Testing Embedded Software

Bart Broekman and
Edwin Notenboom

Testing Embedded Software

Addison-Wesley

An imprint of PEARSON EDUCATION

London · Boston · Indianapolis · New York · Mexico City · Toronto · Sydney · Tokyo · Singapore
Hong Kong · Cape Town · New Delhi · Madrid · Paris · Amsterdam · Munich · Milan · Stockholm

PEARSON EDUCATION LIMITED

Head Office
Edinburgh Gate
Harlow CM20 2JE
Tel: +44 (0)1279 623623
Fax: +44 (0)1279 431059

London Office
128 Long Acre
London WC2E 9AN
Tel: +44 (0)20 7447 2000
Fax: +44 (0)20 7447 2170

Website: www.it-minds.com
www.aw.professional.com

First Published in Great Britain in 2003

ISBN 0 321 15986 1

British Library Cataloguing in Publication Data
A CIP catalogue record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data
Applied for.

Approval has been obtained from ETAS GmbH, Stuttgart, to use the pictures that they provided.
TMap® is a registered trademark of Sogeti Nederland BV.

10 9 8 7 6 5 4 3 2 1

Typeset by Pantek Arts Ltd, Maidstone, Kent.
Printed and bound in Great Britain by Biddles Ltd, Guildford and King's Lynn.

The Publishers' policy is to use paper manufactured from sustainable forests.

Contents

Foreword

The importance of software is increasing dramatically in nearly all sectors of industry. In the area of business administration, software has become an indispensable core technology. Furthermore, more and more products and innovations are also based on software in most technical fields. A typical example is provided by the automotive industry in which a rapidly increasing number of innovations are based on electronics and software to enhance the safety of the vehicles, and also to improve the comfort of the passengers and to reduce fuel consumption and emissions. In modern upper-class and luxury cars, 20 to 25 percent of the cost is on electronics and software, and this proportion is estimated to increase to up to 40 percent in the next ten years.

Software has a substantial influence on the quality of products as well as the productivity of a company. Practice, unfortunately, shows that it is impossible to develop a complex software-based system "first-time-right". Hence, comprehensive analytical measures have to be taken to check the results of the different development phases and to detect errors as early as possible. Testing constitutes the most important analysis technique, besides reviews and inspections. It is, however, a very sophisticated and time consuming task, particularly in the field of embedded systems. When testing embedded software, not only the software has to be considered but also the close connection to the hardware components, the frequently severe timing constraints and real-time requirements, and other performance-related aspects.

This book should be regarded as an important and substantial contribution to the significant improvement of the situation in the field of testing embedded systems. It provides a comprehensive description of the world of embedded software testing. It covers important aspects such as the testing lifecycle and testing techniques, as well as infrastructure and organization. The authors' concentration on usability in industrial practice makes the book a valuable guide for any practitioner. Due to the wide range of application of embedded software, the book will be useful in many industrial businesses.

With its comprehensiveness and practical orientation this book provides a significant milestone on the long road to the effective and efficient testing of embedded software, and thus to the economic development of high-quality embedded systems. Several concepts from the book have already established a

foothold in the DaimlerChrysler Group and I hope that it will find its way to many testers and software developers in various different industrial sectors in which the development of dependable embedded systems is part of the core business.

Dr. Klaus Grimm, Berlin, June 2002
Director of Software Technology Research at DaimlerChrysler AG

Preface

A growing industry requires structured testing

The embedded systems world is a fast growing industry. It is a world which is historically dominated by engineers and technicians who excel in their own technical specialism. Historically, the technicians who built the products were also those who performed the testing because they understood best how things were supposed to work. This worked fine in orderly situations where the technicians worked on relatively small and isolated products. However, the embedded industry is changing fast – systems have become larger, more complex, and more integrated. Software now makes up the larger part of the system, often replacing hardware. Systems that used to work in isolation are linked to provide integrated functionality. This cannot be produced by one brilliant individual anymore, but has to be an organized team effort. Similarly, the process of testing has become larger, more complex, and harder to control. It has led to a growing need for a method that helps get the complex testing process under control.

Scope of this book

Embedded systems have to rely on high quality hardware as well as high quality software. Therefore, both hardware testing and software testing are essential parts of the test approach for an embedded system. However, this book concentrates more on the testing of *software* in embedded systems. Many hardware issues are included, but technical details of testing individual hardware components are not discussed – this is a profession in itself. Usually the technical people are competent in dealing with the technical intricacies involved in the testing of hardware on a detailed level. This book is mainly targeted at those who work with the software in embedded systems. It teaches them about the environment in which they work, the specific problems of testing their software, and techniques that are not normally taught in software education.

This book aims at providing answers and solutions to the growing problem of "getting the complex testing process under control." It focuses on the higher level of *how to organize the overall testing process* with its broad range of activities in both

software and hardware environments. The authors have used concepts and material from the book *Software Testing, a Guide to The TMap® Approach*, and have adapted them to fit the embedded software world.

The book is not intended to have the academic quality of a thesis. It is strongly practically oriented and aims at providing overview, insight, and lots of practical guidelines, rather than detailed academic proof.

Structure of the book

Testing is more than just exercising the system and checking if it behaves correctly. It also involves the planning of activities, designing test cases, managing the test infrastructure, setting up an organization, dealing with politics and much more. This book describes the TEmb method for structured testing of embedded software. It covers the broad range of issues that is essential in structured testing, answering the questions "what, when, how, by what and by whom?" TEmb uses the four cornerstones of structured testing as defined by the test management approach TMap® (Pol *et al.*, 2002): *lifecycle* ("what, when") of the development and testing process; *techniques* ("how"); *infrastructure* ("by what"); *organization* ("by whom"). The structure of this book follows those four cornerstones.

The book is divided into six parts:

Part I describes some general principles of structured testing and embedded systems. It provides an overview of the TEmb method, showing how to *assemble* the suitable test approach for a particular embedded system.

Part II deals with lifecycle issues and thus with the *process* of developing and testing embedded software. The lifecycle cornerstone is the core of the testing process, providing the map of what has to be done and in what order. Various issues from the other three cornerstones apply at various points in that lifecycle.

Part III offers several *techniques* that are probably useful for most embedded software test projects. It contains techniques for executing a risk-based test strategy, a testability review, formal inspections, and safety analysis. It offers various techniques for designing test cases that can be used in different circumstances for different purposes.

Part IV deals with the *infrastructure* that testers need to do their job properly. It describes the different test environments required at different stages in the testing process. An overview is provided of the various tools that can be usefully applied for different test activities and purposes. Tools that assist in achieving automated test execution have always been very popular. This part explains the technical and organizational issues involved in this kind of test automation. Finally, specific issues are discussed that arise when the tester works in an environment dealing with analog as well as digital signals (so-called "mixed signals").

Part V describes various organizational aspects of testing. It is about those who have to perform the testing activities and who have to communicate with other

personnel. It contains descriptions of the various test roles as well as the management and organizational structures. It also deals with how the testers should report on the progress of testing and on the quality of the system under test.

Part VI contains various appendices with background information on subjects such as risk classification, statechart models, a blueprint of an automated test suite and an example test plan.

Target audience

This book is targeted at those who are involved in the development and testing of embedded systems (and embedded software in particular). It offers guidelines on organizational as well as technical aspects, on a global as well as a detailed level. Different types of readers will probably have different "favourite chapters." The following may serve as a guideline as to which chapters will be most relevant for you.

It is recommended that all read the introductory chapters in Part I as well as the chapters on the multiple V-model, master test planning, and risk-based test strategy. They encapsulate the essence of the TEmb method.

Managers of development or test projects, test co-ordinators or test team leaders will benefit most from the chapters in Part II and Part V plus the chapter on "risk-based test strategy."

Testers, developers, and others who actually perform the primary software testing activities will find a lot of practical information in Part III and Part IV. If the reader is required to report formally on progress and quality, they will benefit from reading the chapter on test control in Part V.

