ESA PSS-05-10 Issue 1 Revision 1
March 1995

Guide to software verification and validation

Prepared by:
ESA Board for Software
Standardisation and Control
(BSSC)

Approved by:

The Inspector General, ESA

european space agency / agence spatiale européenne
8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: **ESA PSS-05-10 Guide to Software Verification and Validation**			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	1994	First issue
1	1	1995	Minor updates for publication

Issue 1 Revision 1 approved, May 1995
Board for Software Standardisation and Control
M. Jones and U. Mortensen, co-chairmen

Issue 1 approved by:
The Inspector General, ESA

Published by ESA Publications Division,
ESTEC, Noordwijk, The Netherlands.
Printed in the Netherlands.
ESA Price code: E2
ISSN 0379-4059

TABLE OF CONTENTS

PREFACE

This document is one of a series of guides to software engineering produced by the Board for Software Standardisation and Control (BSSC), of the European Space Agency. The guides contain advisory material for software developers conforming to ESA's Software Engineering Standards, ESA PSS-05-0. They have been compiled from discussions with software engineers, research of the software engineering literature, and experience gained from the application of the Software Engineering Standards in projects.

Levels one and two of the document tree at the time of writing are shown in Figure 1. This guide, identified by the shaded box, provides guidance about implementing the mandatory requirements for software verification and validation described in the top level document ESA PSS-05-0.

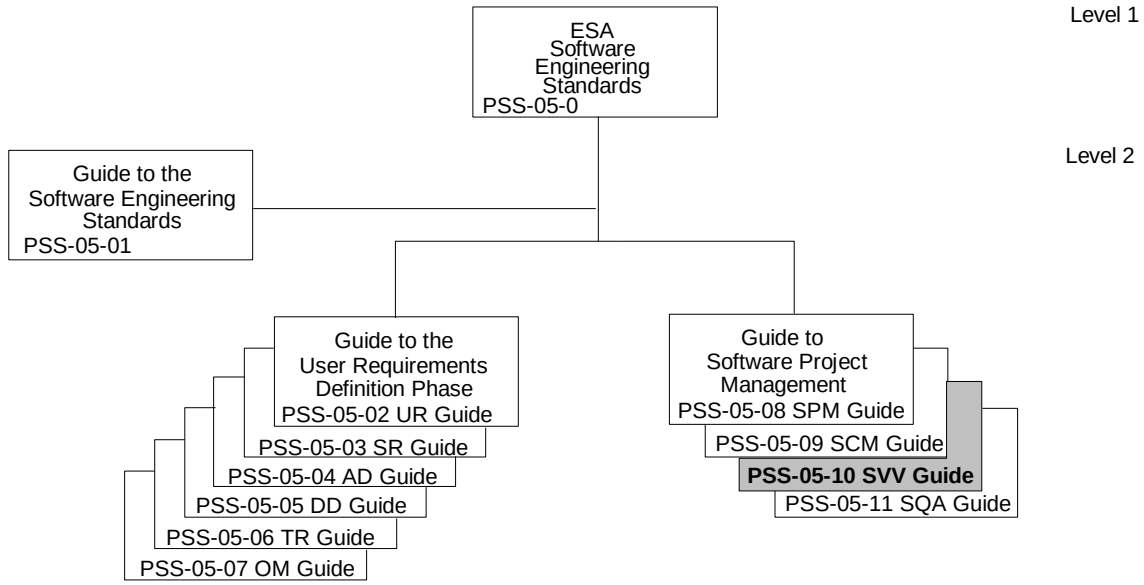

Figure 1: ESA PSS-05-0 document tree

The Guide to the Software Engineering Standards, ESA PSS-05-01, contains further information about the document tree. The interested reader should consult this guide for current information about the ESA PSS-05-0 standards and guides.

The following past and present BSSC members have contributed to the production of this guide: Carlo Mazza (chairman), Gianfranco Alvisi, Michael Jones, Bryan Melton, Daniel de Pablo and Adriaan Scheffer.

The BSSC wishes to thank Jon Fairclough for his assistance in the development of the Standards and Guides, and to all those software engineers in ESA and Industry who have made contributions.

Requests for clarifications, change proposals or any other comment concerning this guide should be addressed to:

BSSC/ESOC Secretariat
Attention of Mr C Mazza
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr B Melton
ESTEC
Postbus 299
NL-2200 AG Noordwijk
The Netherlands

CHAPTER 1
INTRODUCTION

1.1 PURPOSE

ESA PSS-05-0 describes the software engineering standards to be applied for all deliverable software implemented for the European Space Agency (ESA), either in-house or by industry [Ref 1].

ESA PSS-05-0 requires that software be verified during every phase of its development life cycle and validated when it is transferred. These activities are called 'Software Verification and Validation' (SVV). Each project must define its Software Verification and Validation activities in a Software Verification and Validation Plan (SVVP).

This guide defines and explains what software verification and validation is, provides guidelines on how to do it, and defines in detail what a Software Verification and Validation Plan should contain.

This guide should be read by everyone concerned with developing software, such as software project managers, software engineers and software quality assurance staff. Sections on acceptance testing and formal reviews should be of interest to users.

1.2 OVERVIEW

Chapter 2 contains a general discussion of the principles of software verification and validation, expanding upon the ideas in ESA PSS-05-0. Chapter 3 discusses methods for software verification and validation that can be used to supplement the basic methods described in Chapter 2. Chapter 4 discusses tools for software verification and validation. Chapter 5 describes how to write the SVVP.

All the mandatory practices in ESA PSS-05-0 concerning software verification and validation are repeated in this document. The identifier of the practice is added in parentheses to mark a repetition. This document contains no new mandatory practices.

1.3 IEEE STANDARDS USED FOR THIS GUIDE

Six standards of the Institute of Electrical and Electronics Engineers (IEEE) have been used to ensure that this guide complies as far as possible with internationally accepted standards for verification and validation terminology and documentation. The IEEE standards are listed in Table 1.3 below.

Reference	Title
610.12-1990	Standard Glossary of Software Engineering Terminology
829-1983	Standard for Software Test Documentation
1008-1987	Standard for Software Unit Testing
1012-1986	Standard for Software Verification and Validation Plans
1028-1988	Standard for Software Reviews and Audits

Table 1.3: IEEE Standards used for this guide.

IEEE Standard 829-1983 was used to define the table of contents for the SVVP sections that document the unit, integration, system and acceptance testing activities (i.e. SVVP/UT, SVVP/IT, SVVP/ST, SVVP/AT).

IEEE Standard 1008-1987 provides a detailed specification of the unit testing process. Readers who require further information on the unit testing should consult this standard.

IEEE Standard 1012-1986 was used to define the table of contents for the SVVP sections that document the non-testing verification and validation activities (i.e. SVVP/SR, SVVP/AD, SVVP/DD).

IEEE Standard 1028-1988 was used to define the technical review, walkthrough, inspection and audit processes.

Because of the need to integrate the requirements of six standards into a single approach to software verification and validation, users of this guide should not claim complete compliance with any one of the IEEE standards.

CHAPTER 2
SOFTWARE VERIFICATION AND VALIDATION

2.1 INTRODUCTION

Software verification and validation activities check the software against its specifications. Every project must verify and validate the software it produces. This is done by:

- checking that each software item meets specified requirements;
- checking each software item before it is used as an input to another activity;
- ensuring that checks on each software item are done, as far as possible, by someone other than the author;
- ensuring that the amount of verification and validation effort is adequate to show each software item is suitable for operational use.

Project management is responsible for organising software verification and validation activities, the definition of software verification and validation roles (e.g. review team leaders), and the allocation of staff to those roles.

Whatever the size of project, software verification and validation greatly affects software quality. People are not infallible, and software that has not been verified has little chance of working. Typically, 20 to 50 errors per 1000 lines of code are found during development, and 1.5 to 4 per 1000 lines of code remain even after system testing [Ref 20]. Each of these errors could lead to an operational failure or non-compliance with a requirement. The objective of software verification and validation is to reduce software errors to an acceptable level. The effort needed can range from 30% to 90% of the total project resources, depending upon the criticality and complexity of the software [Ref 12].

This chapter summarises the principles of software verification and validation described in ESA PSS-05-0 and then discusses the application of these principles first to documents and then to code.

2.2 PRINCIPLES OF SOFTWARE VERIFICATION AND VALIDATION

Verification can mean the:

- act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services or documents conform to specified requirements [Ref.5];
- process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase [Ref. 6]
- formal proof of program correctness [Ref.6].

The first definition of verification in the list above is the most general and includes the other two. In ESA PSS-05-0, the first definition applies.

Validation is, according to its ANSI/IEEE definition, 'the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements'. Validation is, therefore, 'end-to-end' verification.

Verification activities include:

- technical reviews, walkthroughs and software inspections;
- checking that software requirements are traceable to user requirements;
- checking that design components are traceable to software requirements;
- unit testing;
- integration testing;
- system testing;
- acceptance testing;
- audit.

Verification activities may include carrying out formal proofs.

The activities to be conducted in a project are described in the Software Verification and Validation Plan (SVVP).

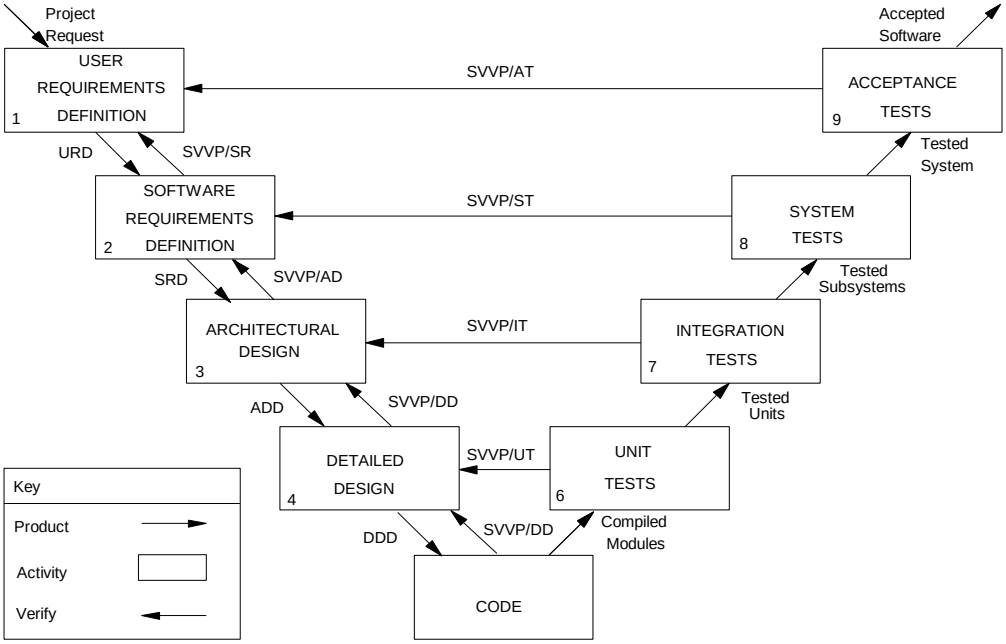

Figure 2.2: Life cycle verification approach

Figure 2.2 shows the life cycle verification approach. Software development starts in the top left-hand corner, progresses down the left-hand 'specification' side to the bottom of the 'V' and then onwards up the right-hand 'production' side. The V-formation emphasises the need to verify each output specification against its input specification, and the need to verify the software at each stage of production against its corresponding specification.

In particular the:

- SRD must be verified with respect to the URD by means of the SVVP/SR;
- ADD must be verified with respect to the SRD by means of the SVVP/AD;
- DDD must be verified with respect to the ADD by means of the SVVP/DD;
- code must be verified with respect to the DDD by means of the SVVP/DD;
- unit tests verify that the software subsystems and components work correctly in isolation, and as specified in the detailed design, by means of the SVVP/UT;

- integration tests verify that the major software components work correctly with the rest of the system, and as specified in the architectural design, by means of the SVVP/IT;
- system tests verify that the software system meets the software requirements, by means of the SVVP/ST;
- acceptance tests verify that the software system meets the user requirements, by means of the SVVP/AT.

These verification activities demonstrate compliance to specifications. This may be done by showing that the product:

- performs as specified;
- contains no defects that prevent it performing as specified.

Demonstration that a product meets its specification is a mechanical activity that is driven by the specification. This part of verification is efficient for demonstrating conformance to functional requirements (e.g. to validate that the system has a function it is only necessary to exercise the function). In contrast, demonstration that a product contains no defects that prevent it from meeting its specification requires expert knowledge of what the system must do, and the technology the system uses. This expertise is needed if the non-functional requirements (e.g. those for reliability) are to be met. Skill and ingenuity are needed to show up defects.

In summary, software verification and validation should show that the product conforms to all the requirements. Users will have more confidence in a product that has been through a rigorous verification programme than one subjected to minimal examination and testing before release.

2.3 REVIEWS

A review is 'a process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval' [Ref 6].

Reviews may be formal or informal. Formal reviews have explicit and definite rules of procedure. Informal reviews have no predefined procedures. Although informal reviews can be very useful for educating project members and solving problems, this section is only concerned with reviews that have set procedures, i.e. formal reviews.

Three kinds of formal review are normally used for software verification:

- technical review;
- walkthrough;
- audits.

These reviews are all 'formal reviews' in the sense that all have specific objectives and procedures. They seek to identify defects and discrepancies of the software against specifications, plans and standards.

Software inspections are a more rigorous alternative to walkthroughs, and are strongly recommended for software with stringent reliability, security and safety requirements. Methods for software inspections are described in Section 3.2.

The software problem reporting procedure and document change procedure defined in Part 2, Section 3.2.3.2 of ESA PSS-05-0, and in more detail ESA PSS-05-09 'Guide to Software Configuration Management', calls for a formal review process for all changes to code and documentation. Any of the first two kinds of formal review procedure can be applied for change control. The SRB, for example, may choose to hold a technical review or walkthrough as necessary.

2.3.1 Technical reviews

Technical reviews evaluate specific software elements to verify progress against the plan. The technical review process should be used for the UR/R, SR/R, AD/R, DD/R and any critical design reviews.

The UR/R, SR/R, AD/R and DD/R are formal reviews held at the end of a phase to evaluate the products of the phase, and to decide whether the next phase may be started (UR08, SR09, AD16 and DD11).

Critical design reviews are held in the DD phase to review the detailed design of a major component to certify its readiness for implementation (DD10).

The following sections describe the technical review process. This process is based upon the ANSI/IEEE Std 1028-1988, 'IEEE Standard for Software Reviews and Audits' [Ref 10], and Agency best practice.

2.3.1.1 Objectives

The objective of a technical review is to evaluate a specific set of review items (e.g. document, source module) and provide management with evidence that:

- they conform to specifications made in previous phases;
- they have been produced according to the project standards and procedures;
- any changes have been properly implemented, and affect only those systems identified by the change specification (described in a RID, DCR or SCR).

2.3.1.2 Organisation

The technical review process is carried out by a review team, which is made up of:

- a leader;
- a secretary;
- members.

In large and/or critical projects, the review team may be split into a review board and a technical panel. The technical panel is usually responsible for processing RIDs and the technical assessment of review items, producing as output a technical panel report. The review board oversees the review procedures and then independently assesses the status of the review items based upon the technical panel report.

The review team members should have skills to cover all aspects of the review items. Depending upon the phase, the review team may be drawn from:

- users;
- software project managers;
- software engineers;
- software librarians;
- software quality assurance staff;
- independent software verification and validation staff;
- independent experts not involved in the software development.

Some continuity of representation should be provided to ensure consistency.

The leader's responsibilities include:

- nominating the review team;
- organising the review and informing all participants of its date, place and agenda;
- distribution of the review items to all participants before the meeting;
- organising as necessary the work of the review team;
- chairing the review meetings;
- issuing the technical review report.

The secretary will assist the leader as necessary and will be responsible for documenting the findings, decisions and recommendations of the review team.

Team members examine the review items and attend review meetings. If the review items are large, complex, or require a range of specialist skills for effective review, the leader may share the review items among members.

2.3.1.3 Input

Input to the technical review process includes as appropriate:

- a review meeting agenda;
- a statement of objectives;
- the review items;
- specifications for the review items;
- plans, standards and guidelines that apply to the review items;
- RID, SPR and SCR forms concerning the review items;
- marked up copies of the review items;
- reports of software quality assurance staff.

2.3.1.4 Activities

The technical review process consists of the following activities:

- preparation;
- review meeting.

The review process may start when the leader considers the review items to be stable and complete. Obvious signs of instability are the presence of TBDs or of changes recommended at an earlier review meeting not yet implemented.

Adequate time should be allowed for the review process. This depends on the size of project. A typical schedule for a large project (20 man years or more) is shown in Table 2.3.1.4.

Event	Time
Review items distributed	R - 20 days
RIDs categorised and distributed	R - 10 days
Review Meeting	R
Issue of Report	R + 20 days

Table 2.3.1.4: Review Schedule for a large project

Members may have to combine their review activities with other commitments, and the review schedule should reflect this.

2.3.1.4.1 Preparation

The leader creates the agenda and distributes it, with the statements of objectives, review items, specifications, plans, standards and guidelines (as appropriate) to the review team.

Members then examine the review items. Each problem is recorded by completing boxes one to six of the RID form. A RID should record only one problem, or group of related problems. Members then pass their RIDs to the secretary, who numbers each RID uniquely and forwards them to the author for comment. Authors add their responses in box seven and then return the RIDs to the secretary.

The leader then categorises each RID as major, minor, or editorial. Major RIDs relate to a problem that would affect capabilities, performance, quality, schedule and cost. Minor RIDs request clarification on specific points and point out inconsistencies. Editorial RIDs point out defects in format, spelling and grammar. Several hundred RIDs can be generated in a large project review, and classification is essential if the RIDs are to be dealt

with efficiently. Failure to categorise the RIDs can result in long meetings that concentrate on minor problems at the expense of major ones.

Finally the secretary sorts the RIDs in order of the position of the discrepancy in the review item. The RIDs are now ready for input to the review meeting.

Preparation for a Software Review Board follows a similar pattern, with RIDs being replaced by SPRs and SCRs.

2.3.1.4.2 Review meeting

A typical review meeting agenda consists of:

1. Introduction;
2. Presentation of the review items;
3. Classification of RIDs;
4. Review of the major RIDs;
5. Review of the other RIDs;
6. Conclusion.

The introduction includes agreeing the agenda, approving the report of any previous meetings and reviewing the status of outstanding actions.

After the preliminaries, authors present an overview of the review items. If this is not the first meeting, emphasis should be given to any changes made since the items were last discussed.

The leader then summarises the classification of the RIDs. Members may request that RIDs be reclassified (e.g. the severity of a RID may be changed from minor to major). RIDs that originate during the meeting should be held over for decision at a later meeting, to allow time for authors to respond.

Major RIDs are then discussed, followed by the minor and editorial RIDs. The outcome of the discussion of any defects should be noted by the secretary in the review decision box of the RID form. This may be one of CLOSE, UPDATE, ACTION or REJECT. The reason for each decision should be recorded. Closure should be associated with the successful completion of an update. The nature of an update should be agreed. Actions should be properly formulated, the person responsible identified, and the completion date specified. Rejection is equivalent to closing a RID with no action or update.

