the syntax forming the model, as follows:

INPUT1	float = int "e" int
INPUT2	int = ["+"|"-"] nat
INPUT3	nat = {dig}
INPUT4	dig = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

B.2.4.5 Identify test coverage items (TD2)

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 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 INPUT2:

TCOVER1:	there is no "+" or "-" sign	(for INPUT2 & INPUT1)
TCOVER2:	there is a "+" sign	(for INPUT2 & INPUT1)
TCOVER3:	there is a "-" sign	(for INPUT2 & INPUT1)

NOTE 1 Separate test coverage items can 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 INPUT3 & INPUT2)
TCOVER5:	nat is a multiple digit number	(for INPUT3 & INPUT2)

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

`dig` has ten options:

TCOVER6:	integer is a "0"	(for INPUT4 & INPUT3)
TCOVER7:	integer is a "1"	(for INPUT4 & INPUT3)
TCOVER8:	integer is a "2"	(for INPUT4 & INPUT3)
TCOVER9:	integer is a "3"	(for INPUT4 & INPUT3)
TCOVER10:	integer is a "4"	(for INPUT4 & INPUT3)

TCOVER11:	integer is a “5”	(for INPUT4 & INPUT3)
TCOVER12:	integer is a “6”	(for INPUT4 & INPUT3)
TCOVER13:	integer is a “7”	(for INPUT4 & INPUT3)
TCOVER14:	integer is an “8”	(for INPUT4 & INPUT3)
TCOVER15:	integer is a “9”	(for INPUT4 & INPUT3)

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

The invalid test coverage items are identified by using the given checklist of generic mutations (see B.2.4.3). These generic mutations are applied to the individual elements of the syntax to yield specific mutations:

TCOVER16	apply m1 to first “`int`”	(for INPUT1)
TCOVER17:	apply m1 to “`e`”	(for INPUT1)
TCOVER18:	apply m1 to second “`int`”	(for INPUT1)
TCOVER19:	apply m1 to “`["+"|"-"]`”	(for INPUT2)
TCOVER20:	apply m1 to “`nat`”	(for INPUT2)
TCOVER21:	apply m2 to substitute “`e`” for first “`int`”	(for INPUT1)
TCOVER22:	apply m2 to substitute “`["+"|"-"]`” first “`int`”	(for INPUT1 & INPUT2)
TCOVER23:	apply m2 to substitute first “`int`” for “`e`”	(for INPUT1)
TCOVER24:	apply m2 to substitute “`["+"|"-"]`” for “`e`”	(for INPUT1 & INPUT2)
TCOVER25:	apply m2 to substitute “`e`” for second “`int`”	(for INPUT1)
TCOVER26:	apply m2 to substitute “`["+"|"-"]`” for second “`int`”	(for INPUT1)
TCOVER27:	apply m2 to substitute “`e`” for “`["+"|"-"]`”	(for INPUT1 & INPUT2)
TCOVER28:	apply m2 to substitute “`e`” for “`nat`”	(for INPUT1 & INPUT2)
TCOVER29:	apply m2 to substitute “`["+"|"-"]`” for “`nat`”	(for INPUT1 & INPUT2)
TCOVER30:	apply m3 to first “`int`”	(for INPUT1)
TCOVER31:	apply m3 to “`e`”	(for INPUT1)
TCOVER32:	apply m3 to second “`int`”	(for INPUT1)
TCOVER33:	apply m4 to add element before first “`int`”	(for INPUT1)
TCOVER34:	apply m4 to add element before “`e`”	(for INPUT1)
TCOVER35:	apply m4 to add element before second “`int`”	(for INPUT1)
TCOVER36:	apply m4 to add element after second “`int`”	(for INPUT1)
TCOVER37:	apply m4 to add element before first “`int`” and “`["+"|"-"]`”	(for INPUT1 & INPUT2)
TCOVER38:	apply m4 to add element between “["+"|"-"]” and first “int”	(for INPUT1 & INPUT2)

TCOVER39: apply m4 to add element between first “int” and “e” (for INPUT1)

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

B.2.4.6 Derive test cases (TD3)

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 shown in Table B.20.

Table B.20 — Valid test cases for syntax testing

Test case	Input ‘float_in’	Test coverage item	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 “Test coverage item” 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 shown in Table B.21.

Table B.21 — Invalid test cases for syntax testing

Test case	Input ‘float_in’	Mutation	Test coverage item	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’

Table B.21 *(continued)*

Test case	Input 'float_in'	Mutation	Test coverage item	Expected result 'check_res'
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 INPUT2 for INPUT4) generates correct syntax as m2 is "substitute an element with another defined element" and INPUT2 and INPUT4 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 INPUT4, which should be an integer, with "+" creates the form "0e+". This is the same input as generated for test case 26 above.

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

B.2.4.7 Syntax testing coverage

As stated in 6.2.4, there is no approach for calculating test coverage item coverage for syntax testing.

B.2.5 Combinatorial test design techniques

B.2.5.1 General

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 creating a test model, 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 Test basis

Consider the test basis for a test item travel_preference, which records the travel preferences of staff members of an organization 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 Determine required test coverage

The following levels of test coverage will be achieved for combinatorial testing of this example:

— all combinations (see B.2.5.5);

— pair-wise testing (see B.2.5.6);

— each choice testing (see B.2.5.7);

— base choice testing (see B.2.5.8).

B.2.5.4 Create test model (TD1)

Every combinatorial technique shares a common approach to creating the test model. The test model is made up of each parameter (P) of the test item taking on a 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:

PV1:	Destination – Paris
PV2:	Destination – London
PV3:	Destination – Sydney
PV4:	Class – First
PV5:	Class – Business
PV6:	Class – Economy
PV7:	Seat – Aisle
PV8:	Seat – Window

B.2.5.5 All combinations

B.2.5.5.1 Identify test coverage items (TD2)

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 PV 1, 4, 7)
TCOVER2:	Destination – Paris,	Class – First,	Seat – Window	(for PV 1, 4, 8)
TCOVER3:	Destination – Paris,	Class – Business,	Seat – Aisle	(for PV 1, 5, 7)

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

B.2.5.5.2 Derive test cases (TD3)

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 test cases shown in Table B.22.

Table B.22 — Test cases for all combinations testing

Test case#	Input values			Expected result	Test coverage item
	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

Table B.22 *(continued)*

Test case#	Input values			Expected result	Test coverage item
	Destination	Class	Seat		
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 All combinations testing coverage

Using the definitions provided in 6.2.5.1 and the test coverage items derived above:

$$C_{\text{all\_combinations}} = \frac{18}{18} \times 100\,\% = 100\,\%$$

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 Identify test coverage items (TD2)

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 PV1, PV4)
TCOVER2:	Paris, Business	(for PV1, PV5)
TCOVER3:	Paris, Economy	(for PV1, PV6)
TCOVER4:	London, First	(for PV2, PV4)
TCOVER5:	London, Business	(for PV2, PV5)
TCOVER6:	London, Economy	(for PV2, PV6)
TCOVER7:	Sydney, First	(for PV3, PV4)
TCOVER8:	Sydney, Business	(for PV3, PV5)
TCOVER9:	Sydney, Economy	(for PV3, PV6)
TCOVER10:	Paris, Aisle	(for PV1, PV7)
TCOVER11:	Paris, Window	(for PV1, PV8)
TCOVER12:	London, Aisle	(for PV2, PV7)
TCOVER13:	London, Window	(for PV2, PV8)
TCOVER14:	Sydney, Aisle	(for PV3, PV7)
TCOVER15:	Sydney, Window	(for PV3, PV8)
TCOVER16:	First, Aisle	(for PV4, PV7)
TCOVER17:	First, Window	(for PV4, PV8)

TCOVER18:	Business, Aisle	(for PV5, PV7)
TCOVER19:	Business, Window	(for PV5, PV8)
TCOVER20:	Economy, Aisle	(for PV6, PV7)
TCOVER21:	Economy, Window	(for PV6, PV8)

B.2.5.6.2 Derive test cases (TD3)

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. See Table B.23.

Table B.23 — Test cases pair-wise testing

Test case #	Input values			Expected result	Test coverage items
	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 Pair-wise testing coverage

Using the definitions provided in 6.2.5.2 and the test coverage items derived above:

$$C_{\text{pair-wise}} = \frac{21}{21} \times 100\% = 100\%$$

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 Identify test coverage items (TD2)

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 PV1)
TCOVER2:	Destination – London	(for PV2)
TCOVER3:	Destination – Sydney	(for PV3)
TCOVER4:	Class – First	(for PV4)
TCOVER5:	Class – Business	(for PV5)

TCOVER6:	Class – Economy	(for PV6)
TCOVER7:	Seat – Aisle	(for PV7)
TCOVER8:	Seat – Window	(for PV8)

B.2.5.7.2 Derive test cases (TD3)

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; see Table B.24.

Table B.24 — Test cases for each choice testing

Test case #	Input values			Expected result	Test coverage items
	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 can be derived that would also achieve the required level of coverage.

B.2.5.7.3 Each choice testing coverage

Using the definitions provided in 6.2.5.3 and the test coverage items derived above:

$$C_{\text{each\_choice}} = \frac{8}{8} \times 100\,\% = 100\,\%$$

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 Identify test coverage items (TD2)

Test coverage items for base choice testing are chosen by selecting a “base choice” value for each parameter. For example, the base choice can 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	(PV2, PV6 & PV8)

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

TCOVER2	Destination – Paris,	Class – Economy,	Seat – Window	(PV1, PV6 & PV8)
TCOVER3	Destination – Sydney,	Class – Economy,	Seat – Window	(PV3, PV6 & PV8)
TCOVER4	Destination – London,	Class – First,	Seat – Window	(PV2, PV4 & PV8)
TCOVER5	Destination – London,	Class – Business,	Seat – Window	(PV2, PV5 & PV8)
TCOVER6	Destination – London,	Class – Economy,	Seat – Aisle	(PV2, PV6 & PV7)

B.2.5.8.2 Derive test cases (TD3)

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; see Table B.25.

Table B.25 — Test cases for base choice testing

Test case #	Input values			Expected result	Test coverage item
	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 Base choice testing coverage

Using the definitions provided in 6.2.5.4 and the test coverage items derived above:

$$C_{\text{base\_choice}} = \frac{6}{6} \times 100\,\% = 100\,\%$$

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

B.2.6 Decision table testing

B.2.6.1 General

The aim of decision table testing is to derive a set of test cases that cover the logical associations between inputs and outputs (which are 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 Test basis

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 Determine required test coverage

Full coverage of all (feasible) rules in the decision table.

B.2.6.4 Create test model (TD1)

The test model consists of the decision table that can be derived from the test basis.

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 decision table shown as Table B.26, which identifies 8 decision rules, 6 of which are feasible and hence results in the definition of 6 test coverage items.

Table B.26 — Decision table of the cheque debit function

	Decision rules:	1	2	3	4	5	6	7	8
Conditions	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
Actions	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 Table B.26 to denote “True” and “False”, other notations can be used (e.g. the words “true” and “false” can be used instead).

NOTE 2 In Table B.26, 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.2.6.5 Identify test coverage items (TD2)

The test coverage items are the six feasible decision rules in the decision table (rules 1 to 6 in Table B.26).

B.2.6.6 Derive test cases (TD3)

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 test cases shown in Table B.27 would be required to achieve 100 % decision table coverage, and correspond to the decision rules in Table B.26 (no test cases are derived for decision rules 7 and 8 since they are infeasible):

Table B.27 — Test case table of the cheque debit function

Test case	CONDITIONS/INPUTS				ACTIONS/RESULTS		Test coverage item
	Account type	Overdraft limit	Current balance	Debit amount	New balance	Action code	
1	'c'	£100	-£70	£50	-£70	'L'	1

Table B.27 *(continued)*

Test case	CONDITIONS/INPUTS				ACTIONS/RESULTS		Test coverage item
	Account type	Overdraft limit	Current balance	Debit amount	New balance	Action code	
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 Decision table testing coverage

Using the definitions provided in 6.2.6 and the test coverage items derived above:

$$C_{\text{decision\_table}} = \frac{6}{6} \times 100\,\% = 100\,\%$$

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

B.2.7 Cause-effect graphing

B.2.7.1 General

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.

B.2.7.2 Test basis

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 Determine required test coverage

Full coverage of all (feasible) cause-effect relationships shown in the cause-effect graph.

NOTE This is equivalent to coverage of all (feasible) rules in the corresponding decision table.

B.2.7.4 Create test model (TD1)

The test model is the cause-effect graph.

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 in Figure B.4.