Those who are involved in development or testing of hardware are advised to read the chapters on the multiple V-model in Part II and embedded software test environments in Part IV. They show that it is essential for both disciplines (hardware and software) to work towards a common goal and synchronize efforts to reach this goal. For the same reason, those chapters are of interest to software developers and testers.

Human resource management departments will find information that is especially suited to their line of work in the chapters on test roles and human resource management in Part V.

Acknowledgments

It will be difficult to avoid clichés such as "This book would never ....," but we just don't want to ignore the fact that we received valuable help from many people. Mentioning their contribution here is the least they deserve.

We got the chance to develop a testing method for the embedded industry when Rob Dekker and Martin Pol asked us (very persuasively) to join a European ITEA project on development methods for embedded systems in the automotive industry. Klaas Brongers managed and stimulated our efforts, providing the required freedom and creative atmosphere. The project proved to be a fertile soil for developing a test method for embedded software, where we could research new ideas and test their practical value. Especially the project partners Mirko Conrad, Heiko Dörr and Eric Sax were an invaluable source of knowledge and experience. They were very supportive as well as highly critical, which greatly increased the quality of our work. We owe a special thanks to Mirko and Eric who provided a complete chapter about the specialized subject of mixed signals.

Somewhere during the ITEA project, it became apparent that the ideas developed in the project would be useful for so many others in the embedded industry – so the idea of the book was born. We were fortunate to have managers and directors with the vision and determination required to pursue such a goal: Luciëlle de Bakker, Hugo Herman de Groot, Jan van Holten, Ronald Spaans, and Wim van Uden. They provided the essential management support.

Once the decision was made to write a book, we knew that we could count on the support of many colleagues. Bart Douven, John Knappers, Peter van Lint, Dre Robben, Coen de Vries, and Paul Willaert provided us with valuable input and feedback. Peter and Coen should be mentioned in particular. They put a lot of effort into developing the thoughts about embedded software test environments that formed the basis of the chapter about this subject. Rob Baarda was a great help in the process of turning a document into a book and getting it published. Jack van de Corput and Boy van den Dungen succeeded in raising enthusiasm at customer sites to put our ideas into practice.

As more and more chapters were completed, external reviews were intensified. We are very happy that we could take advantage of experts in the fields of testing and embedded software: Simon Burton, Rix Groenboom, Klaus Grimm,

Norbert Magnusson, Matthias Pillin, and, in particular, Stuart Reid and Otto Vinter who reviewed the complete final manuscript. We appreciate that they didn't spare us, but gave us their honest opinions and constructive criticism. Many of their remarks and suggestions have been accepted and found a place in this book.

We want to thank all those people for the invaluable contributions to our book. They can all feel as proud of this work as we do.

Bart Broekman
Edwin Notenboom

PART I

Introduction

Introduction

This part describes the general principles of structured testing of embedded systems, and provides an overview of the TEmb method.

Chapter 1 introduces some fundamentals about testing and embedded systems. It explains the purpose of testing and the main elements in a structured test process. A generic scheme of an embedded system is presented, to explain what we mean by an embedded system. This generic scheme will be used in other parts of the book, especially in Chapter 13.

Chapter 2 describes the TEmb method for structured testing of embedded software. It explains that there is no such thing as "the one-test approach" that fits all embedded systems. Instead TEmb is a method that assists in *assembling* a suitable test approach for a particular embedded system. It consists of a "basis test approach," which is furnished with several specific measures to tackle the specific problems of testing a particular system. The chapter explains how a limited set of system characteristics is useful in distinguishing different kinds of embedded systems, and the relevant specific measures that can be included in the test approach. This chapter provides a further overview of this basis test approach and a matrix that relates specific measures to system characteristics.

Fundamentals 1

1.1 Aims of testing

Testing is a process centered around the goal of finding defects in a system. It may be for debugging reasons or acceptance reasons – trying to find defects is an essential part of every test process. Although the whole world agrees that it is much better to prevent defects than to find and correct them, reality is that we are currently unable to produce defect-free systems. Testing is an essential element in system development – it helps to improve the quality of the system.

The ultimate goal of testing is to provide the organization with well-informed advice on how to proceed – advice based on observed defects related to requirements of the system (either explicitly defined or implicitly assumed). The testing in itself does not directly improve the quality of the system. But it does indirectly, by providing a clear insight to the observed weaknesses of the system and the associated risks for the organization. This enables management to make better informed decisions on allocating resources for improving the quality of the system.

To achieve these test goals, every test process contains activities for planning what is needed, specifying what should be tested, and executing those test cases. There is also a universal rule that it is impossible to find all defects and that there is never enough time (or personnel or money) to test everything. Choices have to be made about how to distribute available resources wisely. Because some things are valid for every test process, a generic test approach can be defined providing the basic structured approach for organizing a well-controlled test process.

This will be illustrated through an example involving the testing of a very simple system – a ballpoint pen.

Suppose a company wants to sell ballpoint pens. They give one to our tester with an instruction to test it. This pen can be called the *test object*. Our tester could test many different things about the pen, for instance:

- does the pen write in the right colour, with the right line thickness?
- is the logo on the pen according to company standards?
- is it safe to chew on the pen?

- does the click-mechanism still work after 100 000 clicks?
- does it still write after a car has run over it?

To be able to answer such questions, the tester should have information about what is expected from this pen. This information, preferably written, is the basis for deciding what to test and whether the result is acceptable or not. It can be called the *test basis*.

Testing if the ink or other parts of the pen are poisonous might require expensive equipment and specialists. Performing 100 000 clicks takes a lot of time. Must the tester really put all this money and effort into those tests? The tester should discuss these issues with the stakeholders, such as the director of the company and the intended users of the pens. They decide what is most important to test and in what depth. The result of this is called the *test strategy*.

When the tester performs the tests according to the test strategy, *defects* may be found, which means that the pen does not function as desired. Depending on the severity of the defects and the risk that they introduce to the intended use of the pen, the tester formulates an assessment of the quality of the pen and gives advice on how to proceed.

The tester needs more than the pen to execute the tests – there is a need, for example, for equipment to test for poisonous substances. A proper *infrastructure* should be available. The tester also needs others with specific knowledge and skills, such as an operator for the chemical equipment – a proper *organization* should be in place.

Figure 1.1 illustrates how the above elements interact with the test process. The test process defines the necessary activities and organizes them in a *lifecycle*. For complex activities, specific *techniques* are developed that aid in performing them.

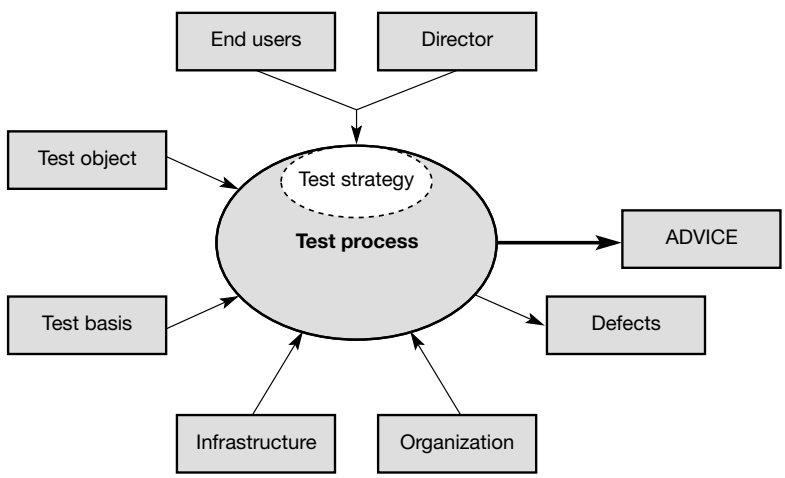


Figure 1.1 Generic elements of a test process

1.2 What is an embedded system?

"Embedded system" is one of those terms that do not really say what exactly it is about. It is a generic term for a broad range of systems covering, for example, cellular phones, railway signal systems, hearing aids, and missile tracking systems. Nevertheless, all embedded systems have a common feature in that they interact with the real physical world, controlling some specific hardware. Figure 1.2 shows a generic layout, which is applicable to virtually all embedded systems, pointing out the typical components of an embedded system.