The conclusions of a review meeting should be agreed during the meeting. Typical conclusions are:

- authorisation to proceed to the next phase, subject to updates and actions being completed;
- authorisation to proceed with a restricted part of the system;
- a decision to perform additional work.

One or more of the above may be applicable.

If the review meeting cannot reach a consensus on RID dispositions and conclusions, possible actions are:

- recording a minority opinion in the review report;
- for one or more members to find a solution outside the meeting;
- referring the problem to the next level of management.

2.3.1.5 Output

The output from the review is a technical review report that should contain the following:

- abstract of the report;
- a list of the members;
- an identification of the review items;
- tables of RIDs, SPRs and SCRs organised according to category, with dispositions marked;
- a list of actions, with persons responsible identified and expected dates for completion defined;
- conclusions.

This output can take the form of the minutes of the meeting, or be a self-standing report. If there are several meetings, the collections of minutes can form the report, or the minutes can be appended to a report summarising the findings. The report should be detailed enough for management to judge what happened. If there have been difficulties in reaching consensus during the review, it is advisable that the output be signed off by members.

2.3.2 Walkthroughs

Walkthroughs should be used for the early evaluation of documents, models, designs and code in the SR, AD and DD phases. The following

sections describe the walkthrough process, and are based upon the ANSI/IEEE Std 1028-1988, 'IEEE Standard for Software Reviews and Audits' [Ref 10].

2.3.2.1 Objectives

The objective of a walkthrough is to evaluate a specific software element (e.g. document, source module). A walkthrough should attempt to identify defects and consider possible solutions. In contrast with other forms of review, secondary objectives are to educate, and to resolve stylistic problems.

2.3.2.2 Organisation

The walkthrough process is carried out by a walkthrough team, which is made up of:

- a leader;
- a secretary;
- the author (or authors);
- members.

The leader, helped by the secretary, is responsible for management tasks associated with the walkthrough. The specific responsibilities of the leader include:

- nominating the walkthrough team;
- organising the walkthrough and informing all participants of the date, place and agenda of walkthrough meetings;
- distribution of the review items to all participants before walkthrough meetings;
- organising as necessary the work of the walkthrough team;
- chairing the walkthrough meeting;
- issuing the walkthrough report.

The author is responsible for the production of the review items, and for presenting them at the walkthrough meeting.

Members examine review items, report errors and recommend solutions.

2.3.2.3 Input

Input to the walkthrough consists of:

- a statement of objectives in the form of an agenda;
- the review items;
- standards that apply to the review items;
- specifications that apply to the review items.

2.3.2.4 Activities

The walkthrough process consists of the following activities:

- preparation;
- review meeting.

2.3.2.4.1 Preparation

The moderator or author distributes the review items when the author decides that they are ready for walkthrough. Members should examine the review items prior to the meeting. Concerns should be noted on RID forms so that they can be raised at the appropriate point in the walkthrough meeting.

2.3.2.4.2 Review meeting

The review meeting begins with a discussion of the agenda and the report of the previous meeting. The author then provides an overview of the review items.

A general discussion follows, during which issues of the structure, function and scope of the review items should be raised.

The author then steps through the review items, such as documents and source modules (in contrast technical reviews step through RIDs, not the items themselves). Members raise issues about specific points as they are reached in the walkthrough.

As the walkthrough proceeds, errors, suggested changes and improvements are noted on RID forms by the secretary.

2.3.2.5 Output

The output from the walkthrough is a walkthrough report that should contain the following:

- a list of the members;
- an identification of the review items;
- a list of changes and defects noted during the walkthrough;
- completed RID forms;
- a list of actions, with persons responsible identified and expected dates for completion defined;
- recommendations made by the walkthrough team on how to remedy defects and dispose of unresolved issues (e.g. further walkthrough meetings).

This output can take the form of the minutes of the meeting, or be a self-standing report.

2.3.3 Audits

Audits are independent reviews that assess compliance with software requirements, specifications, baselines, standards, procedures, instructions, codes and contractual and licensing requirements. To ensure their objectivity, audits should be carried out by people independent of the development team. The audited organisation should make resources (e.g. development team members, office space) available to support the audit.

A 'physical audit' checks that all items identified as part of the configuration are present in the product baseline. A 'functional audit' checks that unit, integration and system tests have been carried out and records their success or failure. Other types of audits may examine any part of the software development process, and take their name from the part of the process being examined, e.g. a 'code audit' checks code against coding standards.

Audits may be routine or non-routine. Examples of routine audits are the functional and physical audits that must be performed before the release of the software (SVV03). Non-routine audits may be initiated by the organisation receiving the software, or management and quality assurance personnel in the organisation producing the software.

The following sections describe the audit process, and are based upon the ANSI/IEEE Std 1028-1988, 'IEEE Standard for Software Reviews and Audits' [Ref 10].

2.3.3.1 Objectives

The objective of an audit is to verify that software products and processes comply with standards, guidelines, specifications and procedures.

2.3.3.2 Organisation

The audit process is carried out by an audit team, which is made up of:

- a leader;
- members.

The leader is responsible for administrative tasks associated with the audit. The specific responsibilities of the leader include:

- nominating the audit team;
- organising the audit and informing all participants of the schedule of activities;
- issuing the audit report.

Members interview the development team, examine review items, report errors and recommend solutions.

2.3.3.3 Input

The following items should be input to an audit:

- terms of reference defining the purpose and scope of the audit;
- criteria for deciding the correctness of products and processes such as contracts, plans, specifications, procedures, guidelines and standards;
- software products;
- software process records;
- management plans defining the organisation of the project being audited.

2.3.3.4 Activities

The team formed to carry out the audit should produce a plan that defines the:

- products or processes to be examined;
- schedule of audit activities;
- sampling criteria, if a statistical approach is being used;
- criteria for judging correctness (e.g. the SCM procedures might be audited against the SCMP);
- checklists defining aspects to be audited;
- audit staffing plan;
- date, time and place of the audit kick-off meeting.

The audit team should prepare for the audit by familiarising themselves with the organisation being audited, its products and its processes. All the team must understand the audit criteria and know how to apply them. Training may be necessary.

The audit team then examines the software products and processes, interviewing project team members as necessary. This is the primary activity in any audit. Project team members should co-operate fully with the auditors. Auditors should fully investigate all problems, document them, and make recommendations about how to rectify them. If the system is very large, the audit team may have to employ a sampling approach.

When their investigations are complete, the audit team should issue a draft report for comment by the audited organisation, so that any misunderstandings can be eliminated. After receiving the audited organisation's comments, the audit team should produce a final report. A follow-up audit may be required to check that actions are implemented.

2.3.3.5 Output

The output from an audit is an audit report that:

- identifies the organisation being audited, the audit team, and the date and place of the audit;
- defines the products and processes being audited;
- defines the scope of the audit, particularly the audit criteria for products and processes being audited;
- states conclusions;
- makes recommendations;
- lists actions.

2.4 TRACING

Tracing is 'the act of establishing a relationship between two or more products of the development process; for example, to establish the relationship between a given requirement and the design element that implements that requirement' [Ref 6]. There are two kinds of traceability:

- forward traceability;
- backward traceability.

Forward traceability requires that each input to a phase must be traceable to an output of that phase (SVV01). Forward traceability shows completeness, and is normally done by constructing traceability matrices. These are normally implemented by tabulating the correspondence between input and output (see the example in ESA PSS-05-03, Guide to Software Requirements Definition [Ref 2]). Missing entries in the matrix display incompleteness quite vividly. Forward traceability can also show duplication. Inputs that trace to more than one output may be a sign of duplication.

Backward traceability requires that each output of a phase must be traceable to an input to that phase (SVV02). Outputs that cannot be traced to inputs are superfluous, unless it is acknowledged that the inputs themselves were incomplete. Backward tracing is normally done by including with each item a statement of why it exists (e.g. source of a software requirement, requirements for a software component).

During the software life cycle it is necessary to trace:

- user requirements to software requirements and vice-versa;
- software requirements to component descriptions and vice versa;
- integration tests to architectural units and vice-versa;
- unit tests to the modules of the detailed design;
- system tests to software requirements and vice-versa;
- acceptance tests to user requirements and vice-versa.

To support traceability, all components and requirements are identified. The SVVP should define how tracing is to be done. References to components and requirements should include identifiers. The SCMP defines the identification conventions for documents and software components. The SVVP should define additional identification conventions to be used within documents (e.g. requirements) and software components.

2.5 FORMAL PROOF

Formal proof attempts to demonstrate logically that software is correct. Whereas a test empirically demonstrates that specific inputs result in specific outputs, formal proofs logically demonstrate that all inputs meeting defined preconditions will result in defined postconditions being met.

Where practical, formal proof of the correctness of software may be attempted. Formal proof techniques are often difficult to justify because of the additional effort required above the necessary verification techniques of reviewing, tracing and testing.

The difficulty of expressing software requirements and designs in the mathematical form necessary for formal proof has prevented the wide application of the technique. Some areas where formal methods have been successful are for the specification and verification of:

- protocols;
- secure systems.

Good protocols and very secure systems depend upon having precise, logical specifications with no loopholes.

Ideally, if formal techniques can prove that software is correct, separate verification (e.g. testing) should not be necessary. However, human errors in proofs are still possible, and ways should be sought to avoid them, for example by ensuring that all proofs are checked independently.

Sections 3.3 and 3.4 discuss Formal Methods and formal Program Verification Techniques.

2.6 TESTING

A test is 'an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component' [Ref 6]. Compared with other verification techniques, testing is the most direct because it executes the software, and is therefore always to be preferred. When parts of a specification cannot be verified by a test, another verification technique (e.g. inspection) should be substituted in the test plan. For example a test of a portability requirement might be to run the software

in the alternative environment. If this not possible, the substitute approach might be to inspect the code for statements that are not portable.

Testing skills are just as important as the ability to program, design and analyse. Good testers find problems quickly. Myers defines testing as 'the process of executing a program with the intent of finding errors' [Ref 14]. While this definition is too narrow for ESA PSS-05-0, it expresses the sceptical, critical attitude required for effective testing.

The testability of software should be evaluated as it is designed, not when coding is complete. Designs should be iterated until they are testable. Complexity is the enemy of testability. When faced with a complex design, developers should ask themselves:

- can the software be simplified without compromising its capabilities?
- are the resources available to test software of this complexity?

Users, managers and developers all need to be assured that the software does what it is supposed to do. An important objective of testing is to show that software meets its specification. The 'V diagram' in Figure 2.2 shows that unit tests compare code with its detailed design, integration tests compare major components with the architectural design, system tests compare the software with the software requirements, and acceptance tests compare the software with the user requirements. All these tests aim to 'verify' the software, i.e. show that it truly conforms to specifications.

In ESA PSS-05-0 test plans are made as soon as the corresponding specifications exist. These plans outline the approach to testing and are essential for estimating the resources required to complete the project. Tests are specified in more detail in the DD phase. Test designs, test cases and test procedures are defined and included in the SVVP. Tests are then executed and results recorded.

Figure 2.6 shows the testing activities common to unit, integration, system and acceptance tests. Input at the top left of the figure are the Software Under Test (SUT), the test plans in the SVVP, and the URD, SRD, ADD and DDD that define the baselines for testing against. This sequence of activities is executed for unit testing, integration testing, system testing and acceptance testing in turn.

The following paragraphs address each activity depicted in Figure 2.6. Section 4.5 discusses the tools needed to support the activities.

1. The 'specify tests' activity takes the test plan in the SVVP, and the product specification in one of the URD, SRD, ADD or DDD and

produces a test design for each requirement or component. Each design will imply a family of test cases. The Software Under Test (SUT) is required for the specification of unit tests.

2. The 'make test software' activity takes the test case specifications and produces the test code (stubs, drivers, simulators, harnesses), input data files and test procedures needed to run the tests.
3. The 'link SUT' activity takes the test code and links it with the SUT, and (optionally) existing tested code, producing the executable SUT.
4. The 'run tests' activity executes the tests according to the test procedures, by means of the input data. The output data produced may include coverage information, performance information, or data produced by the normal functioning of the SUT.

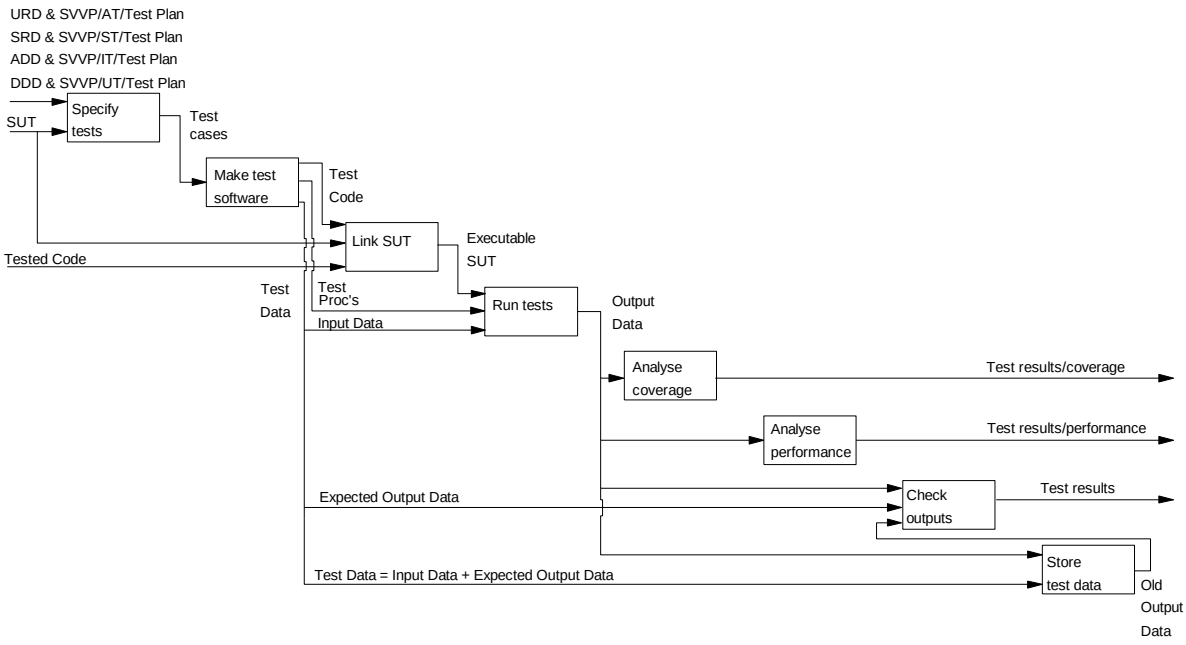

Figure 2.6: Testing activities

5. The 'analyse coverage' activity checks that the tests have in fact executed those parts of the SUT that they were intended to test.
6. The 'analyse performance' activity studies the resource consumption of the SUT (e.g. CPU time, disk space, memory).
7. The 'check outputs' activity compares the outputs with the expected output data or the outputs of previous tests, and decides whether the tests have passed or failed.

8. The 'store test data' activity stores test data for reruns of tests. Test output data needs to be retained as evidence that the tests have been performed.

The following sections discuss the specific approaches to unit testing, integration testing, system testing and acceptance testing. For each type of testing sections are provided on:

- test planning;
- test design;
- test case specification;
- test procedure definition;
- test reporting.

2.6.1 Unit tests

A 'unit' of software is composed of one or more modules. In ESA PSS-05-0, 'unit testing' refers to the process of testing modules against the detailed design. The inputs to unit testing are the successfully compiled modules from the coding process. These are assembled during unit testing to make the largest units, i.e. the components of architectural design. The successfully tested architectural design components are the outputs of unit testing.

An incremental assembly sequence is normally best. When the sequence is top-down, the unit grows during unit testing from a kernel module to the major component required in the architectural design. When the sequence is bottom-up, units are assembled from smaller units. Normally a combination of the two approaches is used, with the objective of minimising the amount of test software, measured both in terms of the number of test modules and the number of lines of test code. This enables the test software to be easily verified by inspection.

Studies of traditional developments show that approximately 65% of bugs can be caught in unit testing, and that half these bugs will be caught by 'white-box' tests [Ref 12]. These results show that unit testing is the most effective type of testing for removing bugs. This is because less software is involved when the test is performed, and so bugs are easier to isolate.

2.6.1.1 Unit test planning

The first step in unit testing is to construct a unit test plan and document it in the SVVP (SVV18). This plan is defined in the DD phase and

should describe the scope, approach, resources and schedule of the intended unit tests. The scope of unit testing is to verify the design and implementation of all components from the lowest level defined in the detailed design up to and including the lowest level in the architectural design. The approach should outline the types of tests, and the amounts of testing, required.

The amount of unit testing required is dictated by the need to execute every statement in a module at least once (DD06). The simplest measure of the amount of testing required is therefore just the number of lines of code.

Execution of every statement in the software is normally not sufficient, and coverage of every branch in the logic may be required. The amount of unit testing then depends principally on the complexity of the software. The 'Structured Testing' method (see Section 3.6) uses the cyclomatic complexity metric to evaluate the testability of module designs. The number of test cases necessary to ensure that every branch in the module logic is covered during testing is equivalent to the cyclomatic complexity of the module. The Structured Testing method is strongly recommended when full branch coverage is a requirement.

2.6.1.2 Unit test design

The next step in unit testing is unit test design (SVV19). Unit test designs should specify the details of the test approach for each software component defined in the DDD, and identify the associated test cases and test procedures. The description of the test approach should state the assembly sequence for constructing the architectural design units, and the types of tests necessary for individual modules (e.g. white-box, black-box).

The three rules of incremental assembly are:

- assemble the architectural design units incrementally, module-by-module if possible, because problems that arise in a unit test are most likely to be related to the module that has just been added;
- introduce producer modules before consumer modules, because the former can provide control and data flows required by the latter.
- ensure that each step is reversible, so that rollback to a previous stage in the assembly is always possible.

A simple example of unit test design is shown in Figure 2.6.1.2A. The unit U1 is a major component of the architectural design. U1 is composed of modules M1, M2 and M3. Module M1 calls M2 and then M3,

as shown by the structure chart. Two possible assembly sequences are shown. The sequence starting with M1 is 'top-down' and the sequence starting with M2 is 'bottom-up'. Figure 2.6.1.2B shows that data flows from M2 to M3 under the control of M1.

Each sequence in Figure 2.6.1.2A requires two test modules. The top-down sequence requires the two stub modules S2 and S3 to simulate M2 and M3. The bottom-up sequence requires the drivers D2 and D3 to simulate M1, because each driver simulates a different interface. If M1, M2 and M3 were tested individually before assembly, four drivers and stubs would be required. The incremental approach only requires two.

The rules of incremental assembly argue for top-down assembly instead of bottom-up because the top-down sequence introduces the:

- modules one-by-one;
- producer modules before consumer modules (i.e. M1 before M2 before M3).