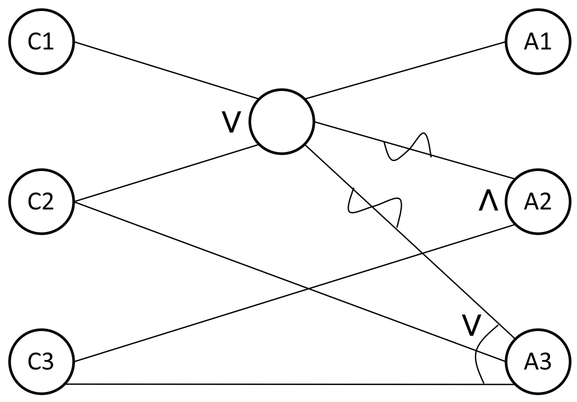
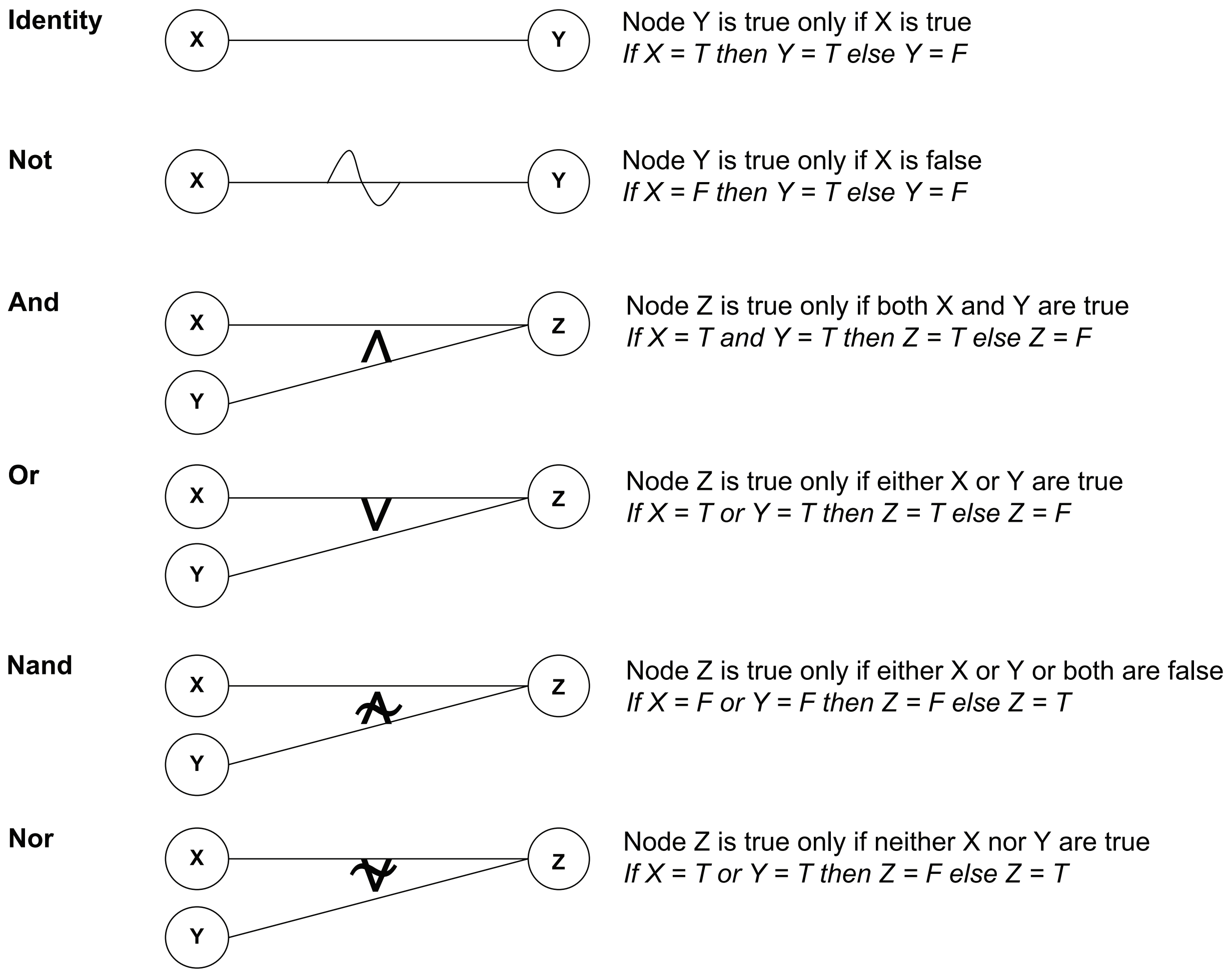


Figure B.4 — Cause-effect graph of the cheque debit function (see Figure B.5 for notation)

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.



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

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

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

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

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

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 B.5 — Notation for illustrating relationships between causes and effects in cause-effect graphing

NOTE 2 Although the "constraint" notations shown in Figure B.6 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 a means for verifying the integrity of the cause-effect graph, the decision table and the test cases that are derived from it.

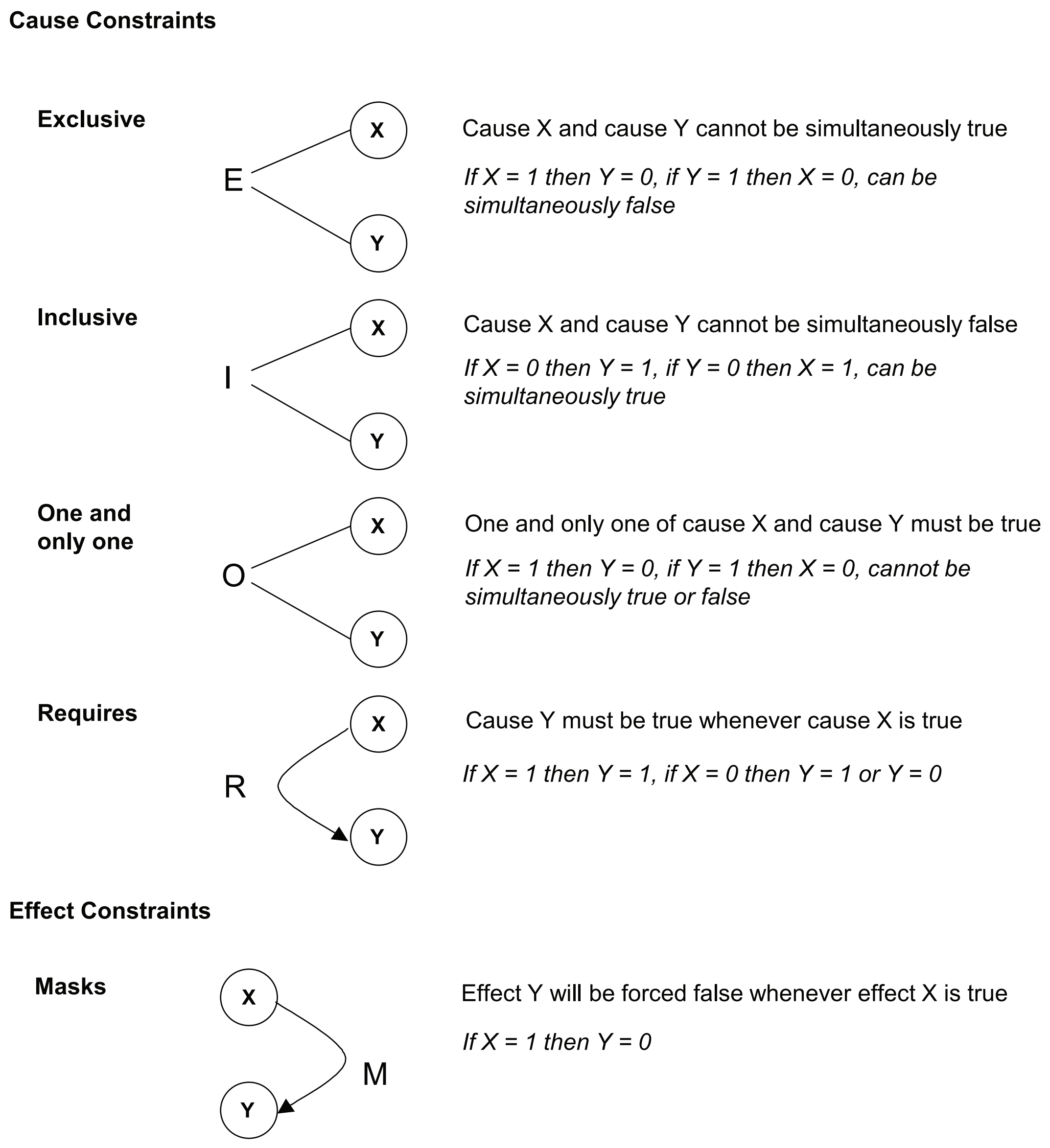

Cause Constraints
Exclusive
E
X
Y
Cause X and cause Y cannot be simultaneously true
If X = 1 then Y = 0, if Y = 1 then X = 0, can be simultaneously false
Inclusive
I
X
Y
Cause X and cause Y cannot be simultaneously false
If X = 0 then Y = 1, if Y = 0 then X = 1, can be simultaneously true
One and only one
O
X
Y
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 simultaneously true or false
Requires
R
X
Y
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
M
X
Y
Effect Y will be forced false whenever effect X is true
If X = 1 then Y = 0

Figure B.6 — Notation for representing cause and effect constraints in cause-effect graphing

B.2.7.5 Identify test coverage items (TD2)

The cause-effect graph is next recast in terms of a decision table (e.g. see References [19] and [20]), 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 decision table shown as Table B.28, which identifies six test coverage items (decision rules 7 and 8 are not test coverage items since they are infeasible).

Table B.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

Table B.28 *(continued)*

Decision rules:	1	2	3	4	5	6	7	8
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 Derive test cases (TD3)

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 test cases shown in Table B.29 achieve 100 % cause-effect coverage and correspond to the decision rules in Table B.28 (no test cases are generated for decision rules 7 and 8 as they are infeasible).

Table B.29 — 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
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 Cause-effect graphing coverage

Using the definitions provided in 6.2.7 and the test coverage items derived above:

$$C_{\text{cause-effect}} = \frac{6}{6} \times 100\,\% = 100\,\%$$

Thus, 100 % coverage of test coverage items for cause-effect graphing has been achieved.

B.2.8 State transition testing

B.2.8.1 General

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 Test basis

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 Determine required test coverage

The following levels of test coverage will be achieved for state transition testing of this example:

— 0-switch (see B.2.8.5 and B.2.8.6);

— all transitions (see B.2.8.7 and B.2.8.8);

— 1-switch coverage (see B.2.8.9 and B.2.8.10).

B.2.8.4 Create test model (TD1)

B.2.8.4.1 General

A state model is produced as the test model for state transition testing.

B.2.8.4.2 Test model for achieving switch coverage

State transition diagrams (STD) are most commonly used as state models when the required test coverage is a form of switch coverage. This is because switch coverage requires valid transitions to be exercised and these are a major attribute of state transition diagrams. The notation for state transition diagrams is illustrated below.

A STD consists of states, transition, events and actions (see Figure B.7). 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. As explained in 5.2.8.1, in state transition testing the test coverage may extend to all states of the state model, all transitions of the state model or the entire state model, depending on the coverage requirements of testing.

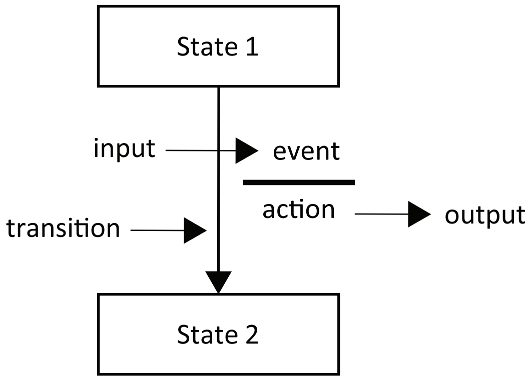


Figure B.7 — Generic state model

The STD for the test item manage_display_changes is show in Figure B.8.

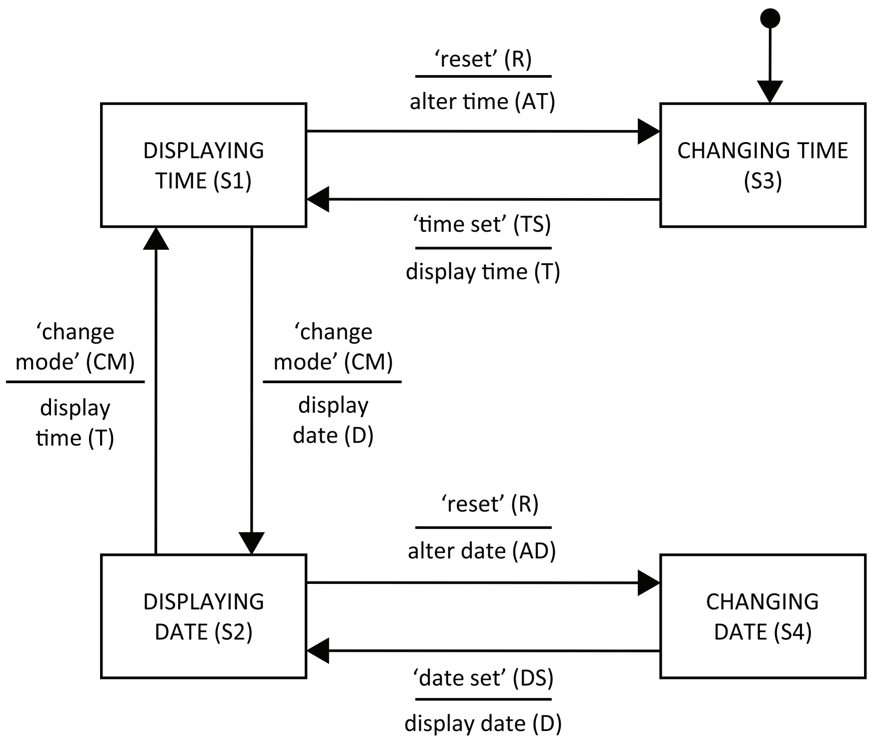


Figure B.8 — State transition diagram for manage_display_changes

B.2.8.4.3 Test model for achieving all transition coverage

State tables are often used as state models when the required test coverage includes attempting to exercise invalid transitions. This is because a state table explicitly shows both valid and invalid transitions, whereas the state transition diagram only explicitly shows valid transitions (all transitions not shown are considered invalid). One notation used for state tables is briefly described in Table B.30.