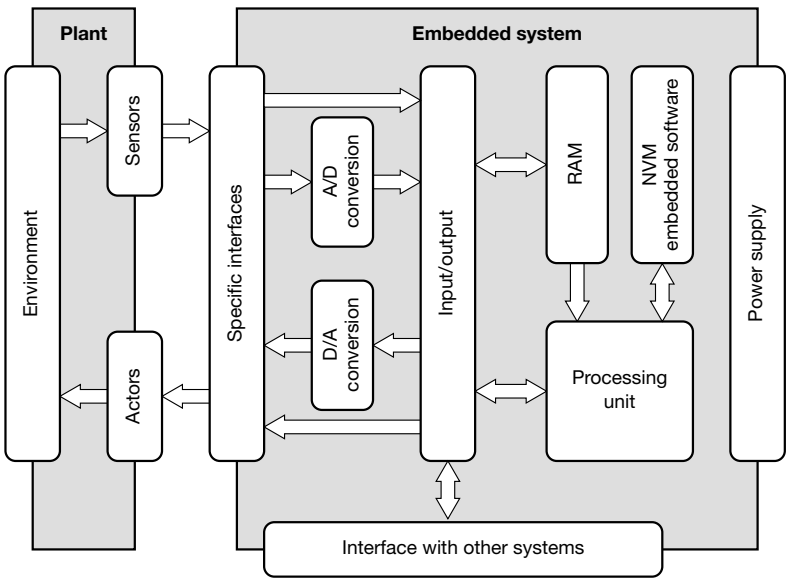


Figure 1.2
Generic scheme of an embedded system

An embedded system interacts with the real world, receiving signals through *sensors* and sending output signals to *actors* that somehow manipulate the *environment*. The environment of an embedded system, including the actors and sensors, is often referred to as the *plant*.

The *embedded software* of the system is stored in any kind of non-volatile memory (*NVM*). Often this is ROM, but the software can also be in flash cards or on hard disk or CD-ROM, and downloaded via a network or satellite. The embedded software is compiled for a particular target processor, the *processing unit*, which usually requires a certain amount of *RAM* to operate. As the processing unit can only process digital signals (ignoring analog computers for now) while the environment possibly deals with analog signals, *digital–analog conversions* (two-way) take place. The processing unit handles all input and output (I/O) of signals through a dedicated I/O layer. The embedded system interacts with the

plant and possibly other (embedded) systems through *specific interfaces*. Embedded systems may draw power from a general source or have their own dedicated *power supply*, such as from batteries.

1.3 Approach to the testing of embedded systems

Obviously, the testing of mobile phones will be significantly different from the testing of video set-top boxes or the cruise control system in cars. They each require specific measures in the test approach to cover specific issues of that system. There is therefore no point in looking for the ONE test approach for embedded systems.

Although there are many reasons why different embedded systems must be tested quite differently, there are also many similar problems, which have similar solutions, that are involved in any test approach. Some kind of basic test principles must apply to all embedded test projects – but somehow they must be differentiated with several specific measures to tackle the specific problems of testing particular systems.

This is why this book offers a *method*, which assists in *assembling* suitable test approaches. This will be explained further in the next chapter.

2 The TEmb method

TEmb is a method that helps to assemble a suitable test approach for a particular embedded system. It provides a *mechanism* for assembling a suitably dedicated test approach from the *generic elements* applicable to any test project and a set of *specific measures* relavant to the observed *system characteristics* of the embedded system. This chapter first provides an overview of the method and then explains the "generic elements," "specific measures," "mechanism," and "system characteristics" in more detail.

2.1 Overview

Figure 2.1 illustrates the TEmb method, which basically works as follows. The basis of the test approach for any embedded system consists of generic elements common to any structured test approach. For instance: planning the test project according a certain lifecycle, applying standardized techniques, dedicated test environments, organizing test teams, formal reporting, etc. They are related to the four cornerstones of structured testing: lifecycle, infrastructure, techniques, organization (referred to as "LITO" in Figure 2.1). This basis test approach is not yet concrete enough and needs to be enhanced with details such as: which design techniques will be applied? which tools and other infrastructure components will be used and must, perhaps, be developed first? etc. In the initial stages of the project, choices must be made of those specific measures that will be included in the test approach. In the TEmb method this is called "the mechanism for assembling the dedicated test approach."

The mechanism is based on the analysis of:

- *Risks*. Measures are chosen to cover the business risks involved in poor quality of the product. This is explained in detail in Chapter 7.
- *System characteristics*. Measures are chosen to deal with the problems related to the (technical) characteristics of the product. They are high-level characteristics such as technical-scientific algorithms, mixed signals, safety critical, etc.

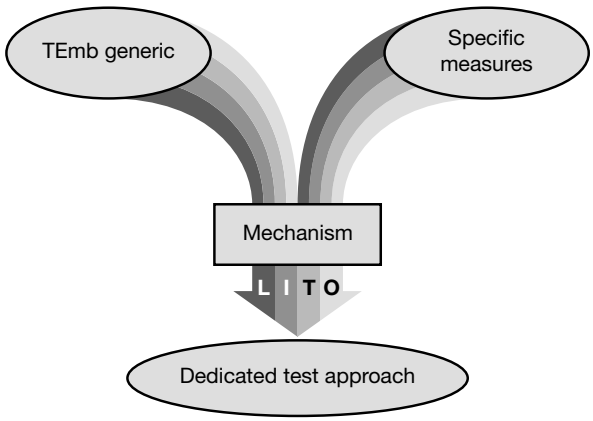


Figure 2.1
Overview of the TEmb method: the dedicated test approach is assembled from generic elements and specific measures, both related to the four cornerstones (LITO) of structured testing

The part related to the analysis of system characteristics will be further explained through this following example. Consider the following ten embedded systems:

- set-top box
- cruise control
- weather forecast
- wafer stepper
- pacemaker
- NMR scan
- infrared (IR) thermometer
- railroad signaling
- telecom switches
- goal keeper (missile defense system).

These are very different systems and it is natural to assume that the test approaches for them will be very different as well. But they don't have to be completely different. What is it that makes each system so special to test? Is there a commonality in what makes them unique? For the sake of this example, look at the following two characteristics that may or may not apply to each system:

- *Safety critical*. Failure of the system can result in physical harm to a human being.
- *Technical-scientific algorithms*. The processing consists of complex calculations, such as solving differential equations or calculating the trajectory of a missile.

Now assess each of the ten systems as to whether these two characteristics apply or not. This divides the systems into four groups, as shown in Table 2.1.

Table 2.1
"What makes the system special" classified using system characteristics

Technical-scientific algorithms	Safety critical: No	Safety critical: Yes
No	set-top box wafer steppers telecom switches	railroad signaling pacemaker
Yes	weather forecast IR thermometer	cruise control NMR scanner goal keeper

The systems that belong to the same group possess similar qualities that can be tackled in a similar way. For instance, the test approaches for both railroad signaling and the pacemaker will contain specific measures to address the safety critical nature of the system but will not bother to tackle technical-scientific algorithms. Because a link is provided between the system characteristics and suitable specific measures, assembling a suitable test approach becomes almost an easy task. For the two characteristics of this example the following set of specific measures can be proposed.