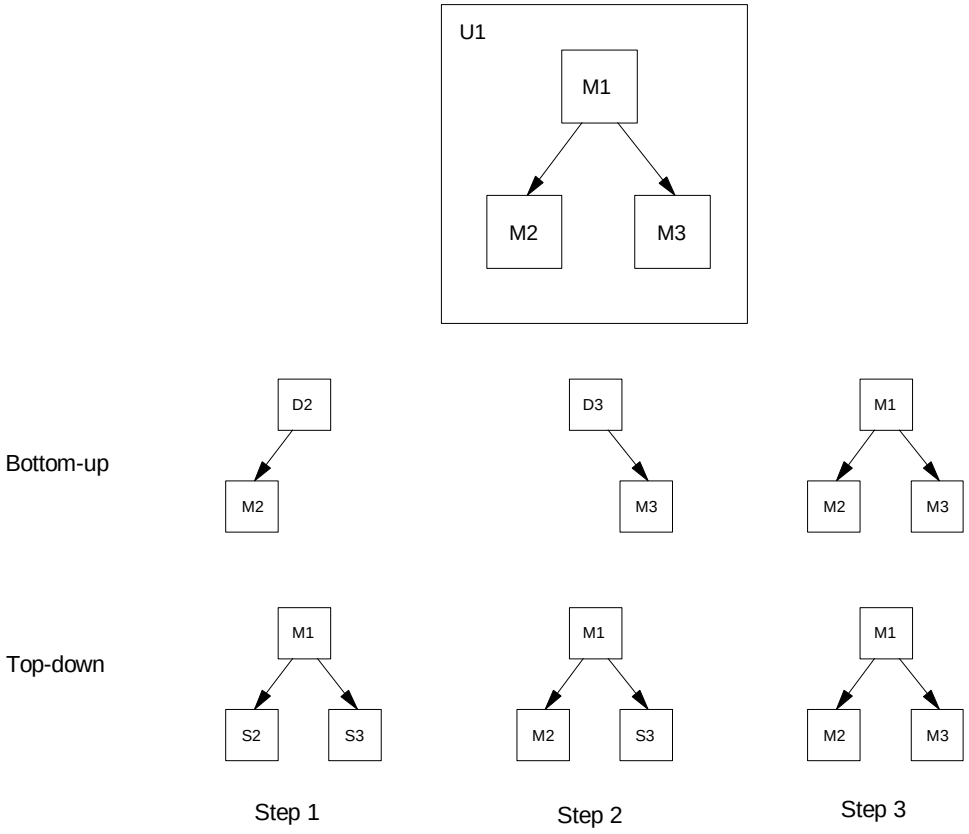

Figure 2.6.1.2A: Example of unit test design

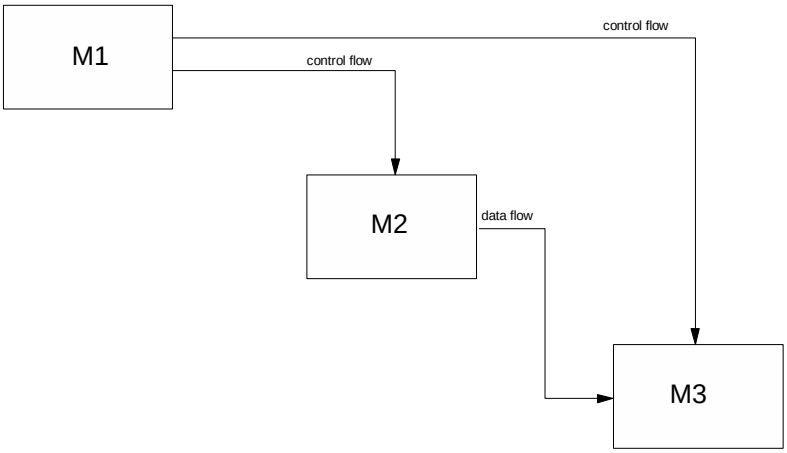

Figure 2.6.1.2B: Data flow dependencies between the modules of U1

2.6.1.2.1 White-box unit tests

The objective of white-box testing is to check the internal logic of the software. White-box tests are sometimes known as 'path tests', 'structure tests' or 'logic tests'. A more appropriate title for this kind of test is 'glass-box test', as the engineer can see almost everything that the code is doing.

White-box unit tests are designed by examining the internal logic of each module and defining the input data sets that force the execution of different paths through the logic. Each input data set is a test case.

Traditionally, programmers used to insert diagnostic code to follow the internal processing (e.g. statements that print out the values of program variables during execution). Debugging tools that allow programmers to observe the execution of a program step-by-step in a screen display make the insertion of diagnostic code unnecessary, unless manual control of execution is not appropriate, such as when real-time code is tested.

When debugging tools are used for white-box testing, prior preparation of test cases and procedures is still necessary. Test cases and procedures should not be invented during debugging. The Structured Testing method (see Section 3.6) is the best known method for white-box unit testing. The cyclomatic complexity value gives the number of paths that must be executed, and the 'baseline method' is used to define the paths. Lastly, input values are selected that will cause each path to be executed. This is called 'sensitising the path'.

A limitation of white-box testing is its inability to show missing logic. Black-box tests remedy this deficiency.

2.6.1.2.2 Black-box unit tests

The objective of black-box tests is to verify the functionality of the software. The tester treats the module as 'black-box' whose internals cannot be seen. Black-box tests are sometimes called 'function tests'.

Black-box unit tests are designed by examining the specification of each module and defining input data sets that will result in different behaviour (e.g. outputs). Each input data set is a test case.

Black-box tests should be designed to exercise the software for its whole range of inputs. Most software items will have many possible input data sets and using them all is impractical. Test designers should partition the range of possible inputs into 'equivalence classes'. For any given error, input data sets in the same equivalence class will produce the same error [Ref 14].

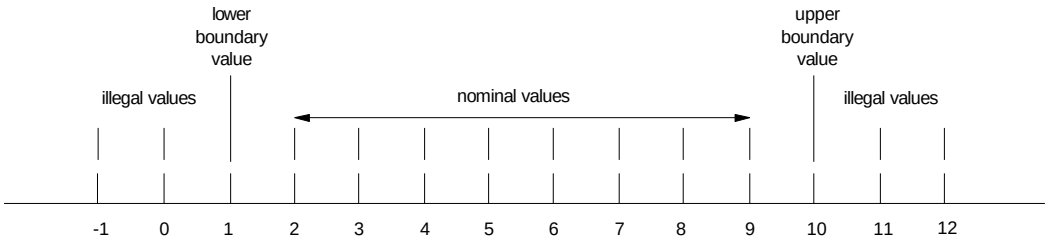

Figure 2.6.1.2.2: Equivalence partitioning example

Consider a module that accepts integers in the range 1 to 10 as input, for example. The input data can be partitioned into five equivalence classes as shown in Figure 2.6.1.2.2. The five equivalence classes are the illegal values below the lower boundary, such as 0, the lower boundary value 1, the nominal values 2 to 9, the upper boundary value 10, and the illegal values above the upper boundary, such as 11.

Output values can be used to generate additional equivalence classes. In the example above, if the output of the routine generated the result TRUE for input numbers less than or equal to 5 and FALSE for numbers greater than 5, the nominal value equivalence class should be split into two subclasses:

- nominal values giving a TRUE result, such as 3;
- boundary nominal value, i.e. 5;
- nominal values giving a FALSE result, such as 7.

Equivalence classes may be defined by considering all possible data types. For example the module above accepts integers only. Test cases could be devised using real, logical and character data.

Having defined the equivalence classes, the next step is to select suitable input values from each equivalence class. Input values close to the boundary values are normally selected because they are usually more effective in causing test failures (e.g. 11 might be expected to be more likely to produce a test failure than 99).

Although equivalence partitioning combined with boundary-value selection is a useful technique for generating efficient input data sets, it will not expose bugs linked to combinations of input data values. Techniques such as decision tables [Ref 12] and cause-effect graphs [Ref 14] can be very useful for defining tests that will expose such bugs.

	1	2	3	4
open_pressed	TRUE	TRUE	FALSE	FALSE
close_pressed	TRUE	FALSE	TRUE	FALSE
action	?	OPEN	CLOSE	?

Table 2.6.1.2.2: Decision table example

Table 2.6.1.2.2 shows the decision table for a module that has Boolean inputs that indicate whether the OPEN or CLOSE buttons of an elevator door have been pressed. When open_pressed is true and close_pressed is false, the action is OPEN. When close_pressed is true and open_pressed is false, the action is CLOSE. Table 2.6.1.2.2 shows that the outcomes for when open_pressed and close_pressed are both true and both false are undefined. Additional test cases setting open_pressed and close_pressed both true and then both false are likely to expose problems.

A useful technique for designing tests for real-time systems is the state-transition table. These tables define what messages can be processed in each state. For example, sending the message 'open doors' to an elevator in the state 'moving' should be rejected. Just as with decision tables, undefined outcomes shown by blank table entries make good candidates for testing.

Decision tables, cause-effect graphs and state-transition diagrams are just three of the many analysis techniques that can be employed for test design. After tests have been devised by means of these techniques, test designers should examine them to see whether additional tests are needed, their judgement being based upon their experience of similar systems or their involvement in the development of the system. This technique, called 'error guessing' [Ref 14], should be risk-driven, focusing on the parts of the design that are novel or difficult to verify by other means, or where quality problems have occurred before.

Test tools that allow the automatic creation of drivers, stubs and test data sets help make black-box testing easier (see Chapter 4). Such tools can define equivalence classes based upon boundary values in the input, but the identification of more complex test cases requires knowledge of the how the software should work.

2.6.1.2.3 Performance tests

The DDD may have placed resource constraints on the performance of a module. For example a module may have to execute within a specified elapsed time, or use less than a specified amount of CPU time, or consume less than a specified amount of memory. Compliance with these constraints should be tested as directly as possible, for example by means of:

- performance analysis tools;
- diagnostic code;
- system monitoring tools.

2.6.1.3 Unit test case definition

Each unit test design will use one or more unit test cases, which must also be documented in the SVVP (SVV20). Test cases should specify the inputs, predicted results and execution conditions for a test case.

2.6.1.4 Unit test procedure definition

The unit test procedures must be described in the SVVP (SVV21). These should provide a step-by-step description of how to carry out each test case. One test procedure may execute one or more test cases. The procedures may use executable 'scripts' that control the operation of test tools. With the incremental approach, the input data required to test a module may be created by executing an already tested module (e.g. M2 is used to create data for M1 and M3 in the example above). The test procedure should define the steps needed to create such data.

2.6.1.5 Unit test reporting

Unit test results may be reported in a variety of ways. Some common means of recording results are:

- unit test result forms, recording the date and outcome of the test cases executed by the procedure;
- execution logfile.

2.6.2 Integration tests

A software system is composed of one or more subsystems, which are composed of one or more units (which are composed of one or more modules). In ESA PSS-05-0, 'integration testing' refers to the process of testing units against the architectural design. During integration testing, the architectural design units are integrated to make the system.

The 'function-by-function' integration method described in Section 3.3.2.1 of ESA PSS-05-05 'Guide to the Detailed Design and Production Phase' [Ref 3] should be used to integrate the software. As with the approach described for unit testing, this method minimises the amount of test software required. The steps are to:

1. select the functions to be integrated;
2. identify the components that carry out the functions;
3. order the components by the number of dependencies (i.e. fewest dependencies first);
4. create a driver to simulate the input of the component later in the order when a component depends on another later in the order;
5. introduce the components with fewest dependencies first.

Though the errors found in integration testing should be much fewer than those found in unit testing, they are more time-consuming to diagnose and fix. Studies of testing [Ref 15] have shown architectural errors can be as much as thirty times as costly to repair as detailed design errors.

2.6.2.1 Integration test planning

The first step in integration testing is to construct an integration test plan and document it in the SVVP (SVV17). This plan is defined in the AD phase and should describe the scope, approach, resources and schedule of the intended integration tests.

The scope of integration testing is to verify the design and implementation of all components from the lowest level defined in the architectural design up to the system level. The approach should outline the types of tests, and the amounts of testing, required.

The amount of integration testing required is dictated by the need to:

- check that all data exchanged across an interface agree with the data structure specifications in the ADD (DD07);
- confirm that all the control flows in the ADD have been implemented (DD08).

The amount of control flow testing required depends on the complexity of the software. The Structured Integration Testing method (see Section 3.7) uses the integration complexity metric to evaluate the testability of architectural designs. The integration complexity value is the number of integration tests required to obtain full coverage of the control flow. The Structured Integration Testing method is strongly recommended for estimating the amount of integration testing.

2.6.2.2 Integration test design

The next step in integration testing is integration test design (SVV19). This and subsequent steps are performed in the DD phase, although integration test design may be attempted in the AD phase. Integration test designs should specify the details of the test approach for each software component defined in the ADD, and identify the associated test cases and test procedures.

The description of the test approach should state the:

- integration sequence for constructing the system;
- types of tests necessary for individual components (e.g. white-box, black-box).

With the function-by-function method, the system grows during integration testing from the kernel units that depend upon few other units, but are depended upon by many other units. The early availability of these kernel units eases subsequent testing.

For incremental delivery, the delivery plan will normally specify what functions are required in each delivery. Even so, the number of dependencies can be used to decide the order of integration of components in each delivery.

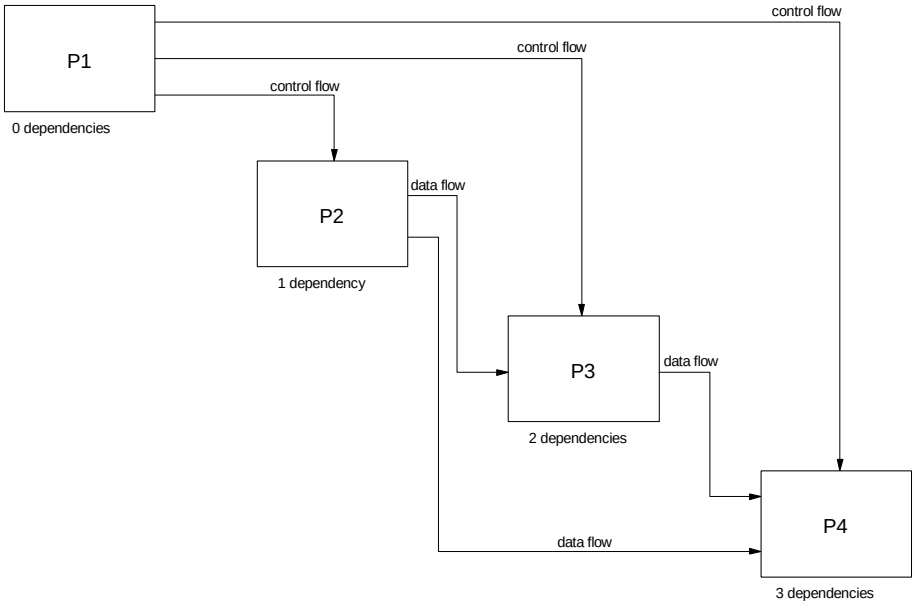

Figure 2.6.2A: Incremental integration sequences

Figure 2.6.2A shows a system composed of four programs P1, P2, P3 and P4. P1 is the 'program manager', providing the user interface and controlling the other programs. Program P2 supplies data to P3, and both P2 and P3 supply data to P4. User inputs are ignored. P1 has zero dependencies, P2 has one, P3 has two and P4 has three. The integration sequence is therefore P1, P2, P3 and then P4.

2.6.2.2.1 White-box integration tests

White-box integration tests should be defined to verify the data and control flow across interfaces between the major components defined in the ADD (DD07 and DD08). For file interfaces, test programs that print the contents of the files provide the visibility required. With real-time systems, facilities for trapping messages and copying them to a log file can be employed. Debuggers that set break points at interfaces can also be useful. When control or data flow traverses an interface where a break point is set, control is passed to the debugger, enabling inspection and logging of the flow.

The Structured Integration Testing method (see Section 3.7) is the best known method for white-box integration testing. The integration complexity value gives the number of control flow paths that must be executed, and the 'design integration testing method' is used to define the control flow paths. The function-by-function integration method (see Section

2.6.2) can be used to define the order of testing the required control flow paths.

The addition of new components to a system often introduces new execution paths through it. Integration test design should identify paths suitable for testing and define test cases to check them. This type of path testing is sometimes called 'thread testing'. All new control flows should be tested.

2.6.2.2.2 Black-box integration tests

Black-box integration tests should be used to fully exercise the functions of each component specified in the ADD. Black-box tests may also be used to verify that data exchanged across an interface agree with the data structure specifications in the ADD (DD07).

2.6.2.2.3 Performance tests

The ADD may have placed resource constraints on the performance of a unit. For example a program may have to respond to user input within a specified elapsed time, or process a defined number of records within a specified CPU time, or occupy less than a specified amount of disk space or memory. Compliance with these constraints should be tested as directly as possible, for example by means of:

- performance analysis tools;
- diagnostic code;
- system monitoring tools.

2.6.2.3 Integration test case definition

Each integration test design will use one or more integration test cases, which must also be documented in the SVVP (SVV20). Test cases should specify the inputs, predicted results and execution conditions for a test case.

2.6.2.4 Integration test procedure definition

The integration test procedures must be described in the SVVP (SVV21). These should provide a step-by-step description of how to carry out each test case. One test procedure may execute one or more test cases. The procedures may use executable 'scripts' that control the operation of test tools.

2.6.2.5 Integration test reporting

Integration test results may be reported in a variety of ways. Some common means of recording results are:

- integration test result forms, recording the date and outcome of the test cases executed by the procedure;
- execution logfile.

2.6.3 System tests

In ESA PSS-05-0, 'system testing' refers to the process of testing the system against the software requirements. The input to system testing is the successfully integrated system.

Wherever possible, system tests should be specified and performed by an independent testing team. This increases the objectivity of the tests and reduces the likelihood of defects escaping the software verification and validation net.

2.6.3.1 System test planning

The first step in system testing is to construct a system test plan and document it in the SVVP (SVV14). This plan is defined in the SR phase and should describe the scope, approach, resources and schedule of the intended system tests. The scope of system testing is to verify compliance with the system objectives, as stated in the SRD (DD09). System testing must continue until readiness for transfer can be demonstrated.

The amount of testing required is dictated by the need to cover all the software requirements in the SRD. A test should be defined for every essential software requirement, and for every desirable requirement that has been implemented.

2.6.3.2 System test design

The next step in system testing is system test design (SVV19). This and subsequent steps are performed in the DD phase, although system test design may be attempted in the SR and AD phases. System test designs should specify the details of the test approach for each software requirement specified in the SRD, and identify the associated test cases and test procedures. The description of the test approach should state the types of tests necessary (e.g. function test, stress test etc).

Knowledge of the internal workings of the software should not be required for system testing, and so white-box tests should be avoided. Black-box and other types of test should be used wherever possible. When a test of a requirement is not possible, an alternative method of verification should be used (e.g. inspection).

System testing tools can often be used for problem investigation during the TR and OM phases. Effort invested in producing efficient easy-to-use diagnostic tools at this stage of development is often worthwhile.

If an incremental delivery or evolutionary development approach is being used, system tests of each release of the system should include regression tests of software requirements verified in earlier releases.

The SRD will contain several types of requirements, each of which needs a distinct test approach. The following subsections discuss possible approaches.

2.6.3.2.1 Function tests

System test design should begin by designing black-box tests to verify each functional requirement. Working from the functional requirements in the SRD, techniques such as decision tables, state-transition tables and error guessing are used to design function tests.

2.6.3.2.2 Performance tests

Performance requirements should contain quantitative statements about system performance. They may be specified by stating the:

- worst case that is acceptable;
- nominal value, to be used for design;
- best case value, to show where growth potential is needed.

System test cases should be designed to verify:

- that all worst case performance targets have been met;
- that nominal performance targets are usually achieved;
- whether any best-case performance targets have been met.