Table B.30 — 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 each "Entry X" = Finish State / Output or Action for the given Start State and Input.

The state table for manage_display_changes is shown as Table B.31.

Table B.31 — 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.

B.2.8.5 Identify test coverage items for 0-switch coverage (TD2)

To achieve full 0-switch coverage only the valid transitions need to be exercised.

The following test coverage items (valid single transitions) were identified from the state transition diagram shown in Figure B.8:

TCOVER1:	S1 to S2 with input CM
TCOVER2:	S1 to S3 with input R
TCOVER3:	S2 to S1 with input CM
TCOVER4:	S2 to S4 with input R
TCOVER5:	S3 to S1 with input TS
TCOVER6:	S4 to S2 with input DS

B.2.8.6 Derive test cases for 0-switch coverage (TD3)

Inputs to exercise the valid transitions can be determined from the state transition diagram (STD), as can the expected result, which can be determined by combining the expected output and final state of the transition in the STD.

The six test cases shown in Table B.32 provide 0-switch test coverage. Each test case is derived by selecting a test coverage item (a transition) and identifying the inputs, expected output and final state from the STD until all valid transitions are covered by a test case.

Table B.32 — 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 Item	TCOVER1	TCOVER2	TCOVER3	TCOVER4	TCOVER5	TCOVER6

NOTE A test procedure can be written for the six text cases in Table B.32 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 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 achieve 0-switch coverage (Cho 1987[6]). 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.2.8.7 Identify test coverage items for all transitions coverage (TD2)

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 (to achieve all transitions coverage). 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 the null transitions.

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 shown as Table B.33.

Table B.33 — State table to test case table mapping for manage_display_changes

	CM	R	TS	DS
S1	S2/D (TCOVER1)	S3/AT (TCOVER2)	S1/– (TCOVER7)	S1/– (TCOVER8)
S2	S1/T (TCOVER3)	S4/AD (TCOVER4)	S2/– (TCOVER9)	S2/– (TCOVER10)
S3	S3/– (TCOVER11)	S3/– (TCOVER12)	S1/T (TCOVER5)	S3/– (TCOVER13)
S4	S4/– (TCOVER14)	S4/– (TCOVER15)	S4/– (TCOVER16)	S2/D (TCOVER6)

The following test coverage items were identified from the state table above for all transitions coverage:

TCOVER1:	S1 to S2 with input CM	(valid transition)
TCOVER2:	S1 to S3 with input R	(valid transition)
TCOVER3:	S2 to S1 with input CM	(valid transition)
TCOVER4:	S2 to S4 with input R	(valid transition)
TCOVER5:	S3 to S1 with input TS	(valid transition)
TCOVER6:	S4 to S2 with input DS	(valid transition)
TCOVER7:	S1 to S1 with input TS	(invalid transition)
TCOVER8:	S1 to S1 with input DS	(invalid transition)
TCOVER9:	S2 to S2 with input TS	(invalid transition)
TCOVER10:	S2 to S2 with input DS	(invalid transition)
TCOVER11:	S3 to S3 with input CM	(invalid transition)
TCOVER12:	S3 to S3 with input R	(invalid transition)
TCOVER13:	S3 to S3 with input DS	(invalid transition)
TCOVER14:	S4 to S4 with input CM	(invalid transition)
TCOVER15:	S4 to S4 with input R	(invalid transition)
TCOVER16:	S4 to S4 with input TS	(invalid transition)

The first six test coverage items are the same as those identified to achieve 0-switch coverage (see B.2.8.5).

B.2.8.8 Derive test cases for all transitions coverage (TD3)

The test cases derived to achieve coverage of the first six test coverage items (TCOVER1 to TCOVER6) are shown in Table B.32.

As the first six test coverage items (TCOVER 1 to TCOVER 6) were derived under the 0-switch coverage tests (see B.2.8.5), the remaining test cases to cover the invalid transitions (TCOVER7 to TCOVER16) can now be derived as shown in Table B.34, where "–" represents a null transition:

Table B.34 — 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
Expected output	–	–	–	–	–	–	–	–	–	–
Finish state	S1	S1	S2	S2	S3	S3	S3	S4	S4	S4
Test coverage item	TCOV-ER7	TCOV-ER8	TCOV-ER9	TCOV-ER10	TCOV-ER11	TCOV-ER12	TCOV-ER13	TCOV-ER14	TCOV-ER15	TCOV-ER16

As Table B.34 shows, test cases that cover invalid test coverage items should not cause a transition away from the starting state (i.e. the start and finish states are the same). The test cases in Tables B.33 and B.34 combined will achieve all transitions coverage.

B.2.8.9 Identify test coverage items - 1-switch testing (TD2)

For 1-switch coverage, pairs of possible transitions need to be exercised. The following test coverage items can be identified from the STD to achieve 1-switch coverage:

TCOVER17:	S1 to S2 to S1 with inputs CM and CM
TCOVER18:	S1 to S2 to S4 with inputs CM and R
TCOVER19:	S1 to S3 to S1 with inputs R and TS
TCOVER20:	S3 to S1 to S2 with inputs TS and CM
TCOVER21	S3 to S1 to S3 with inputs TS and R
TCOVER22:	S2 to S1 to S2 with inputs CM and CM
TCOVER23:	S2 to S1 to S3 with inputs CM and R
TCOVER24:	S2 to S4 to S2 with inputs R and DS
TCOVER25:	S4 to S2 to S1 with inputs DS and CM
TCOVER26:	S4 to S2 to S4 with inputs DS and R

B.2.8.10 Derive test cases for 1-switch coverage (TD3)

Test cases can now be written to exercise all possible sequential pairs of transitions. In this example, there are ten, as shown in Table B.35.

Table B.35 — 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

Table B.35 *(continued)*

Test case	17	18	19	20	21	22	23	24	25	26
Input	CM	CM	R	TS	TS	CM	CM	R	DS	DS
Expected 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
Expected output	T	AD	T	D	AT	D	AT	D	T	AD
Finish state	S1	S4	S1	S2	S3	S2	S3	S2	S1	S4
Test coverage item	17	18	19	20	21	22	23	24	25	26

EXAMPLE 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). The intermediate states, and the inputs and outputs for each transition, are explicitly defined.

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

B.2.8.11 State transition testing coverage

Using the definitions provided in 6.2.8 and the test coverage items derived above:

$$C_{\text{0-switch}} = \frac{6}{6} \times 100\,\% = 100\,\%$$

$$C_{\text{all\_transitions}} = \frac{16}{16} \times 100\,\% = 100\,\%$$

$$C_{\text{N-switch}} = \frac{10}{10} \times 100\,\% = 100\,\%$$

Thus, 100 % coverage of test coverage items for 0-switch testing, all-transitions testing and 1-switch testing has been achieved.

B.2.9 Scenario testing

B.2.9.1 General

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.

There are two types of scenario testing demonstrated in B.2.9.2 and B.2.9.3. The first example is based on a generic form of the technique, while the second is a specific example based on use cases.

B.2.9.2 Scenario testing - example 1

B.2.9.2.1 Test basis

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.

Alternative scenarios

Withdrawal not approved, because:

— the user's bank card is rejected as it is unrecognized by the ATM

— the user enters their PIN incorrectly up to 2 times

— the user enters their PIN incorrectly three 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

NOTE In reality, additional scenarios, which address situations such as the user pressing cancel at any point in the process, are also possible.

B.2.9.2.2 Determine required test coverage

Coverage of main and alternative scenarios.

B.2.9.2.3 Create test model (TD1)

To enable the identification of test coverage items, a model of the test item is produced that identifies the scenarios (and the activities within them). The example model in Figure B.9 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.

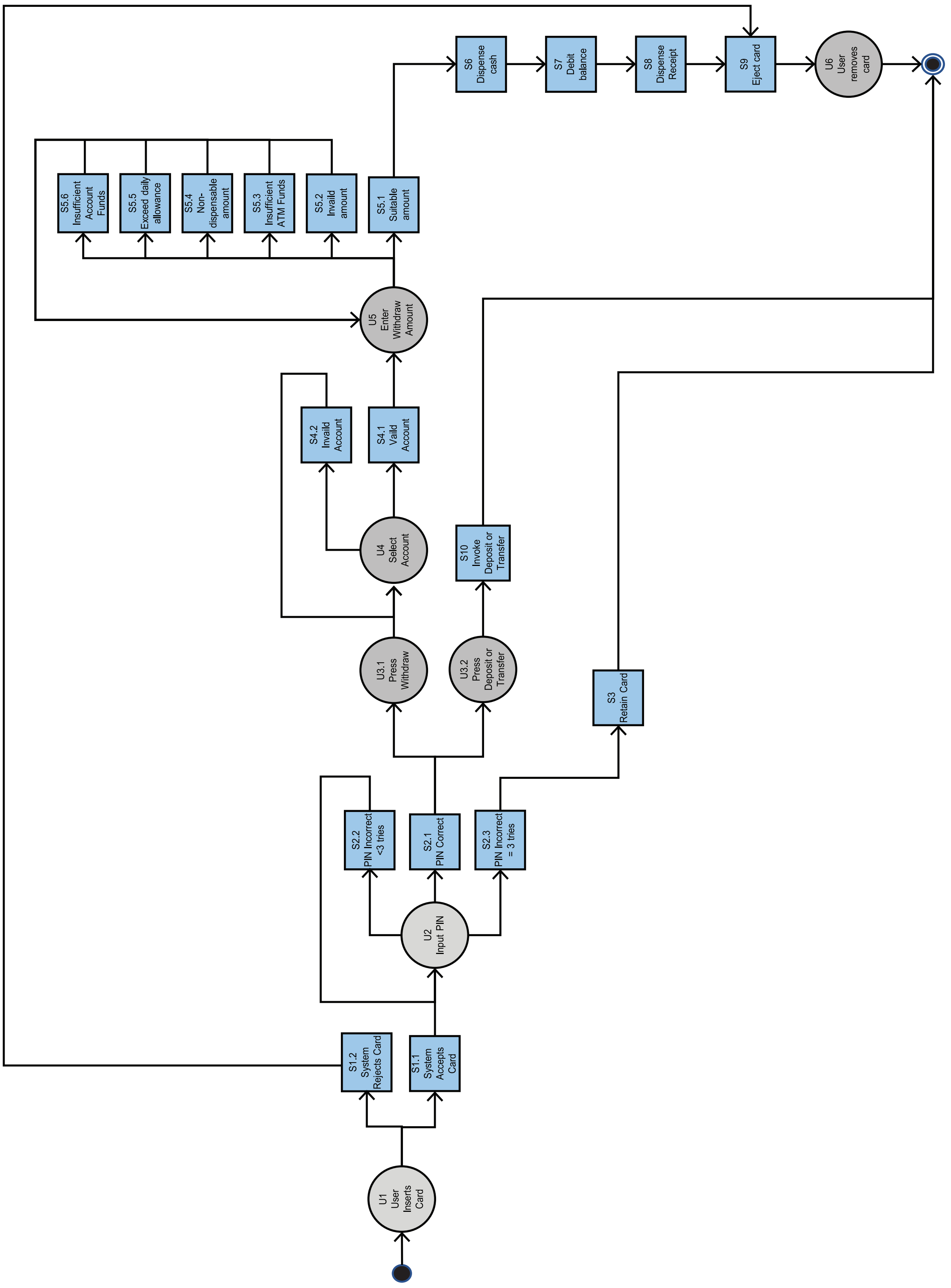


Figure B.9 — Flow of events diagram for withdraw_cash function

B.2.9.2.4 Identify test coverage items (TD2)

In scenario testing, the test coverage items to be covered are the main and alternative scenarios (i.e. they are the sequences of user and system interactions through the flow of events diagram that constitutes one scenario). There are 11 scenarios (SCEN) described in the specification, including one main and ten alternatives:

TCOVER1: Successful withdrawal of funds	(U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.1, S6, S7, S8, S9, U6)
TCOVER2: User's card is unrecognized by ATM	(U1, S1.2, S9, U6)
TCOVER3: User enters PIN incorrectly < 3 times	(U1, S1.1, U2, S2.2)
TCOVER4: User enters pin incorrectly 3 times	(U1, S1.1, U2, S2.2, U2, S2.2, U2, S2.3, S3)
TCOVER5: User selects deposit or transfer	(U1, S1.1, U2, S2.1, U3.2, S10)
TCOVER6: User selects incorrect account	(U1, S1.1, U2, S2.1, U3.1, U4, S4.2)
TCOVER7: User enters invalid withdrawal amount	(U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.2)
TCOVER8: Insufficient cash in the ATM	(U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.3)
TCOVER9: User enters non-dispensable amount	(U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.4)
TCOVER10: User enters amount exceeding daily allowance	(U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.5)
TCOVER11: Insufficient funds in user's account	(U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.6)

B.2.9.2.5 Derive test cases (TD3)

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.