- For safety critical systems, the lifecycle MOD-00-56 (Table 2.2) can be applied and a dedicated test level defined to execute safety tests. The new roles of safety manager and safety engineer appear in the test process. Several techniques are proposed – Failure Mode and Effect Analysis (FMEA), Fault Tree Analysis (FTA), model checking, and formal proofing.
- In order to tackle the technical-scientific algorithms several techniques can be helpful in assuring that the complex processing flow is sufficiently covered – evolutionary algorithms and threat detection. This also requires tools for coverage analysis and threat detection. An activity is defined early in the test (or development) process to explicitly validate the algorithm used. In this specialized field, testing needs the support of mathematical expertise.

This is illustrated in Table 2.2, which also shows to which cornerstone (lifecycle, infrastructure, techniques, organization) the specific solution is related.

Table 2.2
For each system characteristic there is a set of proposed specific measures (related to the four cornerstones)

System characteristic	Lifecycle	Infrastructure	Techniques	Organization
Safety critical	Master test plan, incl. MOD-00-56 Safety test (test level) Load/stress test	Coverage analyzers	FMEA/FTA Model checking Formal proof Rare-event testing	Safety manager Safety engineer
Technical-scientific algorithms	Algorithm validation	Coverage analyzers Threat detectors	Evolutionary algorithms Threat detection	Math expertise

The advantage of focusing on system characteristics is that just a limited set is required to cover an enormous diversity of systems. The uniqueness of each embedded system test project is made more concrete and manageable by analyzing which system characteristics are relevant. Linking possible useful measures to each of the system characteristics then provides a useful guide to what is needed in your dedicated test approach.

The mechanism explained here can be seen as an extension to the risk-based test strategy – the analysis of perceived risks (or usually "the agreement on" because it is rather subjective) leads to choices of what to do (and what not to do) in the test approach. The analysis of system characteristics (which is more objective in nature) leads to proposed solutions which can be adapted in the test approach to tackle the particular issues of your specific embedded system's test project.

2.2 TEmb generic

This section describes the generic elements that constitute the basis of any structured testing process.

During the development of a new system, many different kinds of tests (see section 4.1.2) will be performed by many different people or teams. At the start of the new project, a master test plan is produced which determines who will be responsible for which test and what relationships there are between the different tests (see section 4.1.3).

For each separate test level, the same four cornerstones of structured testing apply (see Figure 2.2). They are the answers to the major questions "what and when," "how," "by what," and "by whom:"

- *Lifecycle*. This defines which activities have to be performed and in what order. It gives testers and managers the desired control over the process.
- *Techniques*. This helps with how to do things, by defining standardized ways to perform certain activities.

- *Infrastructure*. This defines what is needed in the test environment to make it possible to perform the planned activities.
- *Organization*. This defines the roles and required expertise of those who must perform the planned activities and the way they interact with the other disciplines.

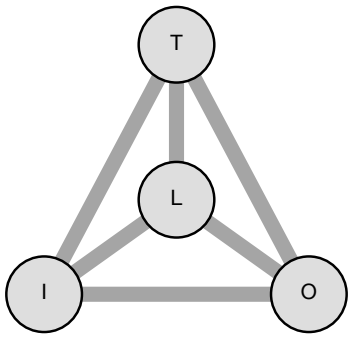


Figure 2.2
The four cornerstones of a structured test process

The basic idea is that all four cornerstones must be covered equally well in the test process. If one is neglected, that is exactly where the test process will run into trouble. The lifecycle can be considered to be the central cornerstone. It is the "glue" between the cornerstones. At different stages in the lifecycle the other three cornerstones will be applied differently. For instance, specification activities require different techniques and expertise to test execution and analysis activities.

When appropriate attention is given to all four cornerstones, the test process can be considered to be structured. It is a major factor in eliminating forgetting vital things and makes the testing more manageable and controllable. It will not guarantee that the testing will never run into trouble. Every project will now and then encounter unforeseen events that throw carefully planned activities into disarray. However, a structured test process has the best chance to recover from such events quickly with a minimum of damage.

The following sections briefly discuss each cornerstone separately.

2.2.1 Lifecycle

In the lifecycle model, the principal test activities are divided into five phases (see Figure 2.3). In addition to the phases of planning & control, preparation, specification, and execution, a completion phase has been defined to round off the test process satisfactorily and to formally deliver the testware to the line organization for subsequent maintenance releases.

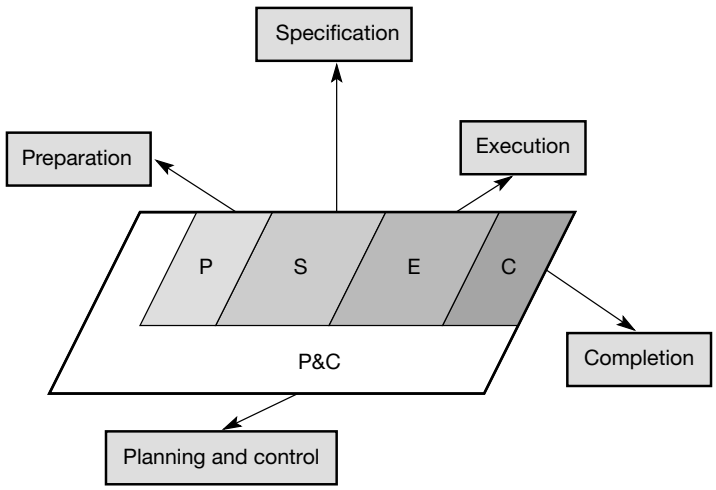


Figure 2.3
The five phases in the lifecycle model

The basic idea behind the lifecycle is to try to do as many activities as possible as soon as possible, away from the critical path of the project. When the system to be tested is delivered to the tester in the execution phase, it would be a waste of precious time if the tester only then starts to think what exactly the test cases should look like. This could, and should, have been done in the previous phase, specification. It would be equally annoying if the tester could not start specifying test cases during the specification phase because it still has to be figured out which design techniques to choose and how they can be applied to the system specifications just received. This should have been done in the previous phase, preparation.

Although the lifecycle may explicitly show more activities than the test organization primarily thought of, it does not necessarily add extra activities. It merely makes a clearer distinction between the things that must be done and arranges them more efficiently and effectively. When applied wisely, the lifecycle is a powerful mechanism for saving time when it matters most, on the critical path.

2.2.2 Techniques

This cornerstone supports the test process by offering testers elaborate, proven, and universal working methods, as well as enabling the management (and auditors) to track the progress of the test process and evaluate the results. In principle, one or more techniques can be devised for every type of activity. New techniques are developed somewhere in the world almost every day. Whenever an activity exists which is likely to be repeated several times, the future test process would be supported by devising a specific technique for that activity.

The TEmb model provides many techniques. Some are 'good old' techniques that have proven their value and need no modifications, some are new, and others are based on existing techniques. Many of the techniques are a mixture of everything.

Here are some examples of techniques that are provided by TEmb.

- *Strategy development*. Test strategy is all about making choices and compromises, based on risk assessment. The technique for strategy development results in an agreement between all stakeholders about how much should be invested in a test in order to find an optimal balance between the required quality and the amount of time, money, and resources needed for it.
- *Test design*. Many techniques have been developed to design test cases. The choice of which design techniques to use depends, among other things, on the quality characteristic to be evaluated, the coverage to be achieved, the available system specification, etc.
- *Safety analysis*. Specific techniques have been developed to evaluate the safety aspects of the system and agree on measures that should be taken.
- *Data driven test automation*. Many powerful tools exist for automating tests. They can not only "capture & replay," but also have support for programming advanced test scripts. Such tools are, in fact, a development environment in which to build automated test suites. The technique of data driven testing can often be successfully applied to achieve maintainable and flexible automated tests.
- *Checklists*. The use of checklists is a simple but effective technique. The tester's experiences can be captured in a checklist and reused in future test projects.

2.2.3 Infrastructure

The infrastructure for testing includes all the facilities required for structured testing. It can be divided into facilities needed for executing the test (test environment), facilities that support efficient test execution (tools and test automation), and facilities for housing the staff (office environment).