In addition, stress tests (see Section 2.6.3.2.13) should be designed to measure the absolute limits of performance.

2.6.3.2.3 Interface tests

System tests should be designed to verify conformance to external interface requirements. Interface Control Documents (ICDs) form the baseline for testing external interfaces. Simulators and other test tools will be necessary if the software cannot be tested in the operational environment.

Tools (not debuggers) should be provided to:

- convert data flows into a form readable by human operators;
- edit the contents of data stores.

2.6.3.2.4 Operations tests

Operations tests include all tests of the user interface, man machine interface, or human computer interaction requirements. They also cover the logistical and organisational requirements. These are essential before the software is delivered to the users.

Operations tests should be designed to show up deficiencies in usability such as:

- instructions that are difficult to follow;
- screens that are difficult to read;
- commonly-used operations with too many steps;
- meaningless error messages.

The operational requirements may have defined the time required to learn and operate the software. Such requirements can be made the basis of straightforward tests. For example a test of usability might be to measure the time an operator with average skill takes to learn how to restart the system.

Other kinds of tests may be run throughout the system-testing period, for example:

- do all warning messages have a red background?
- is there help on this command?

If there is a help system, every topic should be systematically inspected for accuracy and appropriateness.

Response times should normally be specified in the performance requirements (as opposed to operational requirements). Even so, system

tests should verify that the response time is short enough to make the system usable.

2.6.3.2.5 Resource tests

Requirements for the usage of resources such as CPU time, storage space and memory may have been set in the SRD. The best way to test for compliance to these requirements is to allocate these resources and no more, so that a failure occurs if a resource is exhausted. If this is not suitable (e.g. it is usually not possible to specify the maximum size of a particular file), alternative approaches are to:

- use a system monitoring tool to collect statistics on resource consumption;
- check directories for file space used.

2.6.3.2.6 Security tests

Security tests should check that the system is protected against threats to confidentiality, integrity and availability.

Tests should be designed to verify that basic security mechanisms specified in the SRD have been provided, for example:

- password protection;
- resource locking.

Deliberate attempts to break the security mechanisms are an effective way of detecting security errors. Possible tests are attempts to:

- access the files of another user;
- break into the system authorisation files;
- access a resource when it is locked;
- stop processes being run by other users.

Security problems can often arise when users are granted system privileges unnecessarily. The Software User Manual should clearly state the privileges required to run the software.

Experience of past security problems should be used to check new systems. Security loopholes often recur.

2.6.3.2.7 Portability tests

Portability requirements may require the software to be run in a variety of environments. Attempts should be made to verify portability by running a representative selection of system tests in all the required environments. If this is not possible, indirect techniques may be attempted. For example if a program is supposed to run on two different platforms, a programming language standard (e.g. ANSI C) might be specified and a static analyser tool used to check conformance to the standard. Successfully executing the program on one platform and passing the static analysis checks might be adequate proof that the software will run on the other platform.

2.6.3.2.8 Reliability tests

Reliability requirements should define the Mean Time Between Failure (MTBF) of the software. Separate MTBF values may have been specified for different parts of the software.

Reliability can be estimated from the software problems reported during system testing. Tests designed to measure the performance limits should be excluded from the counts, and test case failures should be categorised (e.g. critical, non-critical). The mean time between failures can then be estimated by dividing the system testing time by the number of critical failures.

2.6.3.2.9 Maintainability tests

Maintainability requirements should define the Mean Time To Repair (MTTR) of the software. Separate MTTR values may have been specified for different parts of the software.

Maintainability should be estimated by averaging the difference between the dates of Software Problem Reports (SPRs) reporting critical failures that occur during system testing, and the corresponding Software Modification Reports (SMRs) reporting the completion of the repairs.

Maintainability requirements may have included restrictions on the size and complexity of modules, or even the use of the programming language. These should be tested by means of a static analysis tool. If a static analysis tool is not available, samples of the code should be manually inspected.

2.6.3.2.10 Safety tests

Safety requirements may specify that the software must avoid injury to people, or damage to property, when it fails. Compliance to safety requirements can be tested by:

- deliberately causing problems under controlled conditions and observing the system behaviour (e.g. disconnecting the power during system operations);
- observing system behaviour when faults occur during tests.

Simulators may have to be built to perform safety tests.

Safety analysis classifies events and states according to how much of a hazard they cause to people or property. Hazards may be catastrophic (i.e. life-threatening), critical, marginal or negligible [Ref 24]. Safety requirements may identify functions whose failure may cause a catastrophic or critical hazard. Safety tests may require exhaustive testing of these functions to establish their reliability.

2.6.3.2.11 Miscellaneous tests

An SRD may contain other requirements for:

- documentation (particularly the SUM);
- verification;
- acceptance testing;
- quality, other than reliability, maintainability and safety.

It is usually not possible to test for compliance to these requirements, and they are normally verified by inspection.

2.6.3.2.12 Regression tests

Regression testing is 'selective retesting of a system or component, to verify that modifications have not caused unintended effects, and that the system or component still complies with its specified requirements' [Ref 6].

Regression tests should be performed before every release of the software in the OM phase. If an incremental delivery or evolutionary development approach is being used, regression tests should be performed to verify that the capabilities of earlier releases are unchanged.

Traditionally, regression testing often requires much effort, increasing the cost of change and reducing its speed. Test tools that

automate regression testing are now widely available and can greatly increase the speed and accuracy of regression testing (see Chapter 4). Careful selection of test cases also reduces the cost of regression testing, and increases its effectiveness.

2.6.3.2.13 Stress tests

Stress tests 'evaluate a system or software component at or beyond the limits of its specified requirements' [Ref 6]. The most common kind of stress test is to measure the maximum load the SUT can sustain for a time, for example the:

- maximum number of activities that can be supported simultaneously;
- maximum quantity of data that can be processed in a given time.

Another kind of stress test, sometimes called a 'volume test' [Ref 14], exercises the SUT with an abnormally large quantity of input data. For example a compiler might be fed a source file with very many lines of code, or a database management system with a file containing very many records. Time is not of the essence in a volume test.

Most software has capacity limits. Testers should examine the software documentation for statements about the amount of input the software can accept, and design tests to check that the stated capacity is provided. In addition, testers should look for inputs that have no constraint on capacity, and design tests to check whether undocumented constraints do exist.

2.6.3.3 System test case definition

The system test cases must be described in the SVVP (SVV20). These should specify the inputs, predicted results and execution conditions for a test case.

2.6.3.4 System test procedure definition

The system test procedures must be described in the SVVP (SVV21). These should provide a step-by-step description of how to carry out each test case. One test procedure may execute one or more test cases. The procedures may use executable 'scripts' that control the operation of test tools.

2.6.3.5 System test reporting

System test results may be reported in a variety of ways. Some common means of recording results are:

- system test result forms recording the date and outcome of the test cases executed by the procedure;
- execution logfile.

System test results should reference any Software Problem Reports raised during the test.

2.6.4 Acceptance tests

In ESA PSS-05-0, 'acceptance testing' refers to the process of testing the system against the user requirements. The input to acceptance testing is the software that has been successfully tested at system level.

Acceptance tests should always be done by the user or their representatives. If this is not possible, they should witness the acceptance tests and sign off the results.

2.6.4.1 Acceptance test planning

The first step in acceptance testing is to construct an acceptance test plan and document it in the SVVP (SVV11). This plan is defined in the UR phase and should describe the scope, approach, resources and schedule of the intended acceptance tests. The scope of acceptance testing is to validate that the software is compliant with the user requirements, as stated in the URD. Acceptance tests are performed in the TR phase, although some acceptance tests of quality, reliability, maintainability and safety may continue into the OM phase until final acceptance is possible.

The amount of testing required is dictated by the need to cover all the user requirements in the URD. A test should be defined for every essential user requirement, and for every desirable requirement that has been implemented

2.6.4.2 Acceptance test design

The next step in acceptance testing is acceptance test design (SVV19). This and subsequent steps are performed in the DD phase, although acceptance test design may be attempted in the UR, SR and AD phases. Acceptance test designs should specify the details of the test

approach for a user requirement, or combination of user requirements, and identify the associated test cases and test procedures. The description of the test approach should state the necessary types of tests.

Acceptance testing should require no knowledge of the internal workings of the software, so white-box tests cannot be used.

If an incremental delivery or evolutionary development approach is being used, acceptance tests should only address the user requirements of the new release. Regression tests should have been performed in system testing.

Dry-runs of acceptance tests should be performed before transfer of the software. Besides exposing any faults that have been overlooked, dry-runs allow the acceptance test procedures to be checked for accuracy and ease of understanding.

The specific requirements in the URD should be divided into capability requirements and constraint requirements. The following subsections describe approaches to testing each type of user requirement.

2.6.4.2.1 Capability tests

Capability requirements describe what the user can do with the software. Tests should be designed that exercise each capability. System test cases that verify functional, performance and operational requirements may be reused to validate capability requirements.

2.6.4.2.2 Constraint tests

Constraint requirements place restrictions on how the software can be built and operated. They may predefine external interfaces or specify attributes such as adaptability, availability, portability and security. System test cases that verify compliance with requirements for interfaces, resources, security, portability, reliability, maintainability and safety may be reused to validate constraint requirements.

2.6.4.3 Acceptance test case specification

The acceptance test cases must be described in the SVVP (SVV20). These should specify the inputs, predicted results and execution conditions for a test case.

2.6.4.4 Acceptance test procedure specification

The acceptance test procedures must be described in the SVVP (SVV21). These should provide a step-by-step description of how to carry out each test case. The effort required of users to validate the software should be minimised by means of test tools.

2.6.4.5 Acceptance test reporting

Acceptance test results may be reported in a variety of ways. Some common means of recording results are:

- acceptance test result forms recording the date and outcome of the test cases executed by the procedure;
- execution logfile.

Acceptance test results should reference any Software Problem Reports raised during the test.

CHAPTER 3
SOFTWARE VERIFICATION AND VALIDATION METHODS

3.1 INTRODUCTION

This chapter discusses methods for software verification and validation that may be used to enhance the basic approach described in Chapter 2. The structure of this chapter follows that of the previous chapter, as shown in Table 3.1. Supplementary methods are described for reviews, formal proof and testing.

Activity	Supplementary method
review	software inspection
tracing	none
formal proof	formal methods program verification techniques
testing	structured testing structured integration testing

Table 3.1: Structure of Chapter 3

3.2 SOFTWARE INSPECTIONS

Software inspections can be used for the detection of defects in detailed designs before coding, and in code before testing. They may also be used to verify test designs, test cases and test procedures. More generally, inspections can be used for verifying the products of any development process that is defined in terms of:

- operations (e.g. 'code module');
- exit criteria (e.g. 'module successfully compiles').

Software inspections are efficient. Projects can detect over 50% of the total number of defects introduced in development by doing them [Ref 21, 22].

Software inspections are economical because they result in significant reductions in both the number of defects and the cost of their

removal. Detection of a defect as close as possible to the time of its introduction results in:

- an increase in the developers' awareness of the reason for the defect's occurrence, so that the likelihood that a similar defect will recur again is reduced;
- reduced effort in locating the defect, since no effort is required to diagnose which component, out of many possible components, contains the defect.

Software inspections are formal processes. They differ from walkthroughs (see Section 2.3.2) by:

- repeating the process until an acceptable defect rate (e.g. number of errors per thousand lines of code) has been achieved;
- analysing the results of the process and feeding them back to improve the production process, and forward to give early measurements of software quality;
- avoiding discussion of solutions;
- including rework and follow-up activities.

The following subsections summarise the software inspection process. The discussion is based closely on the description given by Fagan [Ref 21 and 22] and ANSI/IEEE Std 1028-1988, 'IEEE Standard for Software Reviews and Audits' [Ref 10].

3.2.1 Objectives

The objective of a software inspection is to detect defects in documents or code.

3.2.2 Organisation

There are five roles in a software inspection:

- moderator;
- secretary;
- reader;
- inspector;
- author.

The moderator leads the inspection and chairs the inspection meeting. The person should have implementation skills, but not necessarily

be knowledgeable about the item under inspection. He or she must be impartial and objective. For this reason moderators are often drawn from staff outside the project. Ideally they should receive some training in inspection procedures.

The secretary is responsible for recording the minutes of inspection meetings, particularly the details about each defect found.

The reader guides the inspection team through the review items during the inspection meetings.

Inspectors identify and describe defects in the review items under inspection. They should be selected to represent a variety of viewpoints (e.g. designer, coder and tester).

The author is the person who has produced the items under inspection. The author is present to answer questions about the items under inspection, and is responsible for all rework.

A person may have one or more of the roles above. In the interests of objectivity, no person may share the author role with another role.

3.2.3 Input

The inputs to an inspection are the:

- review items;
- specifications of the review items;
- inspection checklist;
- standards and guidelines that apply to the review items;
- inspection reporting forms;
- defect list from a previous inspection.

3.2.4 Activities

A software inspection consists of the following activities:

- overview;
- preparation;
- review meeting;
- rework;
- follow-up.

The overview is a presentation of the items being inspected. Inspectors then prepare themselves for the review meeting by familiarising themselves with the review items. They then examine the review items, identify defects, and decide whether they should be corrected or not, at the review meeting. Rework activities consist of the repair of faults. Follow-up activities check that all decisions made by the review meeting are carried out.

Before the overview, the moderator should:

- check that the review items are ready for inspection;
- arrange a date, time and place for the overview and review meetings;
- distribute the inputs if no overview meeting is scheduled.

Organisations should collect their own inspection statistics and use them for deciding the number and duration of inspections. The following figures may be used as the starting point for inspections of code [Ref 22]:

- preparation: 125 non-comment lines of source code per hour;
- review meeting: 90 non-comment lines of source code per hour.

These figures should be doubled for inspections of pseudo code or program design language.

Review meetings should not last more than two hours. The efficiency of defect detection falls significantly when meetings last longer than this.

3.2.4.1 Overview

The purpose of the overview is to introduce the review items to the inspection team. The moderator describes the area being addressed and then the specific area that has been designed in detail.

For a reinspection, the moderator should flag areas that have been subject to rework since the previous inspection.

The moderator then distributes the inputs to participants.

3.2.4.2 Preparation

Moderators, readers and inspectors then familiarise themselves with the inputs. They might prepare for a code inspection by reading:

- design specifications for the code under inspection;

- coding standards;
- checklists of common coding errors derived from previous inspections;
- code to be inspected.

Any defects in the review items should be noted on RID forms and declared at the appropriate point in the examination. Preparation should be done individually and not in a meeting.

3.2.4.3 Review meeting

The moderator checks that all the members have performed the preparatory activities (see Section 3.2.4.2). The amount of time spent by each member should be reported and noted.

The reader then leads the meeting through the review items. For documents, the reader may summarise the contents of some sections and cover others line-by-line, as appropriate. For code, the reader covers every piece of logic, traversing every branch at least once. Data declarations should be summarised. Inspectors use the checklist to find common errors.

Defects discovered during the reading should be immediately noted by the secretary. The defect list should cover the:

- severity (e.g. major, minor);
- technical area (e.g. logic error, logic omission, comment error);
- location;
- description.

Any solutions identified should be noted. The inspection team should avoid searching for solutions and concentrate on finding defects.

At the end of the meeting, the inspection team takes one of the following decisions:

- accept the item when the rework (if any) is completed;
- make the moderator responsible for accepting the item when the rework is completed;
- reinspect the whole item (usually necessary if more than 5% of the material requires rework).

The secretary should produce the minutes immediately after the review meeting, so that rework can start without delay.

3.2.4.4 Rework

After examination, software authors correct the defects described in the defect list.

3.2.4.5 Follow-up

After rework, follow-up activities verify that all the defects have been properly corrected and that no secondary defects have been introduced. The moderator is responsible for follow-up.

Other follow-up activities are the:

- updating of the checklist as the frequency of different types of errors change;
- analysis of defect statistics, perhaps resulting in the redirection of SVV effort.

3.2.5 Output

The outputs of an inspection are the:

- defect list;
- defect statistics;
- inspection report.

The inspection report should give the:

- names of the participants;
- duration of the meeting;
- amount of material inspected;
- amount of preparation time spent;
- review decision on acceptance;
- estimates of rework effort and schedule.

3.3 FORMAL METHODS

Formal Methods, such as LOTOS, Z and VDM, possess an agreed notation, with well-defined semantics, and a calculus, which allow proofs to be constructed. The first property is shared with other methods for software specification, but the second sets them apart.

Formal Methods may be used in the software requirements definition phase for the construction of specifications. They are discussed in ESA PSS-05-03, 'Guide to the Software Requirements Definition Phase' [Ref 2].

3.4 PROGRAM VERIFICATION TECHNIQUES

Program verification techniques may be used in the detailed design and production phase to show that a program is consistent with its specification. These techniques require that the:

- semantics of the programming language are formally defined;
- program be formally specified in a notation that is consistent with the mathematical verification techniques used.

If these conditions are not met, formal program verification cannot be attempted [Ref 16].

A common approach to formal program verification is to derive, by stepwise refinement of the formal specification, 'assertions' (e.g. preconditions or postconditions) that must be true at each stage in the processing. Formal proof of the program is achieved by demonstrating that program statements separating assertions transform each assertion into its successor. In addition, it is necessary to show that the program will always terminate (i.e. one or more of the postconditions will always be met).

Formal program verification is usually not possible because the programming language has not been formally defined. Even so, a more pragmatic approach to formal proof is to show that the:

- program code is logically consistent with the program specification;
- program will always terminate.

Assertions are placed in the code as comments. Verification is achieved by arguing that the code complies with the requirements present in the assertions.

3.5 CLEANROOM METHOD

The cleanroom method [Ref 23] replaces unit testing and integration testing with software inspections and program verification techniques. System testing is carried out by an independent testing team.

The cleanroom method is not fully compliant with ESA PSS-05-0 because:

- full statement coverage is not achieved (DD06);
- unit and integration testing are omitted (DD07, DD08).

3.6 STRUCTURED TESTING

Structured Testing is a method for verifying software based upon the mathematical properties of control graphs [Ref 13]. The method:

- improves testability by limiting complexity during detailed design;
- guides the definition of test cases during unit testing.

Software with high complexity is hard to test. The Structured Testing method uses the cyclomatic complexity metric for measuring complexity, and recommends that module designs be simplified until they are within the complexity limits.

Structured Testing provides a technique, called the 'baseline method', for defining test cases. The objective is to cover every branch of the program logic during unit testing. The minimum number of test cases is the cyclomatic complexity value measured in the first step of the method.

The discussion of Structured Testing given below is a summary of that given in Reference 13. The reader is encouraged to consult the reference for a full discussion.

3.6.1 Testability

The testability of software should be evaluated during the detailed design phase by measuring its complexity.

The relationships between the parts of an entity determine its complexity. The parts of a software module are the statements in it. These are related to each other by sequence, selection (i.e. branches or conditions) and iteration (i.e. loops). As loops can be simulated by branching on a condition, McCabe defined a metric that gives a complexity of 1 for a simple sequence of statements with no branches. Only one test is required to execute every statement in the sequence. Each branch added to a module increases the complexity by one, and requires an extra test to cover it.

A control graph represents the control flow in a module. Control graphs are simplified flowcharts. Blocks of statements with sequential logic are represented as 'nodes' in the graph. Branches between blocks of statements (called 'edges') are represented as arrows connecting the nodes. McCabe defines the cyclomatic complexity 'v' of a control graph as:

$$v = e - n + 2$$

where:

- e is the number of edges;
- n is the number of nodes.