The three test cases shown in Tables B.36 to B.38 provide examples of typical test cases for scenario-based testing. In practice, it would be necessary to derive a further 8 test cases to provide coverage of all 11 test coverage items (they would follow the same format as the first three).

NOTE 1 Within each test case, there are a wide variety of input values that can be chosen to populate each input field. Equivalence partitioning can be used to derive a set of values for populating each input field.

Table B.36 — Test cases for scenario testing

Test case #	1
Test case name	Successful withdrawal of funds
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

Table B.36 *(continued)*

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 coverage item	TCOVER1

Table B.37 — Test cases for scenario testing continued

Test case #	2
Test case name	User's card is unrecognized by ATM
Scenario path exercised	U1, S1.2, S9, U6
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 with an error message indicating that the card is invalid. The ATM is waiting for the customer to remove the invalid card.
Test coverage item	TCOVER2

Table B.38 — 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. System prompts user to re-enter their pin.
Test coverage item	TCOVER3

NOTE 2 The test cases shown in Tables B.36 to B.38 are each contained in a separate table for readability. In practice, they can be contained in a single table.

B.2.9.2.6 Scenario testing coverage

Using the definitions provided in 6.2.9 and the three test cases derived above:

$$C_{\text{scenarios}} = \frac{3}{11} \times 100\,\% = 27.3\,\%$$

B.2.9.3 Scenario testing - example 2

B.2.9.3.1 General

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 document with a fully worked example of this technique.

B.2.9.3.2 Test basis

Consider the example use case shown in Table B.39 for a test item change_password:

Table B.39 — 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 use 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
Basic 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”
Alternative flow – existing password incorrect	3.1	User enters their current password incorrectly
	3.2	User enters their new password correctly
	3.3	User enters their new password correctly
	3.4	User clicks OK
	3.5	System displays an error message “Current password entered incorrectly. Please try again.” and highlights all text in the Current Password field
Alternative flow – new password less than 8 characters	4.1	User enters a new password that is less than 8 characters long
	4.2	User clicks OK
	4.3	System displays an error message “New password must be at least 8 characters long. Please try again.”
Alternative flow – new password same as current password	5.1	User enters a new password that is the same as their current password
	5.2	User clicks OK
	5.3	System displays error message “New password must not be the same as current password. Please try again.”
Alternative flow – new passwords do not match	6.1	User re-enters new password that does not match the new password they entered at step 4
	6.2	User clicks OK
	6.3	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 (*)	
Frequency	Used the first time a new user logs into the system Typically used twice per user per year on average Can be invoked any time the user clicks the “Change Password” button	

NOTE Additional scenarios, which address situations such as the user entering invalid characters for their new password, can also exist.

B.2.9.3.3 Determine required rest coverage

Coverage of typical and alternative scenarios.

B.2.9.3.4 Create test model (TD1)

The test model is the use case description shown in Table B.39.

B.2.9.3.5 Identify test coverage items (TD2)

The test coverage items in use case testing are the typical and alternative scenarios, as follows.

TCOVER1:	Main flow
TCOVER2:	Alternative flow – existing password incorrect
TCOVER3:	Alternative flow – new password less than 8 characters
TCOVER4:	Alternative flow – new password same as current password
TCOVER5:	Alternative flow – new passwords do not match

B.2.9.3.6 Derive test cases (TD3)

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. See Tables B.40 to B.44.

Table B.40 — 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 coverage item	TCOVER1	
Use case steps covered	1, 2, 3, 4, 5, 6, 7	
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”

Table B.41 — Test cases for use case testing continued

Use case name	change_password
Test case name	Alternative flow – existing password incorrect
Description	User attempts to change password but enters their current password incorrectly

Table B.41 *(continued)*

Actors	User	
Test coverage item	TCOVER2	
Use case steps covered	1, 2, 3.1, 3.2, 3.3, 3.4, 3.5	
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 enters their new password correctly	New password is masked with asterisk (*) symbols
4	User enters their new password correctly	New password is masked with asterisk (*) symbols
5	User clicks OK	System displays error message "Current password entered incorrectly. Please try again."

Table B.42 — Test cases for use case testing continued

Use case name	change_password	
Test case name	Alternative flow – new password less than 8 characters	
Description	User attempts to change password but enters less than 8 characters for password	
Actors	User	
Test coverage item	TCOVER3	
Use case steps covered	1, 2, 3, 4.1, 4.2, 4.3	
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 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 B.43 — Test cases for use case testing continued

Use case name	change_password	
Test case name	Alternative flow – new password same as current password	
Description	User attempts to change password but enters new password matching old password	
Actors	User	
Test coverage item	TCOVER4	
Use case steps covered	1, 2, 3, 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 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 B.44 — Test cases for use case testing continued

Use case name		change_password
Test case name		Alternative flow – new passwords do not match
Description		User attempts to change password but their new passwords do not match
Actors		User
Test coverage item		TCOVER5
Use case steps covered		1, 2, 3, 4, 6.1, 6.2, 6.3
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.2.9.3.7 Use case testing coverage

Using the definitions for calculating scenario test coverage provided in 6.2.9 and the test coverage items derived above:

$$C_{\text{scenarios}} = \frac{5}{5} \times 100\% = 100\%$$

B.2.10 Random testing

B.2.10.1 General

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 randomly from this input domain.

B.2.10.2 Test basis

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.2.10.3 Determine required test coverage

Coverage for random testing cannot be measured. However, it is possible to set a criterion for determining when sufficient test cases have been derived, such as by defining a set number of test cases or by setting an amount of effort to be spent on the random testing activity, among others. For this example, the required test coverage is set at four test cases.

B.2.10.4 Create test model (TD1)

The test model in random testing shows the domain of all possible inputs from which test input values can be selected for each input parameter. It can be shown as:

INPUT1:	*x* - range - 320..+ 320, in increments of $1/2^6$
INPUT2:	*y* - range - 240..+ 240, in increments of $1/2^7$

B.2.10.5 Identify test coverage items (TD2)

There are no recognized test coverage items from random testing.

B.2.10.6 Derive test cases (TD3)

Test cases are constructed by first choosing an input distribution and then applying that input distribution to the input domain and determining the expected result of each test case (shown as 'output' for the two output parameters *r* and *H* in Table B.45). Since no information is available about the operational distribution of the inputs to the test item in this example, a uniform distribution is chosen. From the definitions, it can be seen that any one randomly chosen input for *x* can take one of 41,024 values (641×2^6), while *y* can take one of 61,568 values (481×2^7). 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 values from the input domain of the test item for both *x* and *y*, each test case will cover both test input values. 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 four test cases in Table B.45 can be defined.

Table B.45 — 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
Output *r* calculated as ($r = \text{sqrt}(x^2 + y^2)$)	269.395305	78.804949	323.216025	99.522847
Output *H* calculated as ($\cos H = x/r$)	2.058024	1.430554	0.503870	3.050407

B.2.10.7 Random testing coverage

As stated in 6.2.10, there is currently no industry agreed approach for calculating coverage of test coverage items for random testing.

B.2.10.8 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 or implementation language);

— 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 B.2.10.2, the coordinate transformation test item can be checked automatically using the inverse function approach. In this case, $r\cos H=x$ can be obtained directly from the test basis for the test item. By some analysis $r\sin H=y$ 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[6]) is a useful method for organizing the input domain for random sampling before test case design.

B.2.11 Metamorphic testing

B.2.11.1 General

Metamorphic testing uses metamorphic relations to derive follow-up test cases from a successful source test case. It is particularly useful when the determination of expected results is difficult, such as for a complex system with multiple inputs (e.g. a neural network working with big data).

B.2.11.2 Test basis

Consider a test item that searches for available hotel rooms given a set of search parameters, with the following test basis:

The system will provide users with a list of available accommodation based on their input requirements. Inputs and outputs shall be as follows:

Inputs

start date – dd-mm-yy (entry field will prevent dates in the past)

end date – dd-mm-yy (entry field will prevent dates earlier than one day after the start date) location – picked from a list

number of guests - 1 to 10

room type – single, double, family

swimming pool – yes/no/not specified

air conditioning – yes/no/not specified

minimum guest rating – 1*, 2*, 3*, 4*, 5*/not specified

free cancellation – yes/no/not specified

WiFi – yes/no/not specified

car park – yes/no/not specified

free breakfast – yes/no/not specified

Outputs

List of available accommodation (with links)

B.2.11.3 Determine required test coverage

Coverage for metamorphic testing cannot be measured. However, it is possible to set a criterion for determining when sufficient test cases have been derived, such as by defining a maximum number of required test cases or by setting an amount of effort to be spent on the random testing activity, among others.

B.2.11.4 Create test model (TD1)

The test model in metamorphic testing is a set of metamorphic relations based on the required behaviour that relates pairs of test inputs and pairs of test outputs. The following metamorphic relations have been defined:

MR1: Given a valid search, if the number of guest requirements is reduced, then the size of the list of available accommodation should increase (or stay the same).

MR2: Given a valid search, if the number of guest requirements is increased, then the size of the list of available accommodation should decrease (or stay the same).

MR3: Given a valid search, if the requested dates are changed to a subset of the original dates, then the list of available accommodation should increase (or stay the same).

MR4: Given a valid search, if the order of guest requirements is changed, then the list of available accommodation should stay the same.

MR5: Given a valid search, if the required minimum star rating is reduced, then the list of available accommodation should increase (or stay the same).

B.2.11.5 Identify test coverage items (TD2)

The test coverage items are the individual metamorphic relations:

MR1, MR2, MR3, MR4 and MR5.

B.2.11.6 Derive test cases (TD3)

Test cases can now be derived to cover the metamorphic relations. See Tables B.46 to B.49.

For MR1: Given a valid search, if the number of guest requirements is reduced, then the size of the list of available accommodation should increase (or stay the same).

Table B.46 — Test cases for metamorphic testing

Source test case (executed and validated) – S1
Test inputs: start date = 10-02-20 end date = 12-02-20 location = Clacton-on-sea number of guests = 2 room type = double swimming pool = no air conditioning = no minimum guest rating = 3* free cancellation = not specified WiFi = yes car park = not specified free breakfast = yes
Actual results: — Hotel Miramar — Royal Hotel — Ocean Hotel — Esplanade Hotel — West Hotel

Table B.47 — Test cases for metamorphic testing

Follow-up test case – MR1-S1-F1
Test inputs: start date = 10-02-20 end date = 12-02-20 location = Clacton-on-sea number of guests = 2 room type = double swimming pool = no air conditioning = no minimum guest rating = 3* free cancellation = not specified WiFi = ~~yes~~ not specified ← **modified test input** car park = not specified free breakfast = yes

Table B.47 *(continued)*

Expected results: As a minimum: Hotel Miramar Royal Hotel Ocean Hotel Esplanade Hotel West Hotel

NOTE In the above example, the source test case (S1) and the metamorphic relation (MR1) indicate that the expected result must include all the hotels from S1 but it is expected to also include others now that the requirement for WiFi has been relaxed. However, when the follow-up test case (MR1-S1-F1) is created, there's the restriction to use the information provided in S1 and MR1; and as neither of these list the other hotels in the resort, they cannot be included explicitly in MR1-S1-F1 – all that can be done, as shown, is to state the expected result as the previous result as a minimum.

Table B.48 — Test cases for metamorphic testing

Follow-up test case – MR1-S1-F2
Test inputs: start date = 10-02-20 end date = 12-02-20 location = Clacton-on-sea number of guests = 2 room type = double swimming pool = no air conditioning = ~~no~~ not specified ← **modified test input** minimum guest rating = 3* free cancellation = not specified WiFi = yes car park = not specified free breakfast = yes
Expected results: As a minimum: Hotel Miramar Royal Hotel Ocean Hotel Esplanade Hotel West Hotel

Further follow-up test cases for MR1 and S1 can be derived by changing the requirements for each of the following parameters to 'not specified':

— swimming pool

— minimum guest rating

— free cancellation

— car park

— free breakfast

This example has shown how a number of follow-up test cases can be derived using the MR1 metamorphic relation and one source test case (S1). If more source test cases are executed and validated, then further follow-up test cases can be derived for MR1. For instance, a second source test case, S2, with different search dates can be executed and validated; and then a number of follow-up test cases using the MR1 metamorphic relation with this second source test case (e.g. MR1-S2-F1, MR1-S2-F2) can be derived.