Test environment

The three most important elements of the test environment are:

- *Hardware/software/network*. The system to be tested can have different physical appearances in the different development stages. For instance a model, a prototype, isolated units connected to simulators, production type. For each of these types and stages, different test environments are required.
- *Test databases*. Most tests are designed to be repeatable, which helps to achieve reproducible test results. This means that certain test data has to be stored in one way or another.

- *Simulation and measurement equipment*. When the system (or component) to be tested cannot yet run in the real world but needs external signals, simulators may be required to provide these. Also, the system may produce output that requires special equipment for detection and analysis.

Tools and test automation

Tools are not an absolute requirement for executing the test process, but they can make life a lot easier for testers. The scope of test tools is huge and their diversity has increased enormously. Test tools can be classified according to the activities they support, and thus to the phase of the test process in which they are primarily used. The following list shows some examples of applicable types of test tools for each phase of the lifecycle.

- *Planning and control*
  - planning and progress control
  - defect administration
  - configuration management.
- *Preparation*
  - requirements management
  - complexity analysis.
- *Specification*
  - test case generator
  - test data generator.
- *Execution*
  - capture and playback
  - comparator
  - monitor
  - coverage analyzer.

Office environment

It is obvious – the quality of the test depends largely on the skills and motivation of those who perform it. Unnecessary distractions and irritation should be avoided. An important element here is that testers should have a suitable environment where they can organize and perform their own work, from planning to completion. It seems simple enough, but in reality testers find all too often that their desks and workstations are not ready on time or that they are obliged to share them with colleagues.

2.2.4 Organization

Organization is all about people and how they communicate. Testing is not a straightforward task that can be performed in isolation, unhindered by interferences from a demanding outside world. The involvement of many different disciplines, conflicting interests, unpredictability, shortage of expertise, and time constraints make the set-up and management of the test organization a difficult task.

The organization cornerstone covers the following subjects:

- *Structure of the test organization*. This deals with the position of the test organization within the overall scheme as well as the internal structure of the test organization. It covers the hierarchy of the different disciplines, their responsibilities, and their relationships.
- *Roles*. This defines for each discipline the tasks that must be performed and the skills and expertise required to perform them well.
- *Staff and training*. This covers the topic of how to get the personnel you need and how to keep them – it is about acquisition of staff, training, and career paths.
- *Management and control procedures*. Because testing usually takes place in a world that changes constantly, the test process must have procedures that enable it to handle change. Three different kinds of control procedures are distinguished:
  - test process
  - test infrastructure
  - test products.

2.3 Mechanism for assembling the dedicated test approach

The test projects for different embedded systems will use the same basic principles (see previous paragraph) but will differ at a more detailed level. Each project incorporates many specific concrete measures to achieve its particular goals and to tackle the specific problems of testing its particular embedded system. In the TEmb method this is called "the mechanism for assembling the dedicated test approach". It is based on the analysis of *risks* and *system characteristics*.

The choice of test measures based on risk analysis is a process of negotiating what to and what not to do, and assigning test priorities. It is covered in detail in Chapter 7 and will not be discussed further here.

The analysis of system characteristics is more objective in nature, dealing more with facts than opinions. This will be explained in more detail here. First, the system characteristics are explained along with the kind of impact on the testing process for such a system. Then an overview is presented, the so-called LITO-matrix, in which specific solutions are applicable for each system characteristic.

2.3.1 System characteristics

Chapter 2.1 explained the principle of using system characteristics to state what it is that makes the system special. It is important to note that the TEmb method does not aim at achieving a scientifically accurate and complete taxonomy of embedded systems. Rather its purpose is entirely practical and purely from a tester's perspective. It aims at assisting the test manager in answering the question "What makes this system special and what must be included in the test approach to tackle this?"

The following provides a useful initial set of system characteristics:

- safety critical systems
- technical-scientific algorithms
- autonomous systems
- unique system; "one-shot" development
- analog input and output (in general, mixed signals)
- hardware restrictions
- state-based behavior
- hard real-time behavior
- control systems
- extreme environmental conditions.

The list is not meant to be exhaustive. Organizations are encouraged to define other system characteristics that are applicable to their products.

Note that no system characteristic is defined for *reactive systems*. A system is described as reactive if it is fast enough to respond to every input event (Erpenbach *et al*., 1999). It is fully responsible for synchronization with its environment. This can be sufficiently covered by a combination of the two characteristics state-based behavior and hard real-time behavior.

The system characteristics above will be briefly described here.

Safety critical systems

An embedded system is said to be safety critical if a failure can cause serious damage to health (or worse). Administrative systems are seldom safety critical. Failure in such systems usually leads to annoyance and financial damage, but people rarely get hurt. However, many embedded systems have a close physical interaction with people and failure can cause direct physical harm. Examples of such systems are in avionics, medical equipment, and nuclear reactors. With such systems, risk analysis is extremely important and rigorous techniques are applied to analyze and improve reliability.

Technical-scientific algorithms

Often the behavior of an embedded system seen from the outside is relatively simple and concise. For instance, what cruise-control systems offer to drivers can be described in a few pages. However, realizing such a system may be very difficult and require a lot of control software to process complex scientific calculations. For such systems, the larger and more complex part of its behavior is internal and invisible from the outside. This means that the test effort will be focused on white-box oriented test levels, while relatively less is required for black-box oriented acceptance testing.

Autonomous systems

Some embedded systems are meant to operate autonomously for an indefinite period of time. They are "sent on a mission." After starting up, they require no human intervention or interaction. Examples are traffic signaling systems and some weapon systems that, after being activated, perform their task automatically. The software in such systems is designed to work continuously and react to certain events without human intervention being needed, or even possible. A direct result of this is that such systems cannot be tested manually. A specific test environment with specific test tools is required to execute the test cases, and measure and analyze the results.

Unique system; "one-shot" development

Some systems, such as satellites, are released (or launched) once only and cannot be maintained. They are unique systems (also called "bespoke systems") and meant to be built correctly "in one shot." This is in contrast to mass market products that are released in a competitive market and must be upgraded regularly and have releases to remain attractive. For mass market products, maintenance is an important issue and during development and testing special measures are taken to minimize maintenance costs and time. For instance, reuse of testware and automation of regression testing are probably standard procedures. However, for unique systems the testing has, in principle, no long-term goals because it stops after the first and only release. For this kind of system some rethinking must be done about maintenance, reuse, regression testing, etc.

Analog input and output (in general, mixed signals)

In the administrative world, the result of a transaction can be predicted exactly and measured. An amount on an invoice, a name, or an address is always defined exactly. This is not the case in embedded systems that deal with analog signals. The input and output do not have exact values but have a certain tolerance that defines the acceptable boundaries. Also, the expected output is not always defined exactly. An output value that lies just outside a defined boundary is not always definitely invalid. A grey area exists where sometimes, intuitively, it is decided if the test output is acceptable or not. In such situations terms such as "hard boundaries" and "soft boundaries" are used.

Hardware restrictions

The limitations of hardware resources can put specific restrictions on embedded software, for instance on memory usage and power consumption. It also happens that specific hardware dependent timing aspects are solved in the software. These kinds of restrictions on the software have little to do with the required functionality but are essential if a system is to function at all. They require a significant amount of testing effort of a specialized and technical nature.

State-based behavior

The behavior of a state machine can be described in terms of transitions from a certain state to another state, triggered by certain events. The response of such a system to a certain input depends on the history of previous events (which resulted in a particular state). A system is said to exhibit state-based behavior when identical inputs are not always accepted and, if accepted, may produce different outputs (Binder, 2000).

Hard real-time behavior

The essence of real time is that the exact moment that input or output occurs, influences the system behavior. A salary system for instance is not real time – it doesn't make any difference if the system calculates your salary now or after 15 minutes. (Of course if the system waits a whole year, it does matter, hopefully, but that doesn't make it real time.) In order to test this real-time behavior, the test cases must contain detailed information about the timing of inputs and outputs. Also, the result of test cases will usually be dependent on the sequence in which they are executed. This has a significant impact on both test design and test execution.