Figure 3.6.1A, B and C show several examples of control graphs.

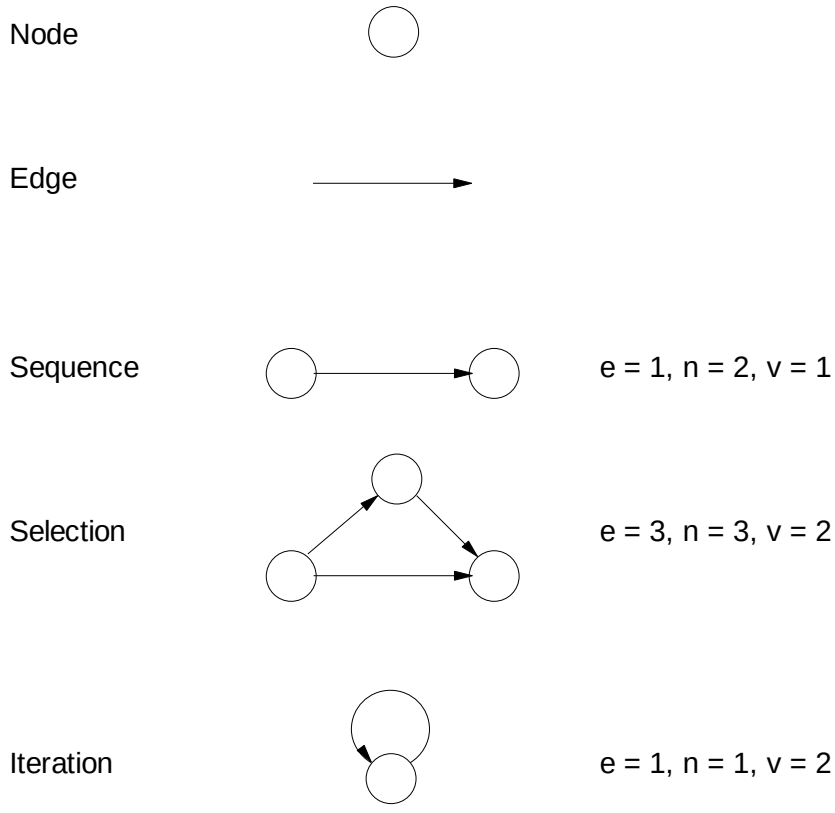

Figure 3.6.1A: Basic control graphs

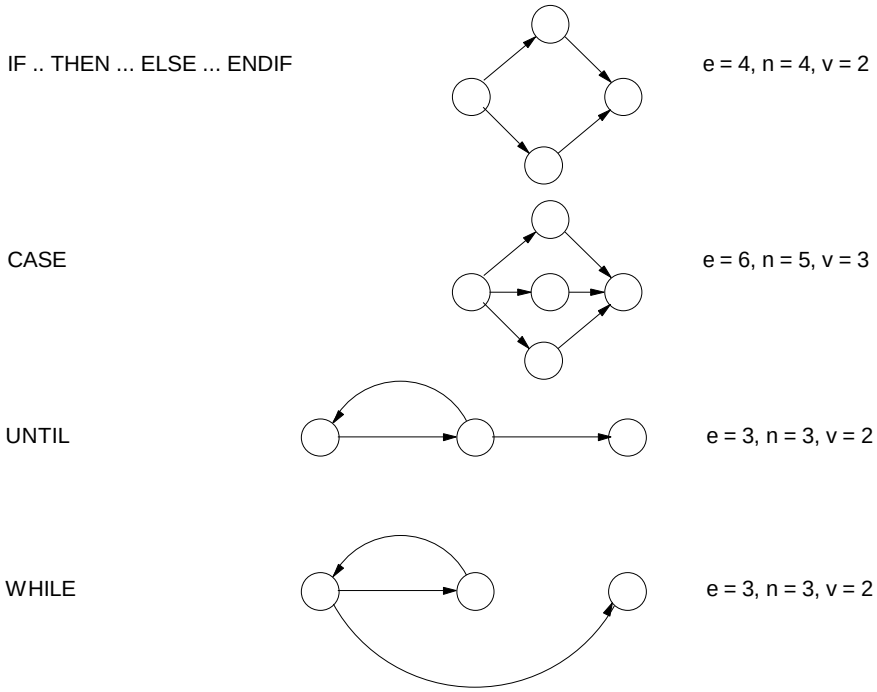

Figure 3.6.1B: Control graphs for structured programming elements

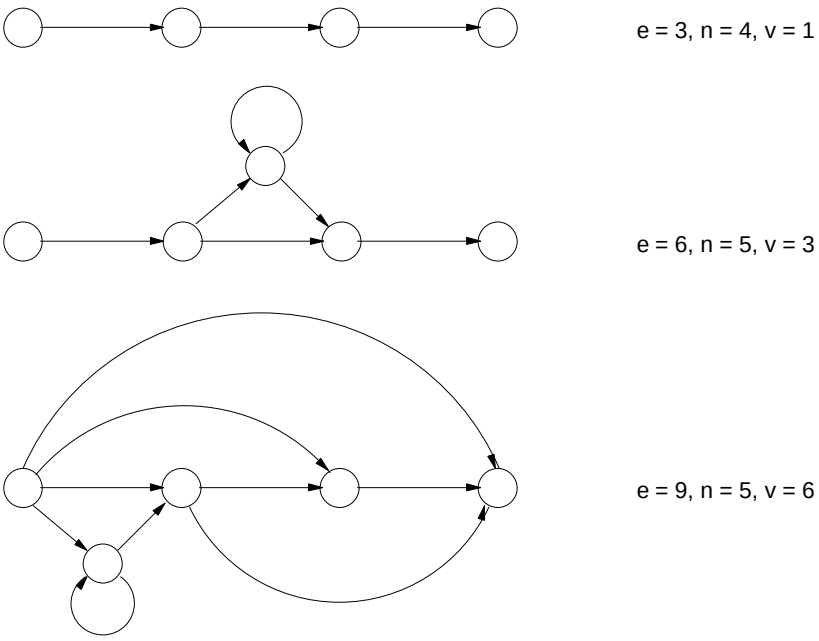

Figure 3.6.1C: Example control graphs

Alternative ways to measure cyclomatic complexity are to count the:

- number of separate regions in the control graph (i.e. areas separated by edges);
- number of decisions (i.e. the second and subsequent branches emanating from a node) and add one.

Myers [Ref 14] has pointed out that decision statements with multiple predicates must be separated into simple predicates before cyclomatic complexity is measured. The decision IF (A .AND. B .AND. C) THEN ... is really three separate decisions, not one.

The Structured Testing Method recommends that the cyclomatic complexity of a module be limited to 10. Studies have shown that errors concentrate in modules with complexities greater than this value. During detailed design, the limit of 7 should be applied because complexity always increases during coding. Modules that exceed these limits should be redesigned. Case statements are the only exception permitted.

The total cyclomatic complexity of a program can be obtained by summing the cyclomatic complexities of the constituent modules. The full cyclomatic complexity formula given by McCabe is:

$$v = e - n + 2p$$

where p is the number of modules. Each module has a separate control graph. Figure 3.6.1D shows how the total cyclomatic complexity can be evaluated by:

- counting all edges (18), nodes (14) and modules (3) and applying the complexity formula, 18 - 14 + 2*3 = 10;
- adding the complexity of the components, 1 + 3 + 6 = 10.

Combining the modules in Figure 3.6.1D into one module gives a module with complexity of eight. Although the total complexity is reduced, this is higher than the complexity of any of the separate modules. In general, decomposing a module into smaller modules increases the total complexity but reduces the maximum module complexity. A useful rule for design decomposition is to continue decomposition until the complexity of each of the modules is 10 or less.

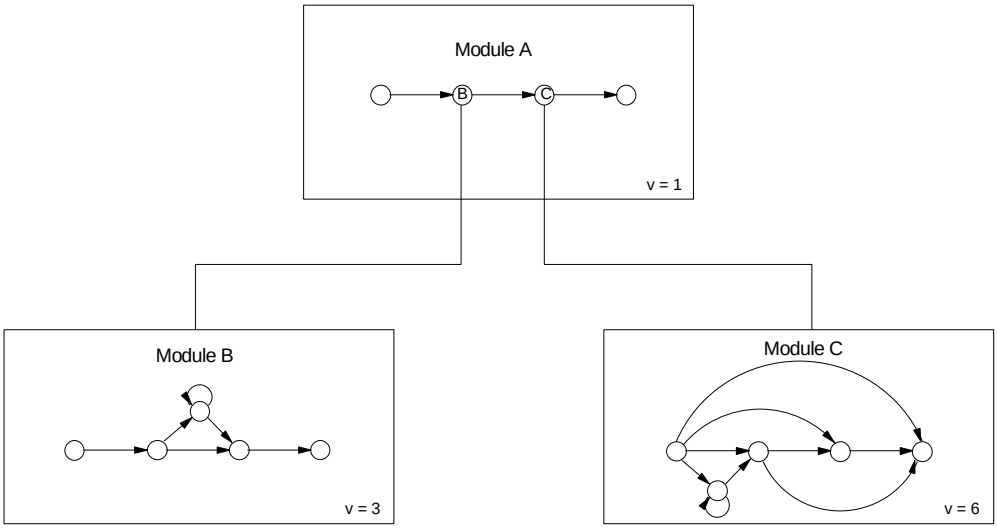

Figure 3.6.1D: Evaluating total cyclomatic complexity

3.6.2 Branch testing

Each branch added to a control graph adds a new path, and in this lies the importance of McCabe's complexity metric for the software tester: the cyclomatic complexity metric, denoted as 'v' by McCabe, measures the minimum number of paths through the software required to ensure that:

- every statement in a program is executed at least once (DD06);
- each decision outcome is executed at least once.

These two criteria imply full 'branch testing' of the software. Every clause of every statement is executed. In simple statement testing, every clause of every statement may not be executed. For example, branch testing of IF (A .EQ. B) X = X/Y requires that test cases be provided for both 'A equal to B' and 'A not equal to B'. For statement testing, only one test case needs to be provided. ESA PSS-05-0 places statement coverage as the minimum requirement. Branch testing should be the verification requirement in most projects.

3.6.3 Baseline method

The baseline method is used in structured testing to decide what paths should be used to traverse every branch. The test designer should

examine the main function of the module and define the 'baseline path' that directly achieves it. Special cases and errors are ignored.

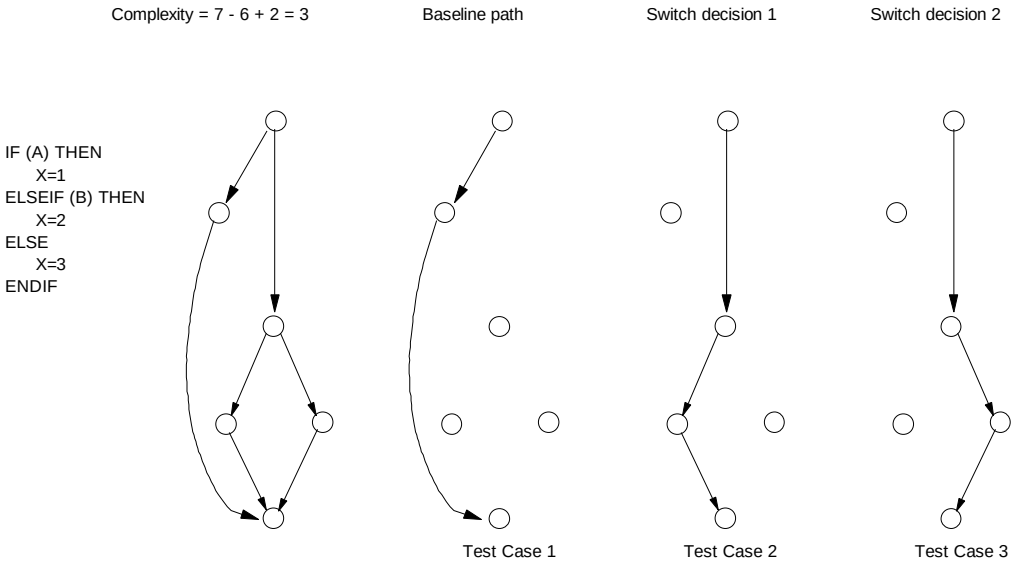

Figure 3.6.3: Baseline Method example.

Figure 3.6.3 shows the principles. The cyclomatic complexity of the module is three, so three test cases are required for full branch coverage. Test case 1 traces the baseline path. The baseline method proceeds by taking each decision in turn and switching it, as shown in test cases 2 and 3. Each switch is reset before switching the next, resulting in the checking of each deviation from the baseline. The baseline method does not require the testing of all possible paths. See the example of untested path of Figure 3.6.3. Tests for paths that are not tested by the baseline method may have to be added to the test design (e.g paths for testing safety-critical functions).

3.7 STRUCTURED INTEGRATION TESTING

Structured Integration Testing [Ref 15] is a method based upon the Structured Testing Method that:

- improves testability by limiting complexity during software architectural design;
- guides the definition of test cases during integration testing.

The method can be applied at all levels of design above the module level. Therefore it may also be applied in unit testing when units assembled from modules are tested.

The discussion of Structured Integration Testing given below is a summary of that given in Reference 15. The reader is encouraged to consult the references for a full discussion.

3.7.1 Testability

Structured Integration Testing defines three metrics for measuring testability:

- module design complexity;
- design complexity;
- integration complexity.

Module design complexity, denoted as 'iv' by McCabe, measures the individual effect of a module upon the program design [Ref 15]. The module design complexity is evaluated by drawing the control graph of the module and then marking the nodes that contain calls to external modules. The control graph is then 'reduced' according to the rules listed below and shown in Figure 3.7.1A:

1. marked nodes cannot be removed;
2. unmarked nodes that contain no decisions are removed;
3. edges that return control to the start of a loop that only contains unmarked nodes are removed;
4. edges that branch from the start of a case statement to the end are removed if none of the other cases contain marked nodes.

As in all control graphs, edges that 'duplicate' other edges are removed. The module design complexity is the cyclomatic complexity of the reduced graph.

In summary, module design complexity ignores paths covered in module testing that do not result in calls to external modules. The remaining paths are needed to test module interfaces.

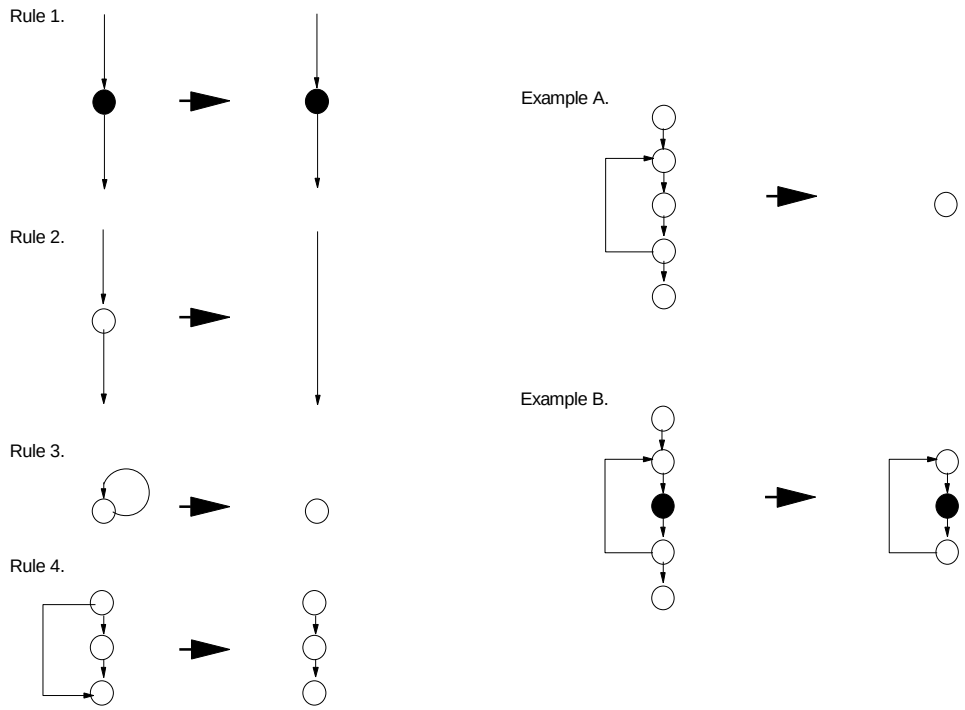

Figure 3.7.1A: Reduction rules

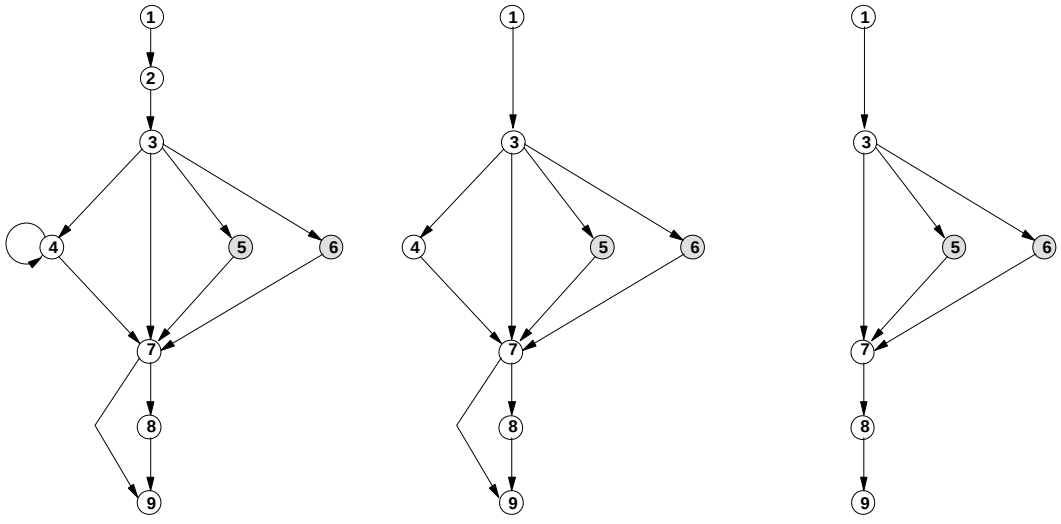

Figure 3.7.1B: Calculating module design complexity - part 1

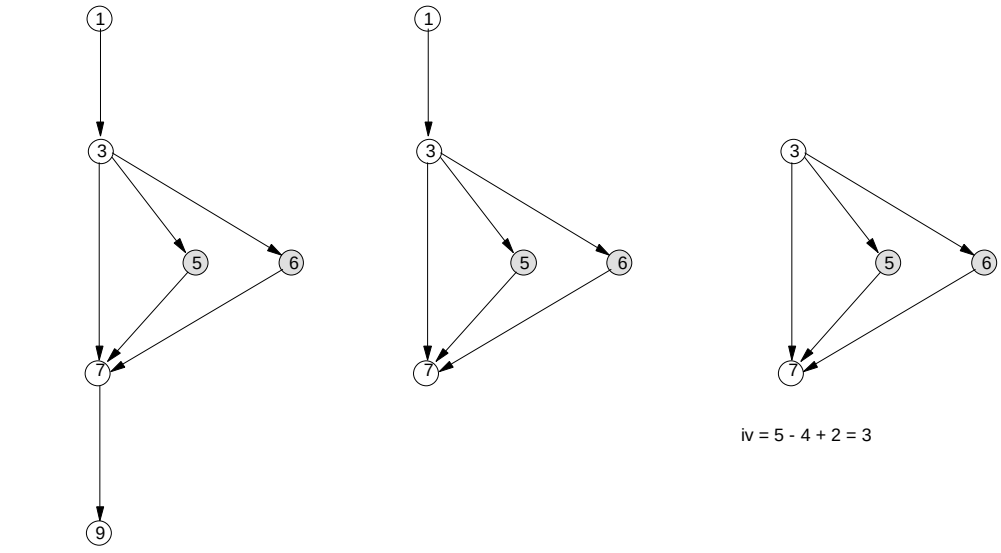

Figure 3.7.1C: Calculating module design complexity - part 2

Figures 3.7.1B and C show an example of the application of these rules. The shaded nodes 5 and 6 call lower level modules. The unshaded nodes do not. In the first reduction step, rule 2 is applied to remove node 2. Rule 3 is then applied to remove the edge returning control in the loop around node 4. Rule 2 is applied to remove node 4. Rule 4 is then applied to remove the edge branching from node 7 to node 9. Rule 2 is then applied to remove nodes 8, 9 and 1. The module design complexity is the cyclomatic complexity (see Section 3.6.1) of the reduced control graph, 3.