For MR2: Given a valid search, if the number of guest requirements is increased, then the size of the list of available accommodation should decrease (or stay the same).

Table B.49 — Test cases for metamorphic testing

Source test case (executed and validated) –S2
Test inputs: start date = 03-04-20 end date = 07-02-20 location = Jaywick Sands number of guests = 1 room type = single swimming pool = yes air conditioning = yes minimum guest rating = 5* free cancellation = yes WiFi = yes car park = yes free breakfast = not specified
Actual results: The Excelsior Hotel
Follow-up test case – MR2-S2-F1
Test inputs: start date = 03-04-20 end date = 07-02-20 location = Jaywick Sands number of guests = 1 room type = single swimming pool = yes air conditioning = yes minimum guest rating = 5* free cancellation = yes WiFi = yes car park = yes free breakfast = yes ← **modified test input**
Expected results: As a maximum: The Excelsior Hotel

No further follow-up test cases for MR2 and MR2-S1 can be derived as there was only one item in the source test case that was 'not specified', however if more source test cases are executed and validated then further follow-up test cases can be derived for MR2

Test cases can be derived for MR3, MR4 and MR5 in a similar manner.

B.2.11.7 Metamorphic testing coverage

As stated in 6.2.11, there is currently no industry agreed approach for calculating coverage of test coverage items for metamorphic testing.

B.2.12 Requirements-based testing

B.2.12.1 General

Requirements-based testing simply covers specified atomic requirements.

B.2.12.2 Test basis

Consider a test item that controls a car spoiler dependent on the speed of the car, with the following test basis:

> The car spoiler control system shall do set spoiler to 'low' when input speed does not exceed 50 km/h, and if the speed is higher than 120 km/h it shall set spoiler to 'high', otherwise it shall set spoiler to 'medium'. Valid input speeds are from 0 km/h to 220 km/h – invalid speeds will cause an error message. All speeds are provided as integers.

B.2.12.3 Determine required test coverage

All specified requirements should be covered by tests.

B.2.12.4 Create test model (TD1)

The test model in requirements-based testing is a set of atomic requirements, typically stated in simple sentences. The following atomic requirements have been specified:

R1: The system shall set spoiler to 'low' when the input speed is between 0 km/h and 50 km/h inclusive.

R2: The system shall set spoiler to 'high' when input speed is higher than 120 km/h and no greater than 220 km/h.

R3: The system shall set spoiler to 'medium' when input speed is higher than 50 km/h and no greater than 120 km/h.

R4: The system shall generate an error message for invalid speeds outside the range from 0 km/h to 220 km/h.

B.2.12.5 Identify test coverage items (TD2)

The test coverage items are the individual atomic requirements:

R1, R2, R3 and R4.

B.2.12.6 Derive test cases (TD3)

Test cases can now be derived to cover the atomic requirements. See Table B.50.

Table B.50 — Test cases for requirements-based testing

Test case	1	2	3	4
Input (speed)	33 km/h	174 km/h	81 km/h	222 km/h
Expected output (spoiler position)	low	high	medium	error message
Requirement exercised (test coverage item)	R1	R2	R3	R4

B.2.12.7 Requirements-based testing coverage

Using the definitions for calculating requirements-based test coverage provided in 6.2.12 and the test coverage items derived above:

$$C_{\text{requirements}} = \frac{4}{4} \times 100\,\% = 100\,\%$$

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 5.3 and 6.3. Each example follows the test design and implementation process that is defined in ISO/IEC/IEEE 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 5.1, in practice most of the techniques defined in this document can be used interchangeably.

C.2 Structure-based test design technique examples

C.2.1 Statement testing

C.2.1.1 General

The aim of statement testing is to derive a set of test cases that cover the executable statements of the test item according to a chosen level of statement coverage.

The two main 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.

Statements which are associated with machine code are typically regarded as executable. For instance, all of the following are regarded as executable:

— assignments;

— loops and selections;

— procedure and function calls;

— variable declarations with explicit initializations;

— 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. Thus full statement coverage can be achieved without exercising with b FALSE.

C.2.1.2 Test basis

Consider the following test item in the Ada programming language, 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 Create test model (TD1)

The test model in statement testing is the source code, showing the executable statements in the code.

C.2.1.4 Identify test coverage items (TD2)

The test coverage items in statement testing are the individual executable statements; these are shown below by their line numbers in the source code.

TCOVER1:	Statement 1
TCOVER2:	Statement 2
TCOVER3	Statement 3
TCOVER4:	Statement 4
TCOVER5:	Statement 5
TCOVER6:	Statement 6
TCOVER7:	Statement 7
TCOVER8:	Statement 10
TCOVER9:	Statement 11
TCOVER10:	Statement 13
TCOVER11:	Statement 15

C.2.1.5 Derive test cases (TD3)

In statement testing, each statement must be covered by at least one test case. Test cases are derived by first identifying sub-paths through the code 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 particular example, only one test case is required to cover all statements in the code, because the iteration in the code allows all statements to be covered by just two input values. See Table C.1.

Table C.1 — Test cases for statement testing

Test case	Input	Expected result	Test coverage items (statements)
1	2	2 is prime	1, 2, 3, 4, 9, 10, 11, 12, 13, 14
	4	2 is a factor of 4	2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14
	EOF	End of prime number program	2, 15

NOTE In this example, there is one test case, but three separate sets of input values and expected results, as there is an iteration included in the code.

C.2.1.6 Statement testing coverage

Using the definitions provided in 6.3.1 and the test coverage items derived above:

$$C_{\text{statement}}=\frac{11}{11}\times 100\,\%=100\,\%$$

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

C.2.2 Branch/decision testing

C.2.2.1 General

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.

C.2.2.2 Test basis

> 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.2.2.3 Create test model (TD1)

The creation of the test model 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 is to divide the code into basic blocks. These are sequences of instructions with no branches into the block (except to the beginning of the code) and no branches out of the block (except at the end of the code). 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 C.1. The graph has one entry point, B1, and two exit points, B7 and B9.

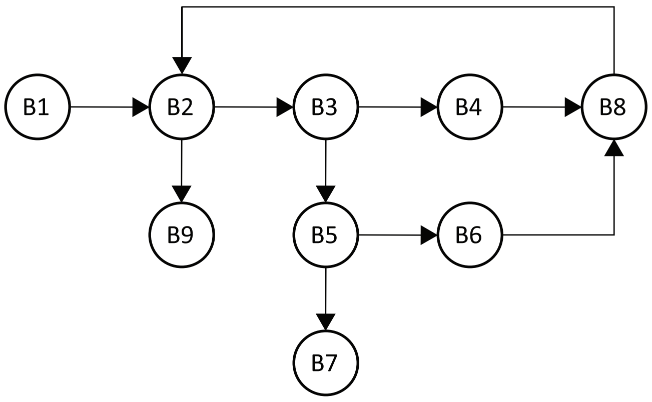


Figure C.1 — Control flow graph for binsearch

The 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.

C.2.2.4 Identify test coverage items (TD2)

C.2.2.4.1 Options for identification of test coverage items

The test coverage items for branch coverage will be different from those for decision coverage. This is demonstrated by options a and b specified in C.2.2.4.2 and C.2.2.4.3.

C.2.2.4.2 Option a: identify test coverage items for branch coverage

For branch coverage, the test coverage items are the branches in the control flow graph. In this example there are ten test coverage items for branch coverage, as follows:

BRANCH-TCOVER1:	B1 → B2
BRANCH-TCOVER2:	B2 → B3
BRANCH-TCOVER3:	B2 → B9
BRANCH-TCOVER4:	B3 → B4
BRANCH-TCOVER5:	B3 → B5
BRANCH-TCOVER6:	B4 → B8
BRANCH-TCOVER7:	B5 → B6
BRANCH-TCOVER8:	B5 → B7
BRANCH-TCOVER9:	B6 → B8
BRANCH-TCOVER10:	B8 → B2

C.2.2.4.3 Option b: identify test coverage items for decision coverage

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
DECISION-TCOVER2:	B2 = false
DECISION-TCOVER3:	B3 = true
DECISION-TCOVER4:	B3 = false
DECISION-TCOVER5:	B5 = true
DECISION-TCOVER6:	B5 = false

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 Derive test cases (TD3)

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 decisions that have 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) outcome and hence provides 1/6 = 16.7 % decision coverage. The path executes 2 out of the 10 branches, giving 20 % branch coverage (which is not the same as the decision coverage).

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 in Table C.2.

Table C.2 — Test cases for binsearch

Test case	Inputs			Decisions exercised (underlined)	Test coverage items	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 Branch testing coverage

Using the definitions provided in 6.3.2 and the test coverage items derived above:

$$C_{\text{branch}} = \frac{10}{10} \times 100\,\% = 100\,\%$$

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

C.2.2.7 Decision testing coverage

Using the definitions provided in 6.3.3 and the test coverage items derived above:

$$C_{\text{decision}} = \frac{6}{6} \times 100\,\% = 100\,\%$$

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 General

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 Test basis

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 can 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 Create test model (TD1)

In these three test design techniques, the set of decisions in the source code is the test model. In this example, there is one decision that needs to be covered:

```
A or (B and C)
```

This is applicable to branch condition testing, branch condition combination testing, and modified condition/decision coverage testing.

C.2.3.4 Branch condition testing

C.2.3.4.1 Identify test coverage items (TD2)

Branch condition testing examines the individual conditions within multi-condition decisions, with the aim that each individual condition and each decision takes on both true and false values. The test coverage items are the Boolean values (true/false) of the conditions within decisions and the decision outcomes. 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 plus the decision outcomes to be evaluated both true and false. Therefore, the test coverage items for this technique are:

TCOVER1:	A = TRUE
TCOVER2:	A = FALSE
TCOVER3	B = TRUE
TCOVER4:	B = FALSE
TCOVER5:	C = TRUE
TCOVER6:	C = FALSE
TCOVER7:	A or (B and C) = TRUE
TCOVER8:	A or (B and C) = FALSE

C.2.3.4.2 Derive test cases (TD3)

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 set of test inputs shown in Table C.3 (note that there are alternative sets of test inputs which will also achieve full branch condition coverage):

Table C.3 — Test cases for branch condition testing

Test case	A	B	C	A or (B and C)	Test coverage items
1	FALSE	FALSE	FALSE	FALSE	TCOVER 2, 4, 6
2	TRUE	TRUE	TRUE	TRUE	TCOVER 1, 3, 5

NOTE The test cases contained in Table C.3 are not "complete" in that they do not include expected results.

Branch condition coverage can often be achieved with just two test cases, irrespective of the number of actual Boolean conditions comprising the overall condition.

C.2.3.4.3 Branch condition testing coverage

Using the definitions provided in 6.3.4 and the test coverage items derived above:

$$C_{\text{branch\_condition}} = \frac{8}{8} \times 100\,\% = 100\,\%$$

Thus, 100 % coverage of test coverage items for branch condition testing has been achieved.

C.2.3.5 Branch condition combination testing

C.2.3.5.1 Identify test coverage items (TD2)

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
TCOVER2:	A = TRUE,	B = FALSE,	C = FALSE
TCOVER3:	A = FALSE,	B = TRUE,	C = FALSE
TCOVER4:	A = TRUE,	B = TRUE,	C = FALSE
TCOVER5:	A = FALSE,	B = FALSE,	C = TRUE
TCOVER6:	A = TRUE,	B = FALSE,	C = TRUE
TCOVER7:	A = FALSE,	B = TRUE,	C = TRUE
TCOVER8:	A = TRUE,	B = TRUE,	C = TRUE

C.2.3.5.2 Derive test cases (TD3)

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 test cases shown in Table C.4.

Table C.4 — Test cases for branch condition combination testing

Test case	A	B	C	Test coverage items
1	FALSE	FALSE	FALSE	TCOVER1
2	TRUE	FALSE	FALSE	TCOVER2
3	FALSE	TRUE	FALSE	TCOVER3
4	TRUE	TRUE	FALSE	TCOVER4

Table C.4 *(continued)*

Test case	A	B	C	Test coverage items
5	FALSE	FALSE	TRUE	TCOVER5
6	TRUE	FALSE	TRUE	TCOVER6
7	ALSE	TRUE	TRUE	TCOVER7
8	TRUE	TRUE	TRUE	TCOVER8

NOTE The test cases contained in Table C.4 are not “complete” in that they do not include expected results.

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 in practice for complex conditions.

C.2.3.5.3 Branch condition combination testing coverage

Using the definitions provided in 6.3.5 and the test coverage items derived above:

$$C_{\text{branch\_condition\_combination}} = \frac{8}{8} \times 100\,\% = 100\,\%$$

Thus, 100 % coverage of test coverage items for branch condition combination testing has been achieved.