Control systems

A control system interacts with its environment through a continuous feedback mechanism – the system output influences the environment, which in turn influences the behavior of the control system. Therefore the behavior of the system cannot be described independently, it is dependent on the behavior of the environment. Testing of such systems usually requires an accurate simulation of the environment behavior. Examples of such systems are industrial process control systems and aircraft flight control systems.

Extreme environmental conditions

Some systems are required to continue operating under extreme conditions, such as exposure to extreme heat or cold, mechanical shock, chemicals, or radiation. Testing this usually requires specialized equipment to generate the extreme conditions. Similarly, if testing in the real environment is too dangerous, simulation equipment is used instead.

2.3.2 Specific measures

For a particular embedded system, the basis test approach provided by "TEmb generic" must be furnished with several specific measures to tackle the specific problems of testing that particular system. Typically, each one of such measures may be very helpful in testing one embedded system but be completely irrelevant for testing another embedded system (hence the term "specific" instead of "generic"). It is, of course, impossible to establish a complete overview of all specific measures. The following list merely aims at illustrating specific measures with some typical examples.

- Specific test design techniques can be applied to test the state-based behavior of the system.
- In some development environments, the system can first be modeled and then this model can be dynamically tested. This enables, for instance, testing the real-time behavior of a system before any code has been written.
- Dedicated tools exist for performing so-called "threat detection." They do not find hard defects but they search for conditions in the input domain which *may* cause the software to fail.
- The use of evolutionary algorithms (or "genetic algorithms") is a special test design technique where the test cases are not derived from the system documentation but from the way the test cases themselves behave. It is an optimization process that causes test cases to evolve into better test cases. It is based on the biological principle of "survival of the fittest."
- When testing requires several expensive simulators, a specific controlling department can be installed to maintain and manage them. This is especially useful when many different teams must share these tools at various overlapping stages.
- The British standard MOD-00-56 defines a specific lifecycle for safety analysis activities (see section 10.3.1).

2.3.3 LITO-matrix

System characteristics have an impact on the test process by raising issues that need to be solved by the test approach. Specific measures can help to solve certain issues related to one or more system characteristics. The measures can be attributed to one of the four cornerstones: lifecycle (L), infrastructure (I), techniques (T), and organization (O). The relationship between system characteristics and specific measures (within each cornerstone) can be represented in a matrix, the so-called LITO-matrix. This provides an overview of the system characteristics and the applicable measures that can be included in the test approach.

Table 2.3 is an example of such a matrix. It should be used only as a guideline or starting point. It is not meant to be complete, nor should it be considered the only true overview. Some people may identify other more important system characteristics. Others may argue that certain proposed measures are not very helpful and identify others that are more effective. That is fine, as long as it prompts to think deeper about the impact on testing of "what kind of system this is" and how this can be tackled in the test approach.

Table 2.3
The LITO-matrix. Each system characteristic is related to applicable measures that can be included in the test approach

System characteristic	Lifecycle	Infrastructure	Techniques	Organization
Safety critical	Master test plan, incl. MOD-00-56 Safety test (test level) Load/stress test	Coverage analyzers	FMEA / FTA Model checking Formal proof Rare event testing	Safety manager Safety engineer
Technical scientific algorithms	Algorithm validation	Coverage analyzers Threat detectors	Evolutionary algorithms Threat detection	Mathematical expertise
Autonomous system	Hardware in the loop Software in the loop	Measurement probes	Rare event testing	
One-shot system		Coverage analyzers	Rare event testing Coverage analysis	
Mixed signals		Mixed signal analyzer Signal generator	Analyze mixed signal output Classification-tree method (CTM/ES enhanced)	Controlling department for dedicated tools
Hardware restrictions – power – memory		Host/target testing environment	Algorithm efficiency analysis	
State-based behavior		State modeling and testing tools	State transition testing Statistical usage testing Rare event testing	
Hard real-time		Logic analyzer Time measurement probes	Evolutionary algorithms	
Control system	Hardware in the loop Software in the loop	Simulation of feedback-loop	Statistical usage testing Rare event testing Feedback control test	Control engineering
Extreme environmental conditions	Field test	Environmental simulators	Electromagnetic interference test[1]	

[1] EMI tests may be obligatory due to legal regulations.

Lifecycle PART II

Introduction

This part deals with the *process* of developing and testing embedded software. A lifecycle structures that process by dividing it into phases, describing which activities need to be performed, and in what order.

In Chapter 3, the multiple V-model is introduced to describe the development lifecycle of an embedded system. It is based on the idea that throughout the development process different physical shapes of the system can be produced: first a model that simulates the system's behavior; then various prototypes that evolve in a series of iterations into the final "real" shape. This chapter gives an overview of which test issues are relevant at which moment in the multiple-V development lifecycle.

Organizing testing in the complex environment of multiple-V development is itself a complex task. Many different test activities are performed by many different specialists at many different moments in the lifecycle. In such a situation, a master test plan (see Chapter 4) proves immensely valuable. It provides the overall picture of all relevant test activities and issues, and how they are related.

The tests that take place during the earlier stages in the development lifecycle (on the left side of the multiple V-model) are low-level tests, such as unit tests and integration tests. They are often performed by developers, or in close collaboration with developers. These tests are usually organized as part of the development plan and do not have a separate test plan or budget. Chapter 5 provides guidelines on how to support and structure these low-level tests.

The tests that occur towards the end of the development lifecycle (on the right side of the multiple V-model) are the high-level tests, such as system tests and acceptance tests. They are often performed by an independent test team, which is responsible for a separate test plan and budget. Chapter 6 describes a dedicated detailed lifecycle for these test levels.

Multiple V-model

3

3.1 Introduction

In embedded systems, the test object is not just executable code. The system is usually developed in a sequence of product-appearances that become more real. First a *model* of the system is built on a PC, which simulates the required system behavior. When the *model* is found to be correct, code is generated from the model and embedded in a *prototype*. The experimental hardware of the prototypes is gradually replaced by the real hardware, until the system is built in its final form as it will be used and mass produced. The reason for building those intermediate product-appearances is, of course, that it is cheaper and quicker to change a prototype than to change the final product, and even cheaper and quicker to change the model.

3.1.1 Straightforward multiple V-model

The multiple V-model, based on the well-known V-model (Spillner, 2000) is a development model that takes this phenomenon into account. In principle each of the product appearances (model, prototypes, final product) follows a complete V-development cycle, including design, build and test activities. Hence the term "multiple V-model" (see Figure 3.1). The essence of the multiple V-model is that different physical versions of the same system are developed, each possessing the same required functionality in principle. This means, for instance, that the complete functionality can be tested for the model as well as for the prototype and the final product. On the other hand, certain detailed technical properties cannot be tested very well on the model and must be tested on the prototype – for instance, the impact of environmental conditions can best be tested on the final product. Testing the different physical versions often requires specific techniques and a specific test environment. Therefore a clear relation exists between the multiple V-model and the various test environments (see Chapter 13).

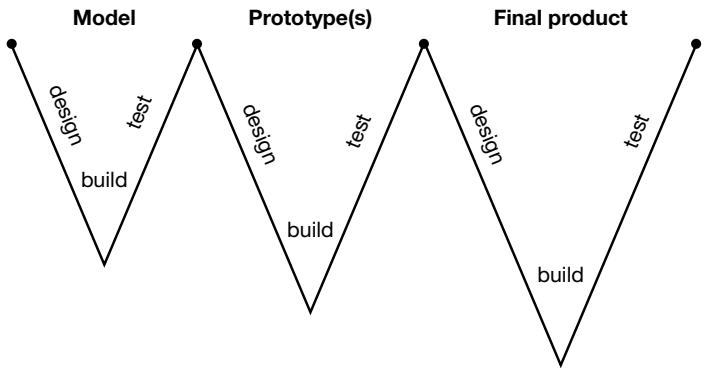


Figure 3.1 Multiple V development lifecycle

3.1.2 Iterative and parallel development

The multiple V-model shows three consecutive V-shaped development cycles (model, prototypes, and final product). It is important to understand that this is a simplified representation of the development process for embedded systems. It should not be regarded as a straightforward sequential process ("waterfall model") where the prototype is first fully designed, then built, tested and made ready as a final product. The middle V especially, where the prototypes are developed, is iterative in nature. Iterative development models that may apply here are, for instance, the Rational Unified Process (Kruchten, 2000) and eXtreme Programming (XP) (Beck, 2000). In reality, developing an embedded system is a complex process for the following reasons.