The design complexity, denoted as 'S_0' by McCabe, of an assembly of modules is evaluated by summing the module design complexities of each module in the assembly.

The integration complexity, denoted as 'S_1' by McCabe, of an assembly of modules, counts the number of paths through the control flow. The integration complexity of an assembly of 'N' modules is given by the formula:

$$S_1 = S_0 - N + 1$$

The integration complexity of N modules each containing no branches is therefore 1.

The testability of a design is measured by evaluating its integration complexity. Formally the integration complexity depends on measuring the module design complexity of each module. During architectural design, the

full control graph of each constituent module will not usually be available. However sufficient information should be available to define the module design complexity without knowing all the module logic.

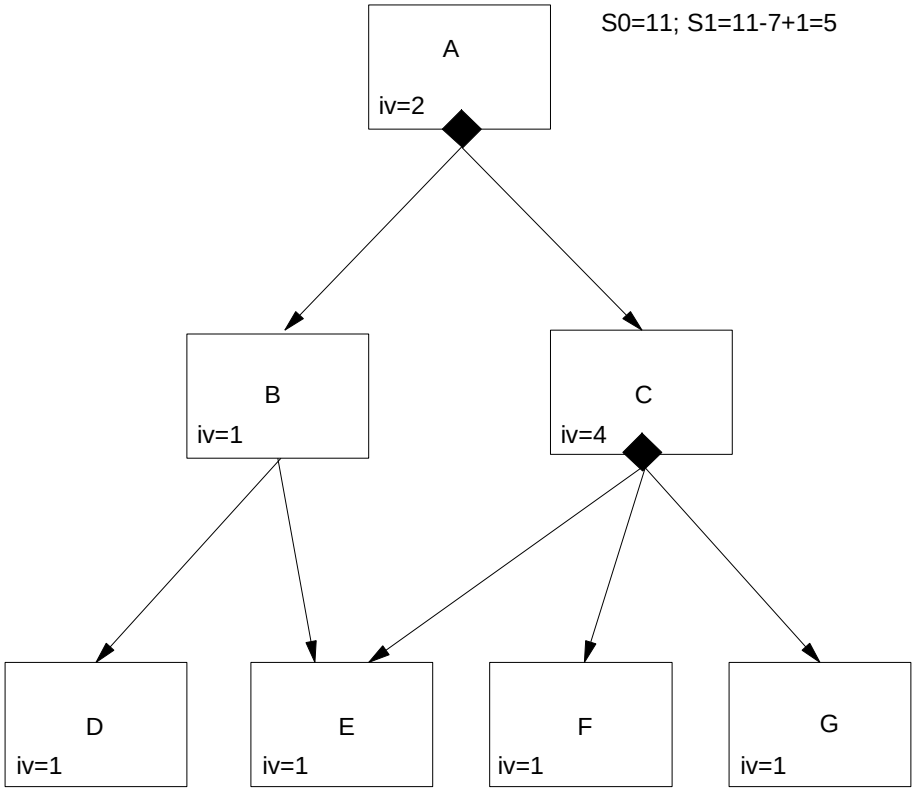

Figure 3.7.1D: Estimating integration complexity from a structure chart

Figure 3.7.1D shows a structure chart illustrating how S_1 can be evaluated just from knowing the conditions that govern the invocation of each component, i.e. the control flow. Boxes in the chart correspond to design components, and arrows mark transfer of the control flow. A diamond indicates that the transfer is conditional. The control flow is defined to be:

- component A calls either component B or component C;
- component B sequentially calls component D and then component E;
- component C calls either component E or component F or component G or none of them.

The integration complexity of the design in shown Figure 3.7.1D is therefore 5.

3.7.2 Control flow testing

Just as cyclomatic complexity measures the number of test cases required to cover every branch of a module, integration complexity measures the number of tests required to cover all the control flow paths. Structured integration testing can therefore be used for verifying that all the control flows defined in the ADD have been implemented (DD08).

3.7.3 Design integration testing method

The design integration testing method is used in structured integration testing to enable the test designer to decide what control flow paths should be tested. The test designer should:

- evaluate the integration complexity, S_1 for the N modules in the design (see Section 3.7.1);
- construct a blank matrix of dimension S_1 rows by N columns, called the 'integration path test matrix';
- mark each column, the module of which is conditionally called, with a 'p' (for predicate), followed by a sequence number for the predicate;
- fill in the matrix with 1's or 0's to show whether the module is called in each test case.

The example design shown in Figure 3.7.1D has an integration complexity of 5. The five integration test cases to verify the control flow are shown in Table 3.7.3.

Case	A	P1 B	P2 C	D	P3 E	P4 F	P5 G	Control flow path
1	1	1	0	1	1	0	0	A calls B; B calls D and E
2	1	0	1	0	0	0	0	A calls C and then returns
3	1	0	1	0	1	0	0	A calls C and then C calls E
4	1	0	1	0	0	1	0	A calls C and then C calls F
5	1	0	1	0	0	0	1	A calls C and then C calls G

Table 3.7.3: Example Integration Path Test Matrix

In Table 3.7.3, P1 and P2 are two predicates (contained within module A) that decide whether B or C are called. Similarly, P3, P4 and P5 are three predicates (contained within module C) that decide whether E, F or G are called.

This page is intentionally left blank.

CHAPTER 4
SOFTWARE VERIFICATION AND VALIDATION TOOLS

4.1 INTRODUCTION

A software tool is a 'computer program used in the development, testing, analysis, or maintenance of a program or its documentation' [Ref 6]. Software tools, more commonly called 'Computer Aided Software Engineering' (CASE) tools enhance the software verification and validation process by:

- reducing the effort needed for mechanical tasks, therefore increasing the amount of software verification and validation work that can be done;
- improving the accuracy of software verification and validation (for example, measuring test coverage is not only very time consuming, it is also very difficult to do accurately without tools).

Both these benefits result in improved software quality and productivity.

ESA PSS-05-0 defines the primary software verification and validation activities as:

- reviewing;
- tracing;
- formal proof;
- testing;
- auditing.

The following sections discuss tools for supporting each activity. This chapter does not describe specific products, but contains guidelines for their selection.

4.2 TOOLS FOR REVIEWING

4.2.1 General administrative tools

General administrative tools may be used for supporting reviewing, as appropriate. Examples are:

- word processors that allow commenting on documents;

- word processors that can show changes made to documents;
- electronic mail systems that support the distribution of review items;
- notes systems that enable communal commenting on a review item;
- conferencing systems that allow remote participation in reviews.

4.2.2 Static analysers

Static analysis is the process of evaluating a system or component based on its form, structure, content or documentation [Ref 6]. Reviews, especially software inspections, may include activities such as:

- control flow analysis to find errors such as unreachable code, endless loops, violations of recursion conventions, and loops with multiple entry and exit points;
- data-use analysis to find errors such as data used before initialisation, variables declared but not used, and redundant writes;
- range-bound analysis to find errors such as array indices outside the boundaries of the array;
- interface analysis to find errors such as mismatches in argument lists between called modules and calling modules;
- information flow analysis to check the dependency relations between input and output;
- verification of conformance to language standards (e.g. ANSI C);
- verification of conformance to project coding standards, such as departures from naming conventions;
- code volume analysis, such as counts of the numbers of modules and the number of lines of code in each module;
- complexity analysis, such as measurements of cyclomatic complexity and integration complexity.

Tools that support one or more of these static analysis activities are available and their use is strongly recommended.

Compilers often perform some control flow and data-use analysis. Some compile/link systems, such as those for Ada and Modula-2, automatically do interface analysis to enforce the strong type checking demanded by the language standards. Compilers for many languages, such as FORTRAN and C, do not do any interface analysis. Static analysis tools are especially necessary when compilers do not check control flow, data

flow, range bounds and interfaces. The use of a static analyser should be an essential step in C program development, for example.

Static analysers for measuring complexity and constructing call graphs are essential when the Structured Testing method is used for unit testing (see Section 4.5.1). Producing call graphs at higher levels (i.e. module tree and program tree) is also of use in the review and testing of the detailed design and architectural design.

4.2.3 Configuration management tools

Reviewing is an essential part of the change control process and therefore some configuration management tools also support review processes. The preparation and tracking of RIDs and SPRs can be supported by database management systems for example. See ESA PSS-05-09, 'Guide to Software Configuration Management' [Ref 4].

4.2.4 Reverse engineering tools

Although reverse engineering tools are more commonly used during maintenance to enable code to be understood, they can be used during development to permit verification that 'as-built' conforms to 'as-designed'. For example the as-built structure chart of a program can be generated from the code by means of a reverse engineering tool, and then compared with the as-designed structure chart in the ADD or DDD.

4.3 TOOLS FOR TRACING

The basic tracing method is to:

- uniquely identify the items to be tracked;
- record relationships between items.

ESA PSS-05-0 states that cross-reference matrices must be used to record relationships between:

- user requirements and software requirements;
- software requirements and architectural design components;
- software requirements and detailed design components.

In addition, test designs should be traced to software components and requirements.

Tracing tools should:

- ensure that identifiers obey the naming conventions of the project;
- ensure that identifiers are unique;
- allow attributes to be attached to the identified items such as the need, priority, or stability of a requirement, or the status of a software component (e.g. coded, unit tested, integration tested, system tested, acceptance tested);
- record all instances where an identified part of a document references an identified part of another document (e.g. when software requirement SR100 references user requirement UR49, the tool should record the relationship SR100-UR49);
- accept input in a variety of formats;
- store the traceability records (e.g. a table can be used to store a traceability matrix);
- allow easy querying of the traceability records;
- allow the extraction of information associated with a specific traceability record (e.g. the user requirement related to a software requirement);
- generate traceability reports (e.g. cross-reference matrices);
- inform users of the entities that may be affected when a related entity is updated or deleted;
- be integrated with the project repository so that the relationship database is automatically updated after a change to a document or a software component;
- provide access to the change history of a review item so that the consistency of changes can be monitored.

In summary, tracing tools should allow easy and efficient navigation through the software.

Most word processors have indexing and cross-referencing capabilities. These can also support traceability; for example, traceability from software requirements to user requirements can be done by creating index entries for the user requirement identifiers in the SRD. The disadvantage of this approach is that there is no system-wide database of relationships. Consequently dedicated tracing tools, normally based upon commercial database management systems, are used to build relationship databases. Customised frontends are required to make tracing tool capabilities easily accessible. The database may form part of, or be integrated with, the software development environment's repository. Tracing

tools should accept review items (with embedded identifiers) and generate the traceability records from that input.

Traceability tool functionality is also provided by 'requirements engineering' tools, which directly support the requirements creation and maintenance process. Requirements engineering tools contain all the specific requirements information in addition to the information about the relationships between the requirements and the other parts of the software.

4.4 TOOLS FOR FORMAL PROOF

Formal Methods (e.g. Z and VDM) are discussed in ESA PSS-05-03, 'Guide to the Software Requirements Definition Phase' [Ref 2]. This guide also discusses the criteria for the selection of CASE tools for software requirements definition. These criteria should be used for the selection of tools to support Formal Methods.

Completely automated program verification is not possible without a formally defined programming language. However subsets of some languages (e.g. Pascal, Ada) can be formally defined. Preprocessors are available that automatically check that the code is consistent with assertion statements placed in it. These tools can be very effective in verifying small well-structured programs.

Semantics is the relationship of symbols and groups of symbols to their meaning in a given language [Ref 6]. In software engineering, semantic analysis is the symbolic execution of a program by means of algebraic symbols instead of test input data. Semantic analysers use a source code interpreter to substitute algebraic symbols into the program variables and present the results as algebraic formulae [Ref 17]. Like program verifiers, semantic analysers may be useful for the verification of small well-structured programs.

4.5 TOOLS FOR TESTING

Testing involves many activities, most of which benefit from tool support. Figure 4.5 shows what test tools can be used to support the testing activities defined in Section 2.6.

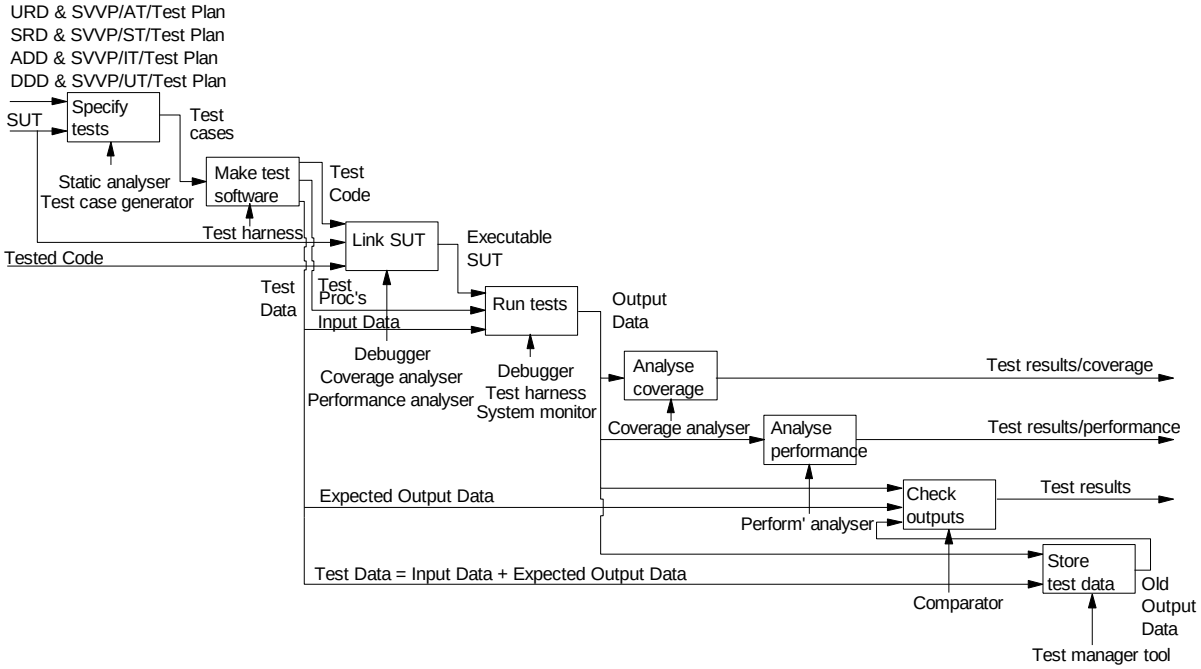

Figure 4.5: Testing Tools

The following paragraphs step through each activity depicted in Figure 4.5, discussing the tools indicated by the arrows into the bottom of the activity boxes.

1. The 'specify tests' activity may use the Structured Testing Method and Structured Integration Testing methods (see Sections 3.6 and 3.7). Static analysers should be used to measure the cyclomatic complexity and integration complexity metrics. This activity may also use the equivalence partitioning method for defining unit test cases. Test case generators should support this method.

2. The 'make test software' activity should be supported with a test harness tool. At the unit level, test harness tools that link to the SUT act as drivers for software modules, providing all the user interface functions required for the control of unit tests. At the integration and system levels, test harness tools external to the SUT (e.g. simulators) may be required. Test procedures should be encoded in scripts that can be read by the test harness tool.

3. The 'link SUT' activity links the test code, SUT, and existing tested code, producing the executable SUT. Debuggers, coverage analysers and performance analysers may also be built into the executable SUT. Coverage analysers and performance analysers (collectively known as 'dynamic analysers') require 'instrumentation' of the code so that post-test analysis data can be collected.

4. The 'run tests' activity executes the tests according to the test procedures, using the input data. Test harnesses permit control of the test. These should have a capture and playback capability if the SUT has an interactive user interface. White-box test runs may be conducted under the control of a debugger. This permits monitoring of each step of execution, and immediate detection of defects that cause test failures. System monitoring tools can be used to study program resource usage during tests.
5. The 'analyse coverage' activity checks that the tests have executed the parts of the SUT they were intended to. Coverage analysers show the paths that have been executed, helping testers keep track of coverage.
6. The 'analyser performance' activity should be supported with a performance analyser (also called a 'profiler'). These examine the performance data collected during the test run and produce reports of resource usage at component level and statement level.
7. The 'check outputs' activity should be supported with a comparator. This tool can automatically compare the test output data with the outputs of previous tests or the expected output data.
8. The 'store test data' activity should be supported by test manager tools that store the test scripts, test input and output data for future use.

Tool support for testing is weakest in the area of test design and test case generation. Tool support is strongest in the area of running tests, analysing the results and managing test software. Human involvement is required to specify the tests, but tools should do most of the tedious, repetitive work involved in running them. Automated support for regression testing is especially good. In an evolving system this is where it is most needed. The bulk of software costs usually accrue during the operations and maintenance phase of a project. Often this is due to the amount of regression testing required to verify that changes have not introduced faults in the system.

The following subsections discuss the capabilities of test tools in more detail.

4.5.1 Static analysers

Static analysers that measure complexity are needed to support the Structured Testing method and for checking adherence to coding standards. These tools may also support review activities (see Section 4.2.2). The measure of complexity defines the minimum number of test

cases required for full branch coverage. Static analysers should also produce module control graphs to support path definition.

The output of a static analyser should be readable by the test harness tool so that checks on values measured during static analysis can be included in test scripts (e.g. check that the complexity of a module has not exceeded some threshold, such as 10). This capability eases regression testing.

4.5.2 Test case generators

A test case generator (or automated test generator) is 'a software tool that accepts as input source code, test criteria, specifications, or data structure definitions; uses these inputs to generate test input data; and, sometimes, determines the expected results' [Ref 6].

The test case generation methods of equivalence partitioning and boundary value analysis can be supported by test case generators. These methods are used in unit testing. Automated test case generation at integration, system and acceptance test level is not usually possible.

Expected output data need to be in a form usable by comparators.