C.2.3.6 Modified condition/decision coverage testing

C.2.3.6.1 General

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-178C and automotive software, as required by ISO 26262 (all parts). 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.2.3.6.2 Identify test coverage items (TD2)

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 a pair 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
TCOVER2:	A = TRUE,	B = FALSE,	C = TRUE	OUTCOME = TRUE

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

TCOVER3:	A = FALSE,	B = FALSE,	C = TRUE	OUTCOME = FALSE
TCOVER4:	A = FALSE,	B = TRUE,	C = TRUE	OUTCOME = TRUE

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

TCOVER5:	A = FALSE,	B = TRUE,	C = FALSE	OUTCOME = FALSE
TCOVER6:	A = FALSE,	B = TRUE,	C = TRUE	OUTCOME = TRUE

Having created these pairs of test coverage items for each decision condition separately, it can be seen that TCOVER1 and TCOVER3 are the same, and that TCOVER4 and TCOVER6 are the same. Therefore, the duplicate test coverage items TCOVER3 and TCOVER6 will not be used as a basis for deriving test cases in the next step.

C.2.3.6.3 Derive test cases (TD3)

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 with the set of test cases shown in Table C.5.

Table C.5 — Overall set of test cases

Test case	A	B	C	Expected result	Test coverage items
1	FALSE	FALSE	TRUE	FALSE	TCOVER1, TCOVER3
2	TRUE	FALSE	TRUE	TRUE	TCOVER2
3	FALSE	TRUE	TRUE	TRUE	TCOVER4, TCOVER6
4	FALSE	TRUE	FALSE	FALSE	TCOVER5

NOTE The test cases contained in Table C.5 are not “complete” in that they do not include expected results.

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 can have been shown to independently affect the outcome of the condition by the pair of test cases shown in Table C.6.

Table C.6 — Alternative MCDC test cases

Case	A	B	C	Outcome
X	FALSE	TRUE	FALSE	FALSE
Y	TRUE	TRUE	FALSE	TRUE

Test case X is the same as test case 4 above, but test case Y is one which has not been previously used. As MCDC has already been achieved, test case Y is not required for coverage purposes.

To achieve 100 % modified condition/decision coverage requires a minimum of n + 1 test cases, where n is the number of Boolean conditions within the decision condition. In contrast, branch condition combination coverage requires 2^n test 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.2.3.6.4 Modified condition/decision coverage

Using the definitions provided in 6.3.6 and the test coverage items derived above:

$$C_{\mathrm{MCDC}} = \frac{4}{4} \times 100\,\% = 100\,\%$$

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

C.2.3.7 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.2.3.8 Optimised expressions

Some programming languages and compilers “short circuit” the evaluation of Boolean operators by ignoring any part of an expression that does not have a direct impact on the outcome of that expression.

For example, the C and C++ languages always short circuit the Boolean "and" (&&) and "or" (||) operators, and the Ada programming 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.2.3.9 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 one of these three test design techniques and its associated coverage measure as a supplement to branch testing and branch or decision coverage. Branch testing will address all simple decisions, multi-way decisions, and all loops. Condition testing will then address the decisions which include Boolean conditions.

In practice, 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 General

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 definition-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 Test basis