- It is a multidisciplinary project involving both software and hardware development teams. These often work independently and in parallel. This poses the risk of finding out too late that the hardware and software parts don't work together. A good project manager will ensure that there is frequent communication, integration, and testing. This usually results in an iterative process. First a small part of the system functionality is built (both in hardware and software). After successful integration and testing, the next bit of functionality is built, and so on. These integration activities serve as important milestones in the project planning, showing consolidated progress. During this process, more and more functionality is realized and more "experimental" hardware is replaced by real hardware.
- The development of large and complex systems requires (functional) decomposition of the system, leading to *parallel development* of components and *stepwise integration*. (The effect of this on the multiple V-model is explained further in section 3.3.) The multiple V-model applies to each of the components. For each component a model can be developed, followed by iterative development of both hardware and software. Integration of the different components can take place at several points in the development process: very early, on experimental hardware in the prototype stage; or much later as final products when the components are already fully developed.

This explains why the development of a particular embedded system will be a (unique) complex process with many development and test activities happening at various times, requiring much communication and co-ordination. It is important that everyone involved, developers and testers alike, have a clear map of this complex process. The multiple V-model, however simplified it may be, provides a solid basis for achieving this. The following sections provide an overview of test-related issues and show at which point in the multiple V-model they are most relevant.

3.2 Test activities in the multiple Vs

The test process involves a huge number of test activities. There are many test design techniques that can and will be applied, test levels and test types (see section 4.1) that must be executed, and test related issues that require attention. The multiple V-model assists in structuring these activities and issues. By mapping them onto the multiple Vs, it provides insight to the questions: "When can activities best be executed?" "Which test issues are most relevant at which stage in the development process?"

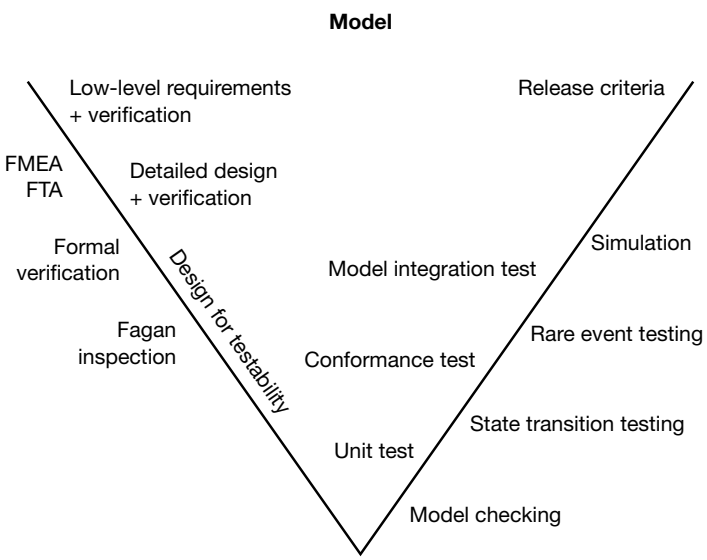
Organizing the testing in the complex situation of multiple V-model development is itself a complex task. In order to plan and manage this well, the test manager needs an overall picture of all relevant test activities and issues, and how they are related (see Chapter 4). Of course the details of this overall picture are unique to each project, but the general principles of structuring the test process always apply.

This section serves as a guideline to mapping the various test activities and issues onto the multiple Vs. It takes a broad range of activities and issues relevant to the testing process. Table 3.1 shows the activities and issues that are to be considered. They are in alphabetical order and divided into three categories: test techniques; test levels and types; and other issues. Each is then put on one or more Vs at the relevant stage of that V. It is possible for certain issues, such as low-level requirements, to appear in the V for the model as well as in the V for the prototype. Figures 3.2 to 3.4 show at which time in which V they are most applicable.

Table 3.1
Test related activities and issues that need to be allocated throughout the development and testing lifecycle

Techniques	Test levels, types	Other issues
Code coverage analysis	Architectural design verification	Architectural design
Control flow testing	Code review	Certification
Fagan inspection	Conformance test	Detailed design
Failure mode and effect analysis (FMEA)	Detailed design verification	Detailed test plan
Fault injection (FTA)	Hardware/software integration test	Design and build tools
Fault tree analysis	Host/target test	Design and build simulator
Formal verification	Model integration test	Design and build stubs
Interface testing	Real-life test, or field test	Design and build drivers
Model checking	Regression test	Design for testability
Mutation testing	Requirements verification	High-level requirements
Random testing	Software acceptance test	Legal requirements
Rare event testing	Software integration test	Low-level requirements
Simulation	System acceptance test	Master test plan
State transition testing	System integration test	Production requirements
Statistical usage testing	Unit test	Release criteria/advise
		Safety plan

Figure 3.2
Allocation of test-related issues on the development cycle of the model



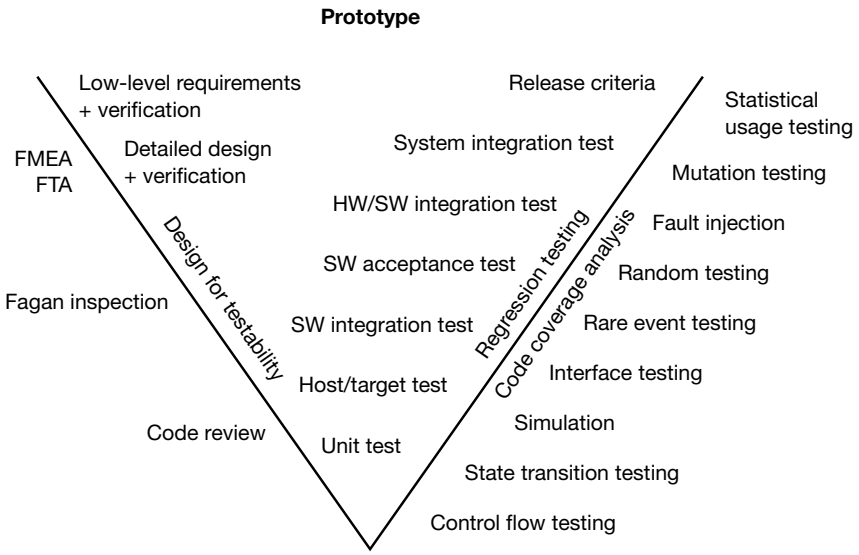


Figure 3.3
Allocation of test-related issues on the development cycle of the prototype

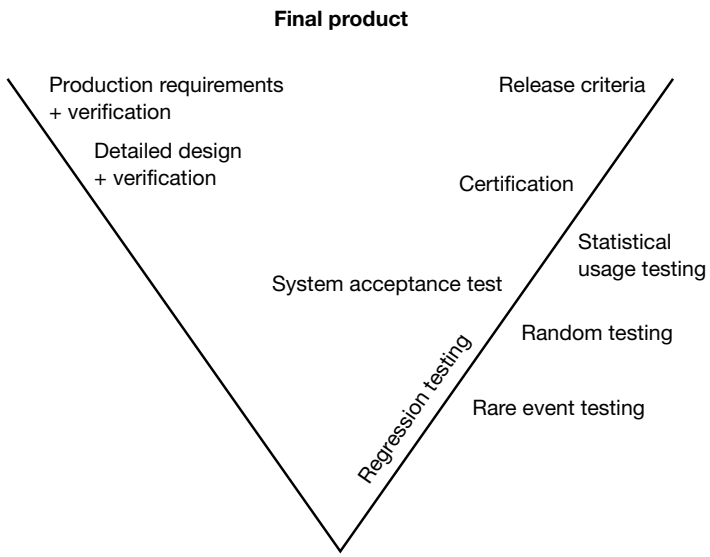


Figure 3.4
Allocation of test related issues on the development cycle of the final product

3.3 The nested multiple V-model

The multiple V-model with the three sequential V-cycles does not take into account the practice of (functional) decomposition of a complex system. The development of such a system starts with the specification of the requirements at a high-level, followed by an architectural design phase where it is determined which components (hardware and software) are required to realize this. Those

components are then developed separately and finally integrated into a full system. In fact the simple V-model can be applied to this development process at a high-level. The left side of the V-model handles the decomposition of the system into its components. The middle part of the V-model consists of parallel development cycles for all components. The right side of the V-model handles the integration of the components. This is illustrated in Figure 3.5.