4.5.3 Test harnesses

A test harness (or test driver) is a 'software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results '[Ref 6]. Test harness tools should:

- provide a language for programming test procedures (i.e. a script language);
- generate stubs for software components called by the SUT;
- not require any modifications to the SUT;
- provide a means of interactive control (for debugging);
- provide a batch mode (for regression testing);
- enable stubs to be informed about the test cases in use, so that they will read the appropriate test input data;
- execute all test cases required of a unit, subsystem or system;
- handle exceptions, so that testing can continue after test failures;
- record the results of all test cases in a log file;
- check all returns from the SUT;
- record in the test results log whether the test was passed or failed;
- be usable in development and target environments.

A test harness is written as a driver for the SUT. The harness is written in a scripting language that provides module, sequence, selection and iteration constructs. Scripting languages are normally based upon a standard programming language. The ability to call one script module from another, just as in a conventional programming language, is very important as test cases may only differ in their input data. The scripting language should include directives to set up, execute, stop and check the results of each test case. Software components that implement these directives are provided as part of the test harness tool. Test harnesses may be compiled and linked with the SUT, or exist externally as a control program.

Test harness tools can generate input at one or more of the following levels [Ref 19]:

- application software level, either as input files or call arguments;
- operating system level, for example as X-server messages;
- hardware level, as keyboard or mouse input.

Test harness tools should capture output at the level at which they input it. It should be possible to generate multiple input and output data streams. This is required in stress tests (e.g. multiple users logging on simultaneously) and security tests (e.g. resource locking).

A synchronisation mechanism is required to co-ordinate the test harness and the software under test. During unit testing this is rarely a problem, as the driver and the SUT form a single program. Synchronisation may be difficult to achieve during integration and system testing, when the test harness and SUT are separate programs. Synchronisation techniques include:

- recording and using the delays of the human operators;
- waiting for a specific output (i.e. a handshake technique);
- using an algorithm to decide the wait time;
- monitoring keyboard lock indicators.

The first technique is not very robust. The other techniques are preferable.

For interactive testing, the most efficient way to define the test script is to capture the manually generated input and store it for later playback. For graphical user interfaces this may be the only practical method.

Graphical user interfaces have window managers to control the use of screen resources by multiple applications. This can cause problems to test harness tools because the meaning of a mouse click depends upon what is beneath the cursor. The window manager decides this, not the test harness. This type of problem can mean that test scripts need to be frequently modified to cope with changes in layout. Comparators that can filter out irrelevant changes in screen layout (see Section 4.5.7) reduce the problems of testing software with a graphical user interface.

4.5.4 Debuggers

Debuggers are used in white-box testing for controlling the execution of the SUT. Capabilities required for testing include the ability to:

- display and interact with the tester using the symbols employed in the source code;
- step through code instruction-by-instruction;
- set watch points on variables;
- set break points;
- maintain screen displays of the source code during test execution;
- log the session for inclusion in the test results.

Session logs can act as test procedure files for subsequent tests and also as coverage reports, since they describe what happened.

4.5.5 Coverage analysers

Coverage analysis is the process of:

- instrumenting the code so that information is collected on the parts of it that are executed when it is run;
- analysing the data collected to see what parts of the SUT have been executed.

Instrumentation of the code should not affect its logic. Any effects on performance should be easy to allow for. Simple coverage analysers will just provide information about statement coverage, such as indications of the program statements executed in one run. More advanced coverage analysers can:

- sum coverage data to make reports of total coverage achieved in a series of runs;
- display control graphs, so that branch coverage and path coverage can be monitored;

- output test coverage information so that coverage can be checked by the test harness, therefore aiding regression testing;
- operate in development and target environments.

4.5.6 Performance analysers

Performance analysis is the process of:

- instrumenting the code so that information is collected about resources used when it is run;
- analysing the data collected to evaluate resource utilisation;
- optimising performance.

Instrumentation of the code should not affect the measured performance. Performance analysers should provide information on a variety of metrics such as:

- CPU usage by each line of code;
- CPU usage by each module;
- memory usage;
- input/output volume.

Coverage analysers are often integrated with performance analysers to make 'dynamic analysers'.

4.5.7 Comparators

A comparator is a 'software tool used to compare two computer programs, files, or sets of data to identify commonalities and differences' [Ref 6]. Differencing tools are types of comparators. In testing, comparators are needed to compare actual test output data with expected test output data.

Expected test output data may have originated from:

- test case generators;
- previous runs.

Comparators should report about differences between actual and expected output data. It should be possible to specify tolerances on differences (floating point operations can produce slightly different results each time they are done). The output of comparators should be usable by test harness tools, so that differences can be used to flag a test failure. This makes regression testing more efficient.

Screen comparators [Ref 19] are useful for testing interactive software. These may operate at the character or bitmap level. The ability to select parts of the data, and attributes of the data, for comparison is a key requirement of a screen comparator. Powerful screen comparison capabilities are important when testing software with a graphical user interface.

Some comparators may be run during tests. When a difference is detected the test procedure is suspended. Other comparators run off-line, checking for differences after the test run.

4.5.8 Test management tools

All the software associated with testing: test specifications, SUT, drivers, stubs, scripts, tools, input data, expected output data and output data must be placed under configuration management (SCM01). The configuration management system is responsible for identifying, storing, controlling and accounting for all configuration items.

Test data management tools provide configuration management functions for the test data and scripts. Specifically designed for supporting testing, they should:

- enable tests to be set up and run with the minimum of steps;
- automatically manage the storage of outputs and results;
- provide test report generation facilities;
- provide quality and reliability statistics;
- provide support for capture and playback of input and output data during interactive testing.

Test managers are essential for efficient regression testing.

CHAPTER 5
THE SOFTWARE VERIFICATION AND VALIDATION PLAN

5.1 INTRODUCTION

All software verification and validation activities must be documented in the Software Verification and Validation Plan (SVVP) (SVV04). The SVVP is divided into seven sections that contain the verification plans for the SR, AD and DD phases and the unit, integration, system and acceptance test specifications. Figure 5.1A summarises when and where each software verification and validation activity is documented in the SVVP.

PHASE / SVVP SECTION	USER REQUIREMENTS DEFINITION	SOFTWARE REQUIREMENTS DEFINITION	ARCHITECTURAL DESIGN	DETAILED DESIGN AND PRODUCTION	TRANSFER
SVVP/SR	SR Phase Plan				
SVVP/AD		AD Phase Plan			
SVVP/DD			DD Phase Plan		
SVVP/AT	AT Plan			AT Designs AT Cases AT Procedures	AT Reports
SVVP/ST		ST Plan		ST Designs ST Cases ST Procedures ST Reports	
SVVP/IT			IT Plan	IT Designs IT Cases IT Procedures IT Reports	
SVVP/UT				UT Plan UT Designs UT Cases UT Procedures UT Reports	

Figure 5.1A: Life cycle production of SVV documentation

Each row of Figure 5.1A corresponds to a section of the SVVP and each column entry corresponds to a subsection. For example, the entry 'ST Plan' in the SVVP/ST row means that the System Test Plan is drawn up in the SR phase and placed in the SVVP section 'System Tests', subsection 'Test Plan'.

The SVVP must ensure that the verification activities:

- are appropriate for the degree of criticality of the software (SVV05);
- meet the verification and acceptance testing requirements (stated in the SRD) (SVV06);
- verify that the product will meet the quality, reliability, maintainability and safety requirements (stated in the SRD) (SVV07);
- are sufficient to assure the quality of the product (SVV08).

The table of contents for the verification sections is derived from the IEEE Standard for Verification and Validation Plans (ANSI/IEEE Std 1012-1986). For the test sections it is derived from the IEEE Standard for Software Test Documentation (ANSI/IEEE Std 829-1983).

The relationship between test specifications, test plans, test designs, test cases, test procedures and test results may sometimes be a simple hierarchy but usually it will not be. Figure 5.1B shows the relationships between the sections and subsections of the SVVP. Sections are shown in boxes. Relationships between sections are shown by lines labelled with the verb in the relationship (e.g. 'contains'). The one-to-one relationships are shown by a plain line and one-to-many relationships are shown by the crow's feet.

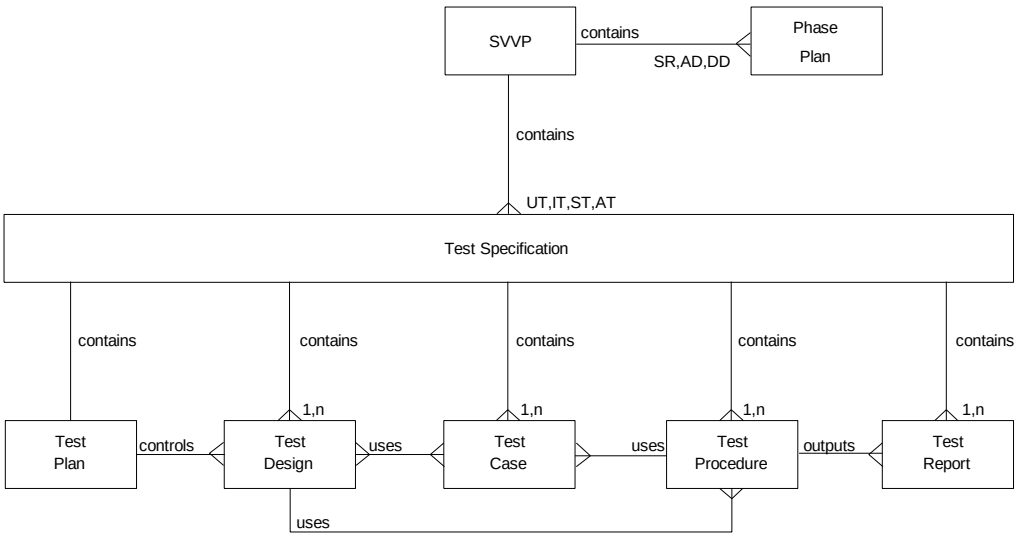

Figure 5.1B: Relationships between sections of the SVVP

Figure 5.1B illustrates that a test plan controls the test design process, defining the software items to be tested. Test designs define the features of each software item to be tested, and specify the test cases and test procedures that will be used to test the features. Test cases may be

used by many test designs and many test procedures. Each execution of a test procedure produces a new set of test results.

Software verification and validation procedures should be easy to follow, efficient and wherever possible, reusable in later phases. Poor test definition and record keeping can significantly reduce the maintainability of the software.

The key criterion for deciding the level of documentation of testing is repeatability. Tests should be sufficiently documented to allow repetition by different people, yet still yield the same results for the same software. The level of test documentation depends very much upon the software tools used to support testing. Good testing tools should relieve the developer from much of the effort of documenting tests.

5.2 STYLE

The SVVP should be plain and concise. The document should be clear, consistent and modifiable.

Authors should assume familiarity with the purpose of the software, and not repeat information that is explained in other documents.

5.3 RESPONSIBILITY

The developer is normally responsible for the production of the SVVP. The user may take responsibility for producing the Acceptance Test Specification (SVVP/AT), especially when the software is to be embedded in a larger system.

5.4 MEDIUM

It is usually assumed that the SVVP is a paper document. The SVVP could be distributed electronically to participants with the necessary equipment.

5.5 SERVICE INFORMATION

The SR, AD, DD, UT, IT, ST and AT sections of the SVVP are produced at different times in a software project. Each section should be kept separately under configuration control and contain the following service information:

a - Abstract
b - Table of Contents
c - Document Status Sheet
d - Document Change records made since last issue

5.6 CONTENT OF SVVP/SR, SVVP/AD & SVVP/DD SECTIONS

These sections define the review, proof and tracing activities in the SR, AD and DD phases of the lifecycle. While the SPMP may summarise these activities, the SVVP should provide the detailed information. For example the SPMP may schedule an AD/R, but this section of the SVVP defines the activities for the whole AD phase review process.

These sections of the SVVP should avoid repeating material from the standards and guidelines and instead define how the procedures will be applied.

ESA PSS-05-0 recommends the following table of contents for the SVVP/SR, SVVP/AD and SVVP/DD sections:

1 Purpose
2 Reference Documents
3 Definitions
4. Verification overview
   4.1 Organisation
   4.2 Master schedule
   4.3 Resources summary
   4.4 Responsibilities
   4.5 Tools, techniques and methods
5. Verification Administrative Procedures
   5.1 Anomaly reporting and resolution
   5.2 Task iteration policy
   5.3 Deviation policy
   5.4 Control procedures
   5.5 Standards, practices and conventions

6. Verification Activities
   6.1 Tracing[1]
   6.2 Formal proofs
   6.3 Reviews
7 Software Verification Reporting

Additional material should be inserted in additional appendices. If there is no material for a section then the phrase 'Not Applicable' should be inserted and the section numbering preserved.

1 PURPOSE

This section should:

- briefly define the purpose of this part of the SVVP, stating the part of the lifecycle to which it applies;
- identify the software project for which the SVVP is written;
- identify the products to be verified, and the specifications that they are to be verified against;
- outline the goals of verification and validation;
- specify the intended readers of this part of the SVVP.

2 REFERENCE DOCUMENTS

This section should provide a complete list of all the applicable and reference documents, such as the ESA PSS-05 series of standards and guides. Each document should be identified by title, author and date. Each document should be marked as applicable or reference. If appropriate, report number, journal name and publishing organisation should be included.

3 DEFINITIONS

This section should provide the definitions of all terms, acronyms, and abbreviations used in the plan, or refer to other documents where the definitions can be found.

---

[1] Note that in ESA PSS-05-0 Issue 2 this section is called 'traceability matrix template'

4 VERIFICATION OVERVIEW

This section should describe the organisation, schedule, resources, responsibilities, tools, techniques and methods necessary to perform reviews, proofs and tracing.

4.1 Organisation

This section should describe the organisation of the review, proofs and tracing activities for the phase. Topics that should be included are:

- roles;
- reporting channels;
- levels of authority for resolving problems;
- relationships to the other activities such as project management, development, configuration management and quality assurance.

The description should identify the people associated with the roles. Elsewhere the plan should only refer to roles.

4.2 Master schedule

This section should define the schedule for the review, proofs and tracing activities in the phase.

Reviews of large systems should be broken down by subsystem. In the DD phase, critical design reviews of subsystems should be held when the subsystem is ready, not when all subsystems have been designed.

4.3 Resources summary

This section should summarise the resources needed to perform reviews, proofs and tracing such as staff, computer facilities, and software tools.

4.4 Responsibilities

This section should define the specific responsibilities associated with the roles described in section 4.1.

4.5 Tools, techniques and methods

This section should identify the software tools, techniques and methods used for reviews, proofs and tracing in the phase. Training plans for the tools, techniques and methods may be included.

5 VERIFICATION ADMINISTRATIVE PROCEDURES

5.1 Anomaly reporting and resolution

The Review Item Discrepancy (RID) form is normally used for reporting and resolving anomalies found in documents and code submitted for formal review. The procedure for handling this form is normally described in Section 4.3.2 of the SCMP, which should be referenced here and not repeated. This section should define the criteria for activating the anomaly reporting and resolution process.

5.2 Task iteration policy

This section should define the criteria for deciding whether a task should be repeated when a change has been made. The criteria may include assessments of the scope of a change, the criticality of the function(s) affected, and any quality effects.

5.3 Deviation policy

This section should describe the procedures for deviating from the plan, and define the levels of authorisation required for the approval of deviations. The information required for deviations should include task identification, deviation rationale and the effect on software quality.

5.4 Control procedures

This section should identify the configuration management procedures of the products of review, proofs and tracing. Adequate assurance that they are secure from accidental or deliberate alteration is required.

5.5 Standards, practices and conventions

This section should identify the standards, practices and conventions that govern review, proof and tracing tasks, including internal organisational standards, practices and policies (e.g. practices for safety-critical software).

6 VERIFICATION ACTIVITIES

This section should describe the procedures for review, proof and tracing activities.

6.1 Tracing

This section should describe the procedures for tracing each part of the input products to the output products, and vice-versa.

6.2 Formal proofs

This section should define or reference the methods and procedures used (if any) for proving theorems about the behaviour of the software.

6.3 Reviews

This section should define or reference the methods and procedures used for technical reviews, walkthroughs, software inspections and audits.

This section should list the reviews, walkthroughs and audits that will take place during the phase and identify the roles of the people participating in them.

This section should not repeat material found in the standards and guides, but should specify project-specific modifications and additions.

7 SOFTWARE VERIFICATION REPORTING

This section should describe how the results of implementing the plan will be documented. Types of reports might include:

- summary report for the phase;
- technical review report;
- walkthrough report;
- audit report.

RIDs should be attached to the appropriate verification report.

5.7 CONTENT OF SVVP/UT, SVVP/IT, SVVP/ST & SVVP/AT SECTIONS

The SVVP contains four sections dedicated to each test development phase. These sections are called:

- Unit Test Specification (SVVP/UT);
- Integration Test Specification (SVVP/IT);
- System Test Specification (SVVP/ST);
- Acceptance Test Specification (SVVP/AT).

ESA PSS-05-0 recommends the following table of contents for each test section of the SVVP.

1 Test Plan
- 1.1 Introduction
- 1.2 Test items
- 1.3 Features to be tested
- 1.4 Features not to be tested
- 1.5 Approach
- 1.6 Item pass/fail criteria
- 1.7 Suspension criteria and resumption requirements
- 1.8 Test deliverables
- 1.9 Testing tasks
- 1.10 Environmental needs
- 1.11 Responsibilities
- 1.12 Staffing and training needs
- 1.13 Schedule
- 1.14 Risks and contingencies
- 1.15 Approvals

2 Test Designs (for each test design...)
- 2.n.1 Test design identifier
- 2.n.2 Features to be tested
- 2.n.3 Approach refinements
- 2.n.4 Test case identification
- 2.n.5 Feature pass/fail criteria

3 Test Case Specifications (for each test case...)
- 3.n.1 Test case identifier
- 3.n.2 Test items
- 3.n.3 Input specifications
- 3.n.4 Output specifications
- 3.n.5 Environmental needs
- 3.n.6 Special procedural requirements
- 3.n.7 Intercase dependencies

4 Test Procedures (for each test case...)
- 4.n.1 Test procedure identifier
- 4.n.2 Purpose
- 4.n.3 Special requirements
- 4.n.4 Procedure steps

5 Test Reports (for each execution of a test procedure ...)
- 5.n.1 Test report identifier
- 5.n.2 Description
- 5.n.3 Activity and event entries

1 TEST PLAN

1.1 Introduction

This section should summarise the software items and software features to be tested. A justification of the need for testing may be included.

1.2 Test items

This section should identify the test items. References to other software documents should be supplied to provide information about what the test items are supposed to do, how they work, and how they are operated.

Test items should be grouped according to release number when delivery is incremental.

1.3 Features to be tested

This section should identify all the features and combinations of features that are to be tested. This may be done by referencing sections of requirements or design documents.

References should be precise yet economical, e.g:

- 'the acceptance tests will cover all requirements in the User Requirements Document except those identified in Section 1.4';
- 'the unit tests will cover all modules specified in the Detailed Design Document except those modules listed in Section 1.4'.

Features should be grouped according to release number when delivery is incremental.

1.4 Features not to be tested

This section should identify all the features and significant combinations of features that are not to be tested, and why.

1.5 Approach

This section should specify the major activities, methods (e.g. structured testing) and tools that are to be used to test the designated groups of features.

Activities should be described in sufficient detail to allow identification of the major testing tasks and estimation of the resources and time needed for the tests. The coverage required should be specified.