Consider the data flow testing of the following test item in the Ada programming language:

```
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 Create test model (TD1)

The test model for data flow testing comprises the definition-use pairs for each of the variables in the program. The first step is to list the variables used in the program. 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, computation-use (c-use) or predicate-use (p-use)). See Table C.7.

Table C.7 — Occurrence of variables and their categories

	Category		
Line	**definition**	**c-use**	**p-use**
0	A, B, C		
1	Discrim	A, B, C	

Table C.7 *(continued)*

Line	Category		
	definition	**c-use**	**p-use**
2			
3			
4			Discrim
5	Is_Complex		
6			
7	Is_Complex		
8			
9			Is_Complex
10	R1	A, B, Discrim	
11	R2	A, B, Discrim	
12			
13		R1, R2, Is_Com-plex	

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. See Table C.8.

Table C.8 — definition-use pairs and their type

Definition-use pair (start line -> end line)	Variables	
	c-use	p-use
$0 \rightarrow 1$	A	
	B	
	C	
$0 \rightarrow 10$	A	
	B	
$0 \rightarrow 11$	A	
	B	
$1 \rightarrow 4$		Discrim
$1 \rightarrow 10$	Discrim	
$1 \rightarrow 11$	Discrim	
$5 \rightarrow 9$		Is_Complex
$7 \rightarrow 9$		Is_Complex
$10 \rightarrow 13$	R1	
$11 \rightarrow 13$	R2	
$5 \rightarrow 13$	Is_Complex	
$7 \rightarrow 13$	Is_Complex	

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.2.4.4 All-definitions testing

C.2.4.4.1 Identify test coverage items for all-definitions testing (TD2)

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.

Table C.9 shows one set of def-use pairs that meet this criterion.

Table C.9 — All-definitions testing

Test coverage items	Variables	definition-use pair
TCOVER1	A	$0 \rightarrow 1$
TCOVER2	B	$0 \rightarrow 1$
TCOVER3	C	$0 \rightarrow 1$
TCOVER4	Discrim	$1 \rightarrow 4$
TCOVER5	Is_Complex	$5 \rightarrow 9$
TCOVER6	Is_Complex	$7 \rightarrow 9$
TCOVER7	R1	$10 \rightarrow 13$
TCOVER8	R2	$11 \rightarrow 13$

C.2.4.4.2 Derive test cases for all-definitions testing (TD3)

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 test set shown in Table C.10 would satisfy this requirement.

Table C.10 — Test cases for all-definitions testing

Test case	All-definitions				Inputs			Expected result		
	Variables	Definition-use pairs	Sub-paths	Test coverage items	A	B	C	Is_Complex	R1	R2
1	Is_Complex	$7 \rightarrow 9$	7-8-9	TCOVER6	1	2	1	FALSE	-1	-1
	R1	$10 \rightarrow 13$	10-11-12-13	TCOVER7						
	R2	$11 \rightarrow 13$	11-12-13	TCOVER8						
2	A,B,C,	$0 \rightarrow 1$	0-1	TCOVER1, TCOVER2, TCOVER3	1	1	1	TRUE	unass.	unass.
	Discrim	$1 \rightarrow 4$	1-2-3-4	TCOVER4						
	Is_Complex	$5 \rightarrow 9$	5-8-9	TCOVER5						

C.2.4.4.3 All-definitions testing coverage

Using the definitions provided in 6.3.7.1 and the test coverage items derived above:

$$C_{\text{all-definitions}} = \frac{8}{8} \times 100\,\% = 100\,\%$$

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

C.2.4.5 All-c-uses testing

C.2.4.5.1 Identify test coverage items for all-c-uses testing (TD2)

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. Table C.11 shows one set of def-use pairs that meet this criterion.

Table C.11 — All-c-uses testing

Test coverage items	Variable	Definition-use pair	Sub-path
TCOVER1	A	0 → 1	0-1
TCOVER2	B	0 → 1	0-1
TCOVER3	C	0 → 1	0-1
TCOVER4	A	0 → 10	0-1-2-3-4-6-7-8-9-10
TCOVER5	B	0 → 10	0-1-2-3-4-6-7-8-9-10
TCOVER6	A	0 → 11	0-1-2-3-4-6-7-8-9-10-11
TCOVER7	B	0 → 11	0-1-2-3-4-6-7-8-9-10-11
TCOVER8	Discrim	1 → 10	1-2-3-4-6-7-8-9-10
TCOVER9	Discrim	1 → 11	1-2-3-4-6-7-8-9-10-11
TCOVER10	R1	10 → 13	10-11-12-13
TCOVER11	R2	11 → 13	11-12-13
TCOVER12	Is_Complex	5 → 13	5-8-9-12-13
z	Is_Complex	7 → 13	7-8-9-10-11-12-13

C.2.4.5.2 Derive test cases for all-c-uses Testing (TD3)

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. See Table C.12.

Table C.12 — Test cases for all-c-uses testing

Test case	All-c-uses				Inputs			Expected result		
	Variables	Definition-use pairs	Sub-paths	Test coverage items	A	B	C	Is_Complex	R1	R2
1	A, B, C	0 → 1	0-1	TCOVER1, TCOVER2, TCOVER3	1	2	1	FALSE	-1	-1
	A, B	0 → 10,	0-1-2-3-4-6-7-8-9-10	TCOVER4, TCOVER5						
	A, B	0 → 11	0-1-2-3-4-6-7-8-9-10-11	TCOVER6, TCOVER7						
	Discrim	1 → 10	1-2-3-4-6-7-8-9-10	TCOVER8						
		1 → 11	1-2-3-4-6-7-8-9-10-11	TCOVER9						
	R1	10 → 13	10-11-12-13	TCOVER10						
	R2	11 → 13	11-12-13	TCOVER11						

Table C.12 *(continued)*

Test case	All-c-uses				Inputs			Expected result		
	Variables	Definition-use pairs	Sub-paths	Test coverage items	A	B	C	Is_Complex	R1	R2
	Is_Complex	$7 \rightarrow 13$	7-8-9-10-11-12-13	TCOVER13						
2	Is_Complex	$5 \rightarrow 13$	5-8-9-12-13	TCOVER12	1	1	1	TRUE	unass.	unass.

C.2.4.5.3 All-c-uses testing coverage

Using the definitions provided in 6.3.7.2 and the test coverage items derived above:

$$C_{\text{all-c-uses}} = \frac{13}{13} \times 100\,\% = 100\,\%$$

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

C.2.4.6 All-p-uses testing

C.2.4.6.1 Identify test coverage items for all-p-uses testing (TD2)

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.

Table C.13 shows one set of def-use pairs that meet this criterion.

Table C.13 — All-p-uses testing

Test coverage items	Variables	Definition-use pair
TCOVER1	Discrim	$1 \rightarrow 4$
TCOVER2	Is_Complex	$5 \rightarrow 9$
TCOVER3	Is_Complex	$7 \rightarrow 9$

C.2.4.6.2 Derive test cases for all-p-uses testing (TD3)

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 test set shown in Table C.14 would satisfy this requirement:

Table C.14 — Test cases for all-p-uses testing

Test case	All-p-uses				Inputs			Expected result		
	Variables	Definition-use pair	sub-paths	Test coverage items	A	B	C	Is_Complex	R1	R2
1	Is_Complex	$7 \rightarrow 9$	7-8-9	TCOVER3	1	2	1	FALSE	-1	-1
2	Discrim	$1 \rightarrow 4$	1-2-3-4	TCOVER1	1	1	1	TRUE	unass.	unass.
	Is_Complex	$5 \rightarrow 9$	5-8-9	TCOVER2						

C.2.4.6.3 All-p-uses testing coverage

Using the definitions provided in 6.3.7.3 and the test coverage items derived above:

$$C_{\text{all-p-uses}} = \frac{3}{3} \times 100\,\% = 100\,\%$$

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

C.2.4.7 All-uses testing

C.2.4.7.1 Identify test coverage items for all-uses testing (TD2)

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.

Table C.15 shows one set of def-use pairs that meet this criterion.

Table C.15 — All-uses testing

Test coverage items	Variables	d-u pair	Sub-path
TCOVER1	A	$0 \rightarrow 1$	0-1
TCOVER2	B	$0 \rightarrow 1$	0-1
TCOVER3	C	$0 \rightarrow 1$	0-1
TCOVER4	A	$0 \rightarrow 10$	0-1-2-3-4-6-7-8-9-10
TCOVER5	B	$0 \rightarrow 10$	0-1-2-3-4-6-7-8-9-10
TCOVER6	A	$0 \rightarrow 11$	0-1-2-3-4-6-7-8-9-10-11
TCOVER7	B	$0 \rightarrow 11$	0-1-2-3-4-6-7-8-9-10-11
TCOVER8	Discrim	$1 \rightarrow 4$	1-2-3-4
TCOVER9	Discrim	$1 \rightarrow 10$	1-2-3-4-6-7-8-9-10
TCOVER10	Discrim	$1 \rightarrow 11$	1-2-3-4-6-7-8-9-10-11
TCOVER11	Is_Complex	$5 \rightarrow 9$	5-8-9
TCOVER12	Is_Complex	$7 \rightarrow 9$	7-8-9
TCOVER13	R1	$10 \rightarrow 13$	10-11-12-13
TCOVER14	R2	$11 \rightarrow 13$	11-12-13
TCOVER15	Is_Complex	$5 \rightarrow 13$	5-8-9-12-13
TCOVER16	Is_Complex	$7 \rightarrow 13$	7-8-9-10-11-12-13

C.2.4.7.2 Derive test cases for all-uses testing (TD3)

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 test set shown in Table C.16 would satisfy this requirement:

Table C.16 — Test cases for all-uses testing

Test case	All-uses				Inputs			Expected result		
	Variables	**d-u pair**	**Sub-paths**	**Test coverage items**	**A**	**B**	**C**	**Is_Complex**	**R1**	**R2**
1	A, B, C	0 → 1	0-1	TCOVER1, TCOVER2, TCOVER3	1	2	1	FALSE	-1	-1
	A, B	0 → 10	0-1-2-3-4-6-7-8-9-10	TCOVER4, TCOVER5						
	A, B	0 → 11	0-1-2-3-4-6-7-8-9-10-11	TCOVER6, TCOVER7						
	Discrim	1 → 4	1-2-3-4	TCOVER8						
		1 → 10	1-2-3-4-6-7-8-9-10	TCOVER9						
		1 → 11	1-2-3-4-6-7-8-9-10-11	TCOVER10						
	Is_Complex	7 → 9	7-8-9	TCOVER12						
	R1	10 → 13	10-11-12-13	TCOVER13						
	R2	11 → 13	11-12-13	TCOVER14						
	Is_Complex	7 → 13	7-8-9-10-11-12-13	TCOVER16						
2	Is_Complex	5 → 9	5-8-9	TCOVER11	1	1	1	TRUE	unass.	unass.
	Is_Complex	5 → 13	5-8-9-12-13	TCOVER15						

C.2.4.7.3 All-uses testing coverage

Using the definitions provided in 6.3.7.4 and the test coverage items derived above:

$$C_{\text{all-uses}} = \frac{16}{16} \times 100\,\% = 100\,\%$$

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

C.2.4.8 All-du-paths testing

C.2.4.8.1 Identify test coverage items for all-du-paths testing (TD2)

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. At a first glance, it appears as if there are 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. However, both of these sub-paths are infeasible (and so no test cases can be generated to exercise them). Thus, they are not considered to be “simple sub-paths” for all-du-paths testing. See Table C.17.

Table C.17 — All-du-paths testing

Test coverage items	Variables	d-u pair	Sub-path
TCOVER1	A	0 → 1	0-1
TCOVER2	B	0 → 1	0-1
TCOVER3	C	0 → 1	0-1
TCOVER4	A	0 → 10	0-1-2-3-4-6-7-8-9-10

Table C.17 *(continued)*

Test coverage items	Variables	d-u pair	Sub-path
TCOVER5	B	0 → 10	0-1-2-3-4-6-7-8-9-10
TCOVER6	A	0 → 11	0-1-2-3-4-6-7-8-9-10-11
TCOVER7	B	0 → 11	0-1-2-3-4-6-7-8-9-10-11
TCOVER8	Discrim	1 → 4	1-2-3-4
TCOVER9	Discrim	1 → 10	1-2-3-4-6-7-8-9-10
TCOVER10	Discrim	1 → 11	1-2-3-4-6-7-8-9-10-11
TCOVER11	Is_Complex	5 → 9	5-8-9
TCOVER12	Is_Complex	7 → 9	7-8-9
TCOVER13	R1	10 → 13	10-11-12-13
TCOVER14	R2	11 → 13	11-12-13
TCOVER15	Is_Complex	5 → 13	5-8-9-12-13
TCOVER16	Is_Complex	7 → 13	7-8-9-10-11-12-13

C.2.4.8.2 Derive test cases for all-du-paths testing (TD3)

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. See Table C.18.

Table C.18 — Test cases for all-du-paths testing

Test case	All-du-paths				Inputs			Expected result		
	Variables	d-u pair	Sub-paths	Test coverage items	A	B	C	Is_Complex	R1	R2
1	A, B, C	0 → 1	0-1	TCOVER1, TCOVER2, TCOVER3	1	2	1	FALSE	-1	-1
	A, B	0 → 10	0-1-2-3-4-6-7-8-9-10	TCOVER4, TCOVER5						
		0 → 11	0-1-2-3-4-6-7-8-9-10-11	TCOVER6, TCOVER7						
	Discrim	1 → 4	1-2-3-4	TCOVER8						
		1 → 10	1-2-3-4-6-7-8-9-10	TCOVER9						
	Is_Complex	1 → 11	1-2-3-4-6-7-8-9-10-11	TCOVER10						
		7 → 9	7-8-9	TCOVER12						
	R1	10 → 13	10-11-12-13	TCOVER13						
	R2	11 → 13	11-12-13	TCOVER14						
	Is_Complex	7 → 13	7-8-9-10-11-12-13	TCOVER16						
2	Is_Complex	5 → 9	5-8-9	TCOVER11	1	1	1	TRUE	unass.	unass.
	Is_Complex	5 → 13	5-8-9-12-13	TCOVER15						

C.2.4.8.3 All-du-paths testing coverage

Using the definitions provided in 6.3.7.5 and the test coverage items derived above:

$$C_{\text{all-du-paths}} = \frac{16}{16} \times 100\,\% = 100\,\%$$

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

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 5.4 and 6.4. 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/IEEE 29119-2.

D.2 Experience-based test design technique examples

D.2.1 Error guessing

D.2.1.1 General

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 Test basis

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 an exam mark (out of 75) and a coursework (c/w) mark (out of 25) as input, from which it outputs a grade for the course in the range 'A' to 'D'. The grade is generated by calculating 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 invalid input(s) are detected (e.g. a mark is outside its expected range) then a fault message ('FM') is generated. All inputs are passed as integers.

D.2.1.3 Create test model (TD1)

The test model is created 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.

The following checklist is created:

a) Program cannot handle NULL exam marks.

b) Program cannot handle exam marks of 0.

c) Program cannot handle exam marks that are negative.

d) Program cannot handle exam and coursework marks in reverse order.

e) Program cannot handle very large number (e.g. 10 digits).

f) Program cannot handle very large string of alphabetic characters (e.g. 10 alphas).

D.2.1.4 Identify test coverage items (TD2)

The test coverage items are the following generic defects derived from the checklist:

TCOVER1:	enter NULL
TCOVER2:	enter 0
TCOVER3:	enter negative number
TCOVER4:	enter inputs in reverse order
TCOVER5:	enter very large number
TCOVER6	enter very large string of alphabetic characters

D.2.1.5 Derive test cases (TD3)

Test cases can now be derived by selecting a defect type for inclusion in the current test case, identifying inputs to cover 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 test cases shown in Tables D.1 to D.4.

Table D.1 — 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 coverage item	TCOVER1 exam mark	TCOVER1 c/w mark	TCOVER1 exam & c/w marks	TCOVER2 exam mark
Expected output	‘FM’	‘FM’	‘FM’	‘FM’

Table D.2 — 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 coverage item	TCOVER2 c/w mark	TCOVER2 exam & c/w marks	TCOVER3 exam mark	TCOVER3 c/w mark
Expected output	‘FM’	‘FM’	‘FM’	‘FM’

Table D.3 — 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 coverage item	TCOVER3 exam & c/w mark	TCOVER4 exam & c/w mark	TCOVER5 exam mark	TCOVER5 c/w mark
Expected output	‘FM’	‘FM’	‘FM’	‘FM’

Table D.4 — 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 coverage item	TCOVER5 exam & c/w mark	TCOVER6 exam mark	TCOVER6 c/w mark	TCOVER6 exam & c/w mark
Expected output	‘FM’	‘FM’	‘FM’	‘FM’

D.2.1.6 Error guessing test coverage

As stated in 6.4.1, there is no approach for calculating coverage of test coverage items for error guessing.

Annex E
(informative)

Guidelines and examples for the application of grey-box test design techniques

E.1 Guidelines and examples for grey-box test design techniques

E.1.1 Overview

Although the techniques presented in this document are classified as structure-based, specification-based or experience-based, in practice some techniques can be used interchangeably and are known as grey-box techniques (see 5.1). This is demonstrated in the following example, which illustrates how branch testing (which is commonly referred to as a structure-based technique) can be applied for specification-based testing.

E.1.2 Branch testing as a specification-based technique

E.1.2.1 Test basis

Consider the following example specification, which defines a login function that takes a username and password as input to determine whether the user is valid:

> The component shall ask for a username and a password. The user must enter a correct username followed by a corresponding password to be logged into the system. The user is given three attempts each and 20 seconds each to enter the username and password. If the user does not enter both the username and the password within 3 tries each and within 20 seconds each, the system will be locked and disallow any further attempts to login.

The control flow graph for this component is shown in Figure E.1.

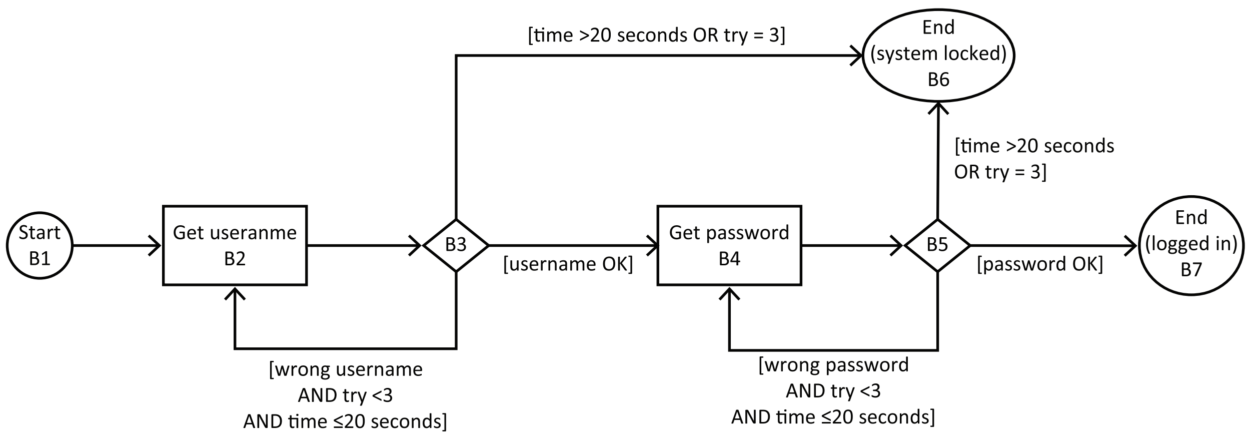

[time >20 seconds OR try = 3]
End (system locked) B6
[time >20 seconds OR try = 3]
Start B1
Get useranme B2
B3
[username OK]
Get password B4
B5
[password OK]
End (logged in) B7
[wrong username AND try <3 AND time ≤20 seconds]
[wrong password AND try <3 AND time ≤20 seconds]

Figure E.1 — Control flow graph for a login function

E.1.2.2 Create test model (TD1)

The control flow graph provides a good overview of the functionality and provides the test model for branch testing. The square blocks in the control flow graph represent segments of program source code in the component, while the diamonds represent decisions. Arrows out of each diamond represent decision outcomes.

E.1.2.3 Identify test coverage items (TD2)

For branch coverage, the test coverage items are the branches in the control flow graph. In this example there are nine test coverage items for branch coverage, as follows:

TCOVER1:	B1 → B2
TCOVER2:	B2 → B3
TCOVER3:	B3 → B2
TCOVER4:	B3 → B4
TCOVER5:	B3 → B6
TCOVER6:	B4 → B5
TCOVER7:	B5 → B4
TCOVER8:	B5 → B6
TCOVER9:	B5 → B7

E.1.2.4 Derive test cases (TD3)

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. For branch coverage, any individual test case will exercise one sub-path and hence potentially many decisions and branches.

Consider a test case that executes the sub-path B1→B2→B3→B4→B5→B7. This case arises when a valid username and matching password are each provided on the first attempt. The path executes 5 out of the 9 branches, giving 56 % coverage (which is not the same as coverage for the decisions).

Now consider a test case which executes the sub-path B1→B2→B3→B2→B3→B4→B5→B4→B5→B6. This sub-path arises when an invalid username and an incorrect password are provided, and the correct password is not provided within the 20 second time limit. Now all branches have been covered except for B3→B6. This sub-path can be covered if, for example, and invalid username is provided three times in a row or waiting too long to enter a valid one. Then all branches have been covered, including all decisions. Note that some conditions within decisions have not been covered.

Test cases to cover each control flow sub-path are shown in Table E.1.

Table E.1 — Test cases for login function

Test case	Inputs				Sub-path	Test coverage items	Expected result
	Username	Username wait time	Password	Password wait time			
1	Andy	≤ 20	Warhol	≤ 20	B1→B2→B3→B4→B5→B7	TCOVER1, TCOVER2, TCOVER4, TCOVER6, TCOVER9	Logged in

Table E.1 *(continued)*

Test case	Inputs				Sub-path	Test coverage items	Expected result
	Username	Username wait time	Password	Password wait time			
2	InVaLiD Andy	≤ 20 ≤ 20	- InVAliD Warhol	- ≤ 20 ≥ 21	B1→B2→B3→B2→B3→B4→B5→B4→B5→B6	TCOVER1, TCOVER2, TCOVER3, TCOVER4, TCOVER6, TCOVER7, TCOVER8	System locked
3	Brandy	≥ 21		-	B1→B2→B3→B6	TCOVER1, TCOVER2, TCOVER5	System locked

E.1.2.5 Branch testing coverage

Using the definitions provided in 6.3.2 and the test coverage items derived above:

$$C_{\text{branch}} = \frac{9}{9} \times 100\,\% = 100\,\%$$

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

Annex F
(informative)

Test design technique effectiveness

Up to this point this document 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 recognized that they take no account of cost.

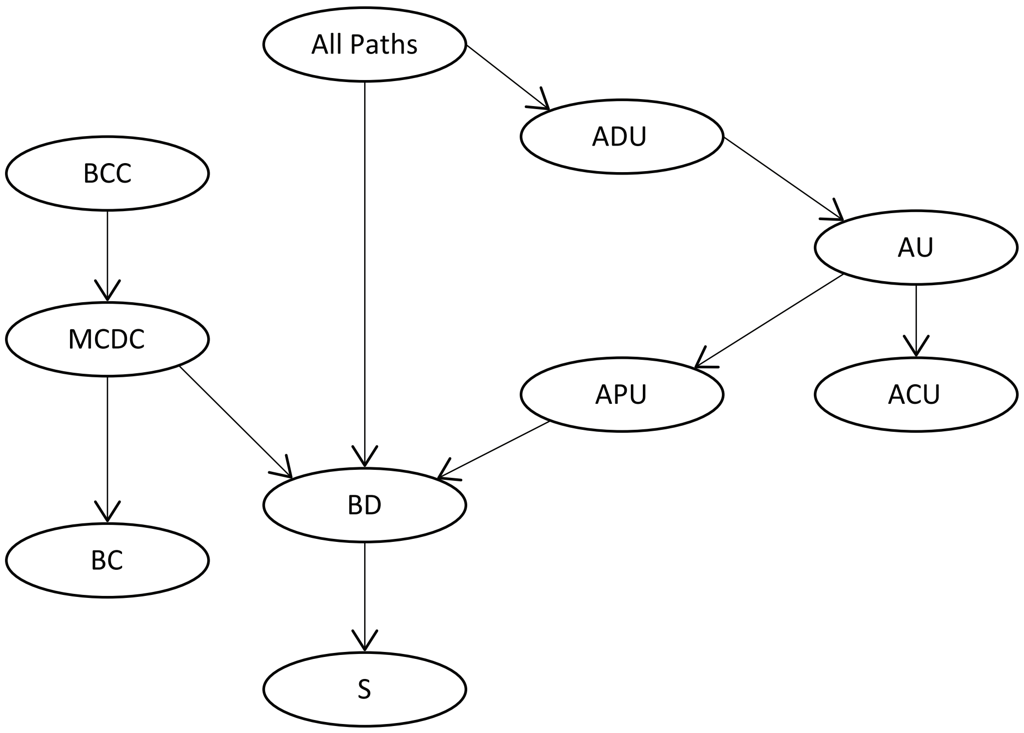
There is no requirement to choose corresponding test case design and test coverage measurement approaches. Specification-based test design techniques are effective at detecting errors of omission, while structure-based test design techniques can only detect errors of commission. So, a test plan can typically 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 can, 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 F.1, where an arrow from one criterion to another indicates that the first criterion subsumes the second. Where a test coverage criterion does not appear in the partial orderings then it is not related to any other criterion by the subsumes relation.



Key

AU	all-uses coverage
ACU	all c-uses coverage
ADU	all du-paths coverage
APU	all p-uses coverage
BD	branch/decision coverage
BC	branch condition coverage
BCC	branch condition combination coverage
MCDC	modified condition/decision coverage
S	statement coverage

Figure F.1 — Partial ordering of structural test coverage criteria (Reid 1996[22])

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

— First, 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.

— Second, 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.

— Third, 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 faults (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 criterion but exercising this subset with more test cases.

The subsumes relation is highly dependent on the definition of full coverage for a criterion and although Figure F.1 is correct for the definitions in Clause 5 it may not apply to alternative definitions used elsewhere.

Annex G
(informative)

ISO/IEC/IEEE 29119-4 and BS 7925-2 test design technique alignment

Table G.1 describes the alignment of the test design techniques in this document and BS 7925-2.

Table G.1 — BS 7925-2:1998 to ISO/IEC/IEEE 29119-4 test design technique mapping

BS 7925-2:1998		ISO/IEC/IEEE 29119-4	
Test design techniques			
3.1	Equivalence Partitioning	5.2.1	Equivalence partitioning
3.2	Boundary Value Analysis	5.2.3	Boundary value analysis
3.3	State Transition Testing	5.2.8	State transition testing
3.4	Cause-Effect Graphing	5.2.7	Cause-effect graphing
3.5	Syntax Testing	5.2.4	Syntax testing
3.6	Statement Testing	5.3.1	Statement testing
3.7	Branch/Decision Testing	5.3.2	Branch testing
		5.3.3	Decision testing
3.8	Data Flow Testing	5.3.7	Data flow testing
3.9	Branch Condition Testing	5.3.4	Branch condition testing
3.10	Branch Condition Combination Testing	5.3.5	Branch condition combination testing
3.11	Modified Condition/Decision Testing	5.3.6	Modified condition/decision coverage (MCDC) testing
3.13	Random Testing	5.2.10	Random testing
Test measurement techniques			
3.1	Equivalence Partition Coverage	6.2.1	Equivalence partition coverage
3.2	Boundary Value Analysis Coverage	6.2.3	Boundary value analysis coverage
3.3	State Transition Coverage	6.2.8	State transition testing coverage
3.4	Cause-Effect Coverage	6.2.7	Cause-effect graphing coverage
3.5	Syntax Coverage	6.2.4	Syntax testing coverage
3.6	Statement Coverage	6.3.1	Statement testing coverage
3.7	Branch and Decision Coverage	6.3.2	Branch testing coverage
		6.3.3	Decision testing coverage
3.8	Data Flow Coverage	6.3.7	Data flow testing coverage
3.9	Branch Condition Coverage	6.3.4	Branch condition testing coverage
3.10	Branch Condition Combination Coverage	6.3.5	Branch condition combination testing coverage
3.11	Modified Condition/Decision Coverage	6.3.6	Modified condition/decision coverage (MCDC)
3.13	Random Testing	6.2.10	Random testing coverage
Guidelines for the Application of Test Design & Measurement Techniques			
B.1	Equivalence Partitioning	B.2.1	Equivalence partitioning
B.2	Boundary Value Analysis	B.2.3	Boundary value analysis

Table G.1 *(continued)*

BS 7925-2:1998		ISO/IEC/IEEE 29119-4	
B.3	State Transition Testing	B.2.8	State transition testing
B.4	Cause-Effect Graphing	B.2.7	Cause-effect graphing
B.5	Syntax Testing	B.2.4	Syntax testing
B.6	Statement Testing	C.2.1	Statement testing
B.7	Branch/Decision Testing	C.2.2	Branch/decision testing
B.8	Data Flow Testing	C.2.4	Data flow testing
B.9	Branch Condition Testing	C.2.3	Branch condition testing, branch condition combination testing and modified condition/decision coverage (MCDC) testing
B.10	Branch Condition Combination Testing		
B.11	Modified Condition/Decision Testing		
B.13	Random Testing	B.2.10	Random testing
Test Technique Effectiveness			
Annex C	Test Technique Effectiveness	Annex F	Test design technique coverage effectiveness

Annex H
(informative)

Test models

In the previous edition of this document, the definitions of test techniques were based on a previous edition of the test design and implementation process from ISO/IEC/IEEE 29119-2, which included the use of test conditions. Feedback on use of the previous edition highlighted a problem with users' understanding of test conditions and their use for deriving test cases – many users described them as 'confusing'. An alternative approach based on test models was chosen instead.

Test models have been successfully used to define a new version of the test design and implementation process from ISO/IEC/IEEE 29119-2. Each of the test case design techniques in this document and corresponding examples have been rewritten using test models.

The use of test models allows the introduction of a simplified test design process, where test coverage items are more easily identified. This simplified test design process is comprised of four activities rather than the previous six, where each activity can be seen to contribute to the overall process. In the previous edition of ISO/IEC/IEEE 29119-2, for some test case design techniques, the two activities of deriving test conditions and deriving test coverage items resulted in the same output, suggesting that one of the two activities was superfluous. By using a test model (instead of test conditions), there is no one-to-one mapping between the test model and test coverage items, which means each activity is performing a valid transformation.

The difference between a test condition and a test model can be challenging to grasp. A test condition is a somewhat more abstract concept that is used at various levels of detail to describe anything that may be used as the basis for testing. At a high level, a test condition can be used to agree what user features are to be covered by tests with a customer or, at a low level, a test condition can describe a line of code that will be executed by a test by a developer. In contrast, the test model is more focused, describing the behaviour of that part of the test item that is being tested in terms of required test coverage. The test model can cover the same contents as one or more test conditions (and will typically replace several) when test conditions relate closely to test coverage items. The test model is created to provide a means of representing the behaviour of a test item that explicitly shows the test coverage items to be covered by tests, whereas test conditions are often less directly linked to test coverage items. Another obvious difference is that when tests are created to satisfy a test objective, a single cohesive test model can normally be used to create many tests (often all the tests for a given test design technique), while to create those same tests, many, less cohesive, test conditions would typically have to be identified.

Bibliography

[1] ISO 26262 (all parts), *Road vehicles — Functional safety*

[2] Beizer B., Black Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons Inc, 1995

[3] Non-Functional Testing, British Computer Society Special Interest Group in Software Testing, [viewed 11 December 2020]. Available from http://www.testingstandards.co.uk/non_functional_testing_techniques.htm

[4] BS 7925-2[1)], *Software testing – Software component testing*

[5] Burnstein I., Practical Software Testing: A Process-Oriented Approach. Springer-Verlag, 2003

[6] Cho C.K., Quality Programming. Wiley, 1987

[7] Chow T.S., Testing Software Design Modelled by Finite-State Machines. In: IEEE Transactions on Software Engineering, Vol. **SE-4**(3), 1978

[8] Craig R., Jaskiel S., Systematic Software Testing. Artech House Inc, 2002

[9] Desikan S., Ramesh G., Software Testing: Principles and Practices. Pearson Education, 2007

[10] Grindal M., Offutt J., Andler S., 2005. Combination Testing Strategies: A Survey. In: Software Testing, Verification and Reliability, John Wiley & Sons Ltd., 15, pp. 167-199.

[11] Grochtmann M., Grimm K., Classification Trees for Partition Testing. In: Software Testing, Verification & Reliability, Wiley, 3(2), pp. 63–82, 1993

[12] Jonassen Hass., A.M., Guide to Advanced Software Testing. Artech House, 2008

[13] ISO 9241-11:2018, *Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts*

[14] ISO/IEC 25010, *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*

[15] ISO/IEC 25030, *Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — Quality requirements framework*

[16] Kaner C., Testing Computer Software. TAB Books Inc, 1998.

[17] Kernighan B.W., Richie D.M., The C Programming Language. Prentice-Hall Software Series, 1998.

[18] Mandl R., Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing. Commun. ACM. 1985, **28** (10) pp. 1054–1058

[19] Myers G., The Art of Software Testing. John Wiley & Sons Inc, 1979

[20] Nursimulu K., Probert R.L., 1995. Cause-Effect Graphing Analysis and Validation of Requirements. In Proceedings of CASCON'1995.

[21] RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification. RTCA, Inc. 2011.

[22] Reid S., Popular Misconceptions in Module Testing. In Proceedings of the Software Testing Conference (STC), Washington DC, 1996.

1) Available from: https://shop.bsigroup.com/, British Standards Institute, 1998.

[23] CHEN et al, , Metamorphic Testing: A Review of Challenges and Opportunities, ACM Comput. Surv. **51**, 1, Article 4, January 2018.

[24] PINKSTER I., VAN DE BURGT D., JANSSEN D., Successful Test Management, Springer, 2006.

IEEE Notices and Abstract

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that the presenter's views should be considered the personal views of that individual rather than the formal position of IEEE, IEEE SA, the Standards Committee, or the Working Group.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE or IEEE SA. However, **IEEE does not provide interpretations, consulting information, or advice pertaining to IEEE Standards documents**.

Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its Societies and Standards Coordinating Committees are not able to provide an instant response to comments, or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in evaluating comments or in revisions to an IEEE standard is welcome to join the relevant IEEE working group. You can indicate interest in a working group using the Interests tab in the Manage Profile & Interests area of the IEEE SA myProject system. An IEEE Account is needed to access the application.

Comments on standards should be submitted using the Contact Us form.

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not constitute compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Data privacy

Users of IEEE Standards documents should evaluate the standards for considerations of data privacy and data ownership in the context of assessing and using the standards in compliance with applicable laws and regulations.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under US and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate licensing fees, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400; https://www.copyright.com/. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Abstract

The purpose of the ISO/IEC/IEEE 29119 series of software testing standards is to define an internationally agreed set of standards for software testing that can be used by any organization when performing any form of software testing. ISO/IEC/IEEE 29119-4 defines a variety of the most common test 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/IEEE 29119-2.

The test techniques presented in this document 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 would be used to determine the set of techniques that are applicable in specific situations (risk-based testing is covered in ISO/IEC/IEEE 29119-1 and ISO/IEC/IEEE 29119-2).

ICS 35.080

ISBN 978-1-5044-7982-0 STD24958 (PDF); 978-1-5044-7983-7 STDPD24958 (Print)

Price based on 135 pages