1.6 Item pass/fail criteria

This section should specify the criteria to be used to decide whether each test item has passed or failed testing.

1.7 Suspension criteria and resumption requirements

This section should specify the criteria used to suspend all, or a part of, the testing activities on the test items associated with the plan.

This section should specify the testing activities that must be repeated when testing is resumed.

1.8 Test deliverables

This section should identify the items that must be delivered before testing begins, which should include:

- test plan;
- test designs;
- test cases;
- test procedures;
- test input data;
- test tools.

This section should identify the items that must be delivered when testing is finished, which should include:

- test reports;
- test output data;
- problem reports.

1.9 Testing tasks

This section should identify the set of tasks necessary to prepare for and perform testing. This section should identify all inter-task dependencies and any special skills required.

Testing tasks should be grouped according to release number when delivery is incremental.

1.10 Environmental needs

This section should specify both the necessary and desired properties of the test environment, including:

- physical characteristics of the facilities including hardware;
- communications software;
- system software;
- mode of use (i.e. standalone, networked);
- security;
- test tools.

Environmental needs should be grouped according to release number when delivery is incremental.

1.11 Responsibilities

This section should identify the groups responsible for managing, designing, preparing, executing, witnessing, and checking tests.

Groups may include developers, operations staff, user representatives, technical support staff, data administration staff, independent verification and validation personnel and quality assurance staff.

1.12 Staffing and training needs

This section should specify staffing needs according to skill. Identify training options for providing necessary skills.

1.13 Schedule

This section should include test milestones identified in the software project schedule and all item delivery events, for example:

- programmer delivers unit for integration testing;
- developers deliver system for independent verification.

This section should specify:

- any additional test milestones and state the time required for each testing task;
- the schedule for each testing task and test milestone;
- the period of use for all test resources (e.g. facilities, tools, staff).

1.14 Risks and contingencies

This section should identify the high-risk assumptions of the test plan. It should specify contingency plans for each.

1.15 Approvals

This section should specify the names and titles of all persons who must approve this plan. Alternatively, approvals may be shown on the title page of the plan.

2 TEST DESIGNS

2.n.1 Test Design identifier

The title of this section should specify the test design uniquely. The content of this section should briefly describe the test design.

2.n.2 Features to be tested

This section should identify the test items and describe the features, and combinations of features, that are to be tested.

For each feature or feature combination, a reference to its associated requirements in the item requirement specification (URD, SRD) or design description (ADD, DDD) should be included.

2.n.3 Approach refinements

This section should describe the results of the application of the methods described in the approach section of the test plan. Specifically it may define the:

- module assembly sequence (for unit testing);
- paths through the module logic (for unit testing);
- component integration sequence (for integration testing);
- paths through the control flow (for integration testing);
- types of test (e.g. white-box, black-box, performance, stress etc).

The description should provide the rationale for test-case selection and the packaging of test cases into procedures. The method for analysing test results should be identified (e.g. compare with expected output, compare with old results, proof of consistency etc).

The tools required to support testing should be identified.

2.n.4 Test case identification

This section should list the test cases associated with the design and give a brief description of each.

2.n.5 Feature pass/fail criteria

This section should specify the criteria to be used to decide whether the feature or feature combination has passed or failed.

3 TEST CASE SPECIFICATION

3.n.1 Test Case identifier

The title of this section should specify the test case uniquely. The content of this section should briefly describe the test case.

3.n.2 Test items

This section should identify the test items. References to other software documents should be supplied to help understand the purpose of the test items, how they work and how they are operated.

3.n.3 Input specifications

This section should specify the inputs required to execute the test case. File names, parameter values and user responses are possible types of input specification. This section should not duplicate information held elsewhere (e.g. in test data files).

3.n.4 Output specifications

This section should specify the outputs expected from executing the test case relevant to deciding upon pass or failure. File names and system messages are possible types of output specification. This section should not duplicate information held elsewhere (e.g. in log files).

3.n.5 Environmental needs

3.n.5.1 Hardware

This section should specify the characteristics and configurations of the hardware required to execute this test case.

3.n.5.2 Software

This section should specify the system and application software required to execute this test case.

3.n.5.3 Other

This section should specify any other requirements such as special equipment or specially trained personnel.

3.n.6 Special procedural requirements

This section should describe any special constraints on the test procedures that execute this test case.

3.n.7 Intercase dependencies

This section should list the identifiers of test cases that must be executed before this test case. The nature of the dependencies should be summarised.

4 TEST PROCEDURES

4.n.1 Test Procedure identifier

The title of this section should specify the test procedure uniquely. This section should reference the related test design.

4.n.2 Purpose

This section should describe the purpose of this procedure. A reference for each test case the test procedure uses should be given.

4.n.3 Special requirements

This section should identify any special requirements for the execution of this procedure.

4.n.4 Procedure steps

This section should include the steps described in the subsections below as applicable.

4.n.4.1 Log

This section should describe any special methods or formats for logging the results of test execution, the incidents observed, and any other events pertinent to the test.

4.n.4.2 Set up

This section should describe the sequence of actions necessary to prepare for execution of the procedure.

4.n.4.3 Start

This section should describe the actions necessary to begin execution of the procedure.

4.n.4.4 Proceed

This section should describe the actions necessary during the execution of the procedure.

4.n.4.5 Measure

This section should describe how the test measurements will be made.

4.n.4.6 Shut down

This section should describe the actions necessary to suspend testing when interruption is forced by unscheduled events.

4.n.4.7 Restart

This section should identify any procedural restart points and describe the actions necessary to restart the procedure at each of these points.

4.n.4.8 Stop

This section should describe the actions necessary to bring execution to an orderly halt.

4.n.4.9 Wrap up

This section should describe the actions necessary to terminate testing.

4.n.4.10 Contingencies

This section should describe the actions necessary to deal with anomalous events that may occur during execution.

5 TEST REPORTS

5.n.1 Test Report identifier

The title of this section should specify the test report uniquely.

5.n.2 Description

This section should identify the items being tested including their version numbers. The attributes of the environment in which testing was conducted should be identified.

5.n.3 Activity and event entries

This section should define the start and end time of each activity or event. The author should be identified.

One or more of the descriptions in the following subsections should be included.

5.n.3.1 Execution description

This section should identify the test procedure being executed and supply a reference to its specification.

The people who witnessed each event should be identified.

5.n.3.2 Procedure results

For each execution, this section should record the visually observable results (e.g. error messages generated, aborts and requests for operator action). The location of any output, and the result of the test, should be recorded.

5.n.3.3 Environmental information

This section should record any environmental conditions specific for this entry, particularly deviations from the normal.

5.8 EVOLUTION

5.8.1 UR phase

By the end of the UR review, the SR phase section of the SVVP must be produced (SVVP/SR) (SVV09). The SVVP/SR must define how to trace user requirements to software requirements so that each software requirement can be justified (SVV10). It should describe how the SRD is to be evaluated by defining the review procedures. The SVVP/SR may include specifications of the tests to be done with prototypes.

The initiator(s) of the user requirements should lay down the principles upon which the acceptance tests should be based. The developer must construct an acceptance test plan in the UR phase and document it in the SVVP (SVV11). This plan should define the scope, approach, resources and schedule of acceptance testing activities.

5.8.2 SR phase

During the SR phase, the AD phase section of the SVVP must be produced (SVVP/AD) (SVV12). The SVVP/AD must define how to trace software requirements to components, so that each software component can be justified (SVV13). It should describe how the ADD is to be evaluated by defining the review procedures. The SVVP/AD may include specifications of the tests to be done with prototypes.

During the SR Phase, the developer analyses the user requirements and may insert 'acceptance testing requirements' in the SRD. These requirements constrain the design of the acceptance tests. This must be recognised in the statement of the purpose and scope of the acceptance tests.

The planning of the system tests should proceed in parallel with the definition of the software requirements. The developer may identify 'verification requirements' for the software. These are additional constraints on the unit, integration and system testing activities. These requirements are also stated in the SRD.

The developer must construct a system test plan in the SR phase and document it in the SVVP (SVV14). This plan should define the scope, approach, resources and schedule of system testing activities.

5.8.3 AD phase

During the AD phase, the DD phase section of the SVVP must be produced (SVVP/DD) (SVV15). The SVVP/AD must describe how the DDD and code are to be evaluated by defining the review and traceability procedures (SVV16).

The developer must construct an integration test plan in the AD phase and document it in the SVVP (SVV17). This plan should describe the scope, approach, resources and schedule of intended integration tests. Note that the items to be integrated are the software components described in the ADD.

5.8.4 DD phase

In the DD phase, the SVVP sections on testing are developed as the detailed design and implementation information become available.

The developer must construct a unit test plan in the DD phase and document it in the SVVP (SVV18). This plan should describe the scope, approach, resources and schedule of the intended unit tests.

The test items are the software components described in the DDD.

The unit, integration, system and acceptance test designs must be described in the SVVP (SVV19). These should specify the details of the test approach for a software feature, or combination of software features, and identify the associated test cases and test procedures.

The unit integration, system and acceptance test cases must be described in the SVVP (SVV20). These should specify the inputs, predicted results and execution conditions for a test case.

The unit, integration, system and acceptance test procedures must be described in the SVVP (SVV21). These should provide a step-by-step description of how to carry out each test case.

The unit, integration, system and acceptance test reports must be contained in the SVVP (SVV22).

This page is intentionally left blank.

APPENDIX A
GLOSSARY

A.1 LIST OF TERMS

Definitions of SVV terms are taken from IEEE Standard Glossary of Software Engineering Terminology ANSI/IEEE Std 610.12-1990 [Ref 6]. If no suitable definition is found in the glossary, the definition is taken from a referenced text or the Concise Oxford Dictionary.

Acceptance testing

Formal testing conducted to determine whether or not a system satisfies its acceptance criteria (i.e. the user requirements) to enable the customer (i.e. initiator) to determine whether or not to accept the system [Ref 6].

Assertion

A logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution [Ref 6].

Audit

An independent examination of a work product or set of work products to assess compliance with specifications, baselines, standards, contractual agreements or other criteria [Ref 6].

Back-to-back test

Back-to-back tests execute two or more variants of a program with the same inputs. The outputs are compared, and any discrepancies are analysed to check whether or not they indicate a fault [Ref 6].

Comparator

A software tool that compares two computer programs, files, or sets of data to identify commonalities and differences [Ref 6].

Component

One of the parts that make up a system [Ref 6]. A component may be a module, a unit or a subsystem. This definition is similar to that used in Reference 1 and more general than the one in Reference 5.

Critical design review

A review conducted to verify that the detailed design of one or more configuration items satisfies specified requirements [Ref 6]. Critical design reviews must be held in the DD phase to review the detailed design of a major component to certify its readiness for implementation (DD10).

Decision table

A table used to show sets of conditions and the actions resulting from them [Ref 6].

Defect

An instance in which a requirement is not satisfied [Ref 22].

Driver

A software module that invokes and, perhaps, controls and monitors the execution of one or more other software modules [Ref 6].

Dynamic analysis

The process of evaluating a computer program based upon its behaviour during execution [Ref 6].

Formal

Used to describe activities that have explicit and definite rules of procedure (e.g. formal review) or reasoning (e.g. formal method and formal proof).

Formal review

A review that has explicit and definite rules of procedure such as a technical review, walkthrough or software inspection [Ref 1].

Inspection

A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems [Ref 6]. Same as 'software inspection'.

Integration

The process of combining software elements, hardware elements or both into an overall system [Ref 6].

Integration testing

Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them [Ref 6]. In ESA PSS-05-0, the lowest level software elements tested during integration are the lowest level components of the architectural design.

Module

A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example the input or output from a compiler or linker; also a logically separable part of a program [Ref 6].

Regression test

Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [Ref 6].

Review

A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval [Ref 6]].

Semantics

The relationships of symbols and groups of symbols to their meanings in a given language [Ref 6].

Semantic analyser

A software tool that substitutes algebraic symbols into the program variables and present the results as algebraic formulae [Ref 17].

Static analysis

The process of evaluating a system or component based on its form, structure, content or documentation [Ref 6].

Stress test

A test that evaluates a system or software component at or beyond the limits of its specified requirements [Ref 6].

System

A collection of components organised to accomplish a specific function or set of functions [Ref 6]. A system is composed of one or more subsystems.

Subsystem

A secondary or subordinate system within a larger system [Ref 6]. A subsystem is composed of one or more units.

System testing

Testing conducted on a complete, integrated system to evaluate the system ŝ compliance with its specified requirements [Ref 6].

Test

An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component [Ref 6].

Test case

A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specified requirement [Ref 6].

Test design

Documentation specifying the details of the test approach for a software feature or combination of software features and identifying associated tests [Ref 6].

Test case generator

A software tool that accepts as input source code, test criteria, specifications, or data structure definitions; uses these inputs to generate test input data, and, sometimes, determines the expected results [Ref 6].

Test coverage

The degree to which a given test or set of tests addresses all specified requirements for a given system or component [Ref 6]; the proportion of branches in the logic that have been traversed during testing.

Test harness

A software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results [Ref 6].

Test plan

A document prescribing the scope, approach resources, and schedule of intended test activities [Ref 6].

Test procedure

Detailed instructions for the setup, operation, and evaluation of the results for a given test [Ref 6].

Test report

A document that describes the conduct and results of the testing carried out for a system or component [Ref 6].

Testability

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [Ref 6].

Tool

A computer program used in the development, testing, analysis, or maintenance of a program or its documentation [Ref 6].

Tracing

The act of establishing a relationship between two or more products of the development process; for example, to establish the relationship between a given requirement and the design element that implements that requirement [Ref 6].

Unit

A separately testable element specified in the design of a computer software component [Ref 6]. A unit is composed of one or more modules.

Validation

The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [Ref 6].

Verification

The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services or documents conform to specified requirements [Ref 5].

Walkthrough

A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems [Ref 6].

A.2 LIST OF ACRONYMS

AD	Architectural Design
AD/R	Architectural Design Review
ADD	Architectural Design Document
ANSI	American National Standards Institute
AT	Acceptance Test
BSSC	Board for Software Standardisation and Control
CASE	Computer Aided Software Engineering
DCR	Document Change Record
DD	Detailed Design and production
DD/R	Detailed Design and production Review
DDD	Detailed Design and production Document
ESA	European Space Agency
ICD	Interface Control Document
IEEE	Institute of Electrical and Electronics Engineers
IT	Integration Test
PSS	Procedures, Specifications and Standards
QA	Quality Assurance
RID	Review Item Discrepancy
SCMP	Software Configuration Management Plan
SCR	Software Change Request
SMR	Software Modification Report
SPR	Software Problem Report
SR	Software Requirements
SR/R	Software Requirements Review
SRD	Software Requirements Document
ST	System Test
SUT	Software Under Test
SVV	Software Verification and Validation
SVVP	Software Verification and Validation Plan
UR	User Requirements
UR/R	User Requirements Review
URD	User Requirements Document
UT	Unit Test

This page is intentionally left blank.

APPENDIX B
REFERENCES

1. ESA Software Engineering Standards, ESA PSS-05-0 Issue 2, February 1991.
2. Guide to the Software Requirements Definition Phase, ESA PSS-05-03, Issue 1, October 1991.
3. Guide to the Detailed Design and Production Phase, ESA PSS-05-05, Issue 1, May 1992.
4. Guide to Software Configuration Management, ESA PSS-05-09, Issue 1, November 1992.
5. ANSI/ASQC A3-1978, Quality Systems Terminology
6. IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12-1990.
7. IEEE Standard for Software Test Documentation, ANSI/IEEE Std 829-1983.
8. IEEE Standard for Software Unit Testing, ANSI/IEEE Std 1008-1987.
9. IEEE Standard for Software Verification and Validation Plans, ANSI/IEEE Std 1012-1986.
10. IEEE Standard for Software Reviews and Audits, ANSI/IEEE Std 1028-1988.
11. Managing the Software Process, Watts S. Humphrey, SEI Series in Software Engineering, Addison-Wesley, August 1990.
12. Software Testing Techniques, B.Beizer, Van Nostrand Reinhold, 1983.
13. Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, T.J.McCabe, National Bureau of Standards Special Publication 500-99, 1982.
14. The Art of Software Testing, G.J.Myers, Wiley-Interscience, 1979.
15. Design Complexity Measurement and Testing, T.J.McCabe and C.W.Butler, Communications of the ACM, Vol 32, No 12, December 1989.
16. Software Engineering, I.Sommerville, Addison-Wesley, 1992.
17. The STARTs Guide - a guide to methods and software tools for the construction of large real-time systems, NCC Publications, 1987.
18. Engineering software under statistical quality control, R.H. Cobb and H.D.Mills, IEEE Software, 7 (6), 1990.

19. Dynamic Testing Tools, a Detailed Product Evaluation, S.Norman, Ovum, 1992.
20. Managing Computer Projects, R.Gibson, Prentice-Hall, 1992.
21. Design and Code Inspections to Reduce Errors in Program Development, M.E.Fagan, IBM Systems Journal, No 3, 1976
22. Advances in Software Inspections, M.E. Fagan, IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986.
23. The Cleanroom Approach to Quality Software Development, Dyer, Wiley, 1992.
24. System safety requirements for ESA space systems and related equipment, ESA PSS-01-40 Issue 2, September 1988, ESTEC

APPENDIX C
MANDATORY PRACTICES

This appendix is repeated from ESA PSS-05-0, appendix D.10.

SVV01 Forward traceability requires that each input to a phase shall be traceable to an output of that phase.

SVV02 Backward traceability requires that each output of a phase shall be traceable to an input to that phase.

SVV03 Functional and physical audits shall be performed before the release of the software.

SVV04 All software verification and validation activities shall be documented in the Software Verification and Validation Plan (SVVP).

The SVVP shall ensure that the verification activities:

SVV05 • are appropriate for the degree of criticality of the software;

SVV06 • meet the verification and acceptance testing requirements (stated in the SRD);

SVV07 • verify that the product will meet the quality, reliability, maintainability and safety requirements (stated in the SRD);

SVV08 • are sufficient to assure the quality of the product.

SVV09 By the end of the UR review, the SR phase section of the SVVP shall be produced (SVVP/SR).

SVV10 The SVVP/SR shall define how to trace user requirements to software requirements, so that each software requirement can be justified.

SVV11 The developer shall construct an acceptance test plan in the UR phase and document it in the SVVP.

SVV12 During the SR phase, the AD phase section of the SVVP shall be produced (SVVP/AD).

SVV13 The SVVP/AD shall define how to trace software requirements to components, so that each software component can be justified.

SVV14 The developer shall construct a system test plan in the SR phase and document it in the SVVP.

SVV15 During the AD phase, the DD phase section of the SVVP shall be produced (SVVP/DD).

SVV16 The SVVP/AD shall describe how the DDD and code are to be evaluated by defining the review and traceability procedures.

SVV17 The developer shall construct an integration test plan in the AD phase and document it in the SVVP.

SVV18 The developer shall construct a unit test plan in the DD phase and document it in the SVVP.

SVV19 The unit, integration, system and acceptance test designs shall be described in the SVVP.

SVV20 The unit integration, system and acceptance test cases shall be described in the SVVP.

SVV21 The unit, integration, system and acceptance test procedures shall be described in the SVVP.

SVV22 The unit, integration, system and acceptance test reports shall be described in the SVVP.

APPENDIX D
INDEX