This principle can be repeated for components that are too big or too complex to develop as one entity. For such a component, an architectural design activity is carried out to determine which subcomponents are required. Because this may result in a V-model within a V-model (within a V-model, …) the development lifecycle model is said to be "nested."

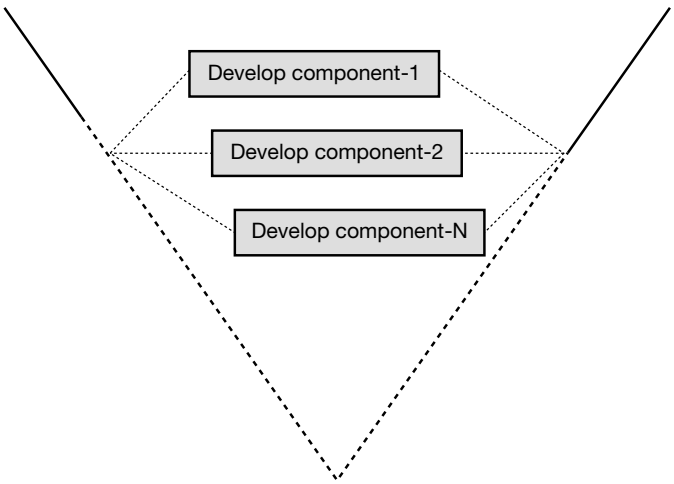


Figure 3.5 Parallel development phases in a V-model

In fact the purely sequential multiple V-model is mainly applicable to the component level. Usually it is not the complete system which is fully modeled first, then completely prototyped, and so on. It is the components that are built in this stepwise way. This explains why some development activities and issues cannot be mapped very well on the 3 Vs in the multiple V-model – for instance high-level and low-level requirements, safety plan, design and build specific tools. That is because they belong to the overall development process.

When the V-model at the system level is combined with the multiple V-model at the component level, it results in the so-called "nested multiple V-model" (Figure 3.6).

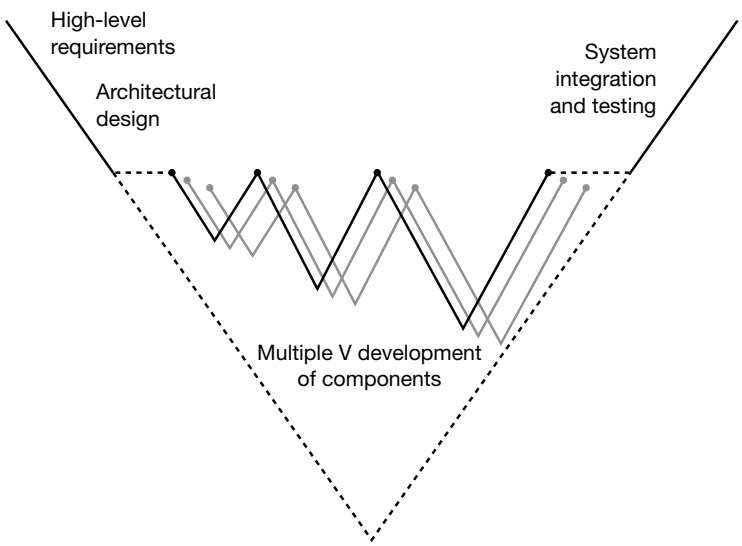


Figure 3.6
The nested multiple V-model

With this model, all test-related activities and issues can be allocated to the correct level in the correct place. At the system level, higher-level test issues can be allocated to the overall development cycle, as shown in Figure 3.7.

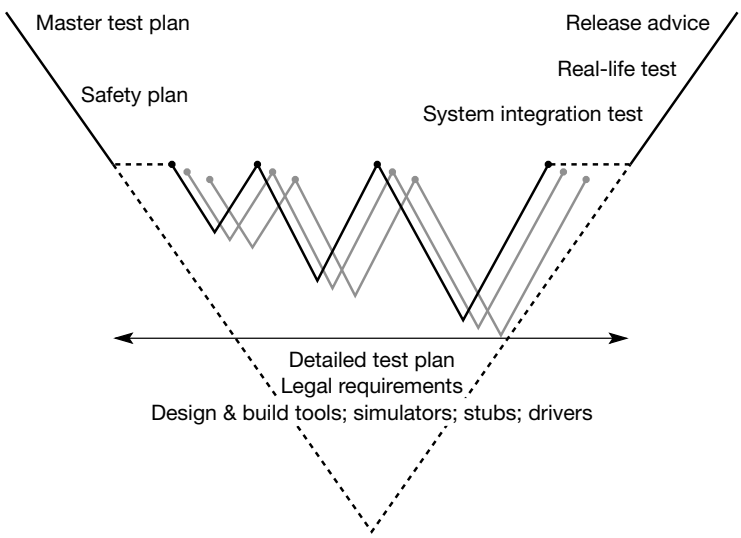


Figure 3.7
Higher-level test issues in the nested multiple V-model

The multiple V-model is not only helpful when a test has to be planned and executed for a completely new (developed) product, but also in the case of new releases of an existing product. In practice, the development project for a product is about the new release of just a few components. Then the multiple V-model and the full (master) test plan related to the development of the complete product can help to identify the relevant test activities. First it should be identified where the development activities fit in the multiple V-model. Then the full master test plan helps in choosing the integration and test activities for an effective master test plan dedicated to this new release.

Master test planning

4

4.1 Elements of master test planning

The testing of a large system, with many hardware components and hundreds of thousands lines of code in software, is a complex endeavour involving many specialists performing different testing tasks at various points in the project. Some test the states and transitions of certain components exhaustively, others test transitions on a higher integrated level, and others evaluate user-friendliness features. Some teams are dedicated to testing performance, others are specialized in simulating the real world environment. For those complex situations, master test planning provides a mechanism to control the overall testing process.

First the difference between test types and test levels is explained. It is the difference between which *aspects* are being tested and which *organizational entity* is executing the test. The master test planning then deals with the combination of both.

4.1.1 Test types

A system can be tested from different points of view – functionality, performance, user-friendliness, etc. These are the quality attributes that describe the various aspects of system behavior. Standards exist, for instance ISO 9126, to define a set of quality attributes that can be used for this purpose. In practice, several quality attributes are often combined in a single test. This gives rise to the concept of *test type*.

> A test type is a group of activities with the aim of evaluating a system on a set of related quality attributes.

Test types state *what* is going to be tested (and what is not). For instance, a tester doing a functionality test is not (yet) interested in the performance displayed, and vice versa. Table 4.1 describes some common test types – it is not intended to be complete. An organization should feel free to invent a new test type when the need arises.

Table 4.1
Test types

Test type	Description	Quality characteristics included
Functionality	Testing functional behavior (includes dealing with input errors)	Functionality
Interfaces	Testing interaction with other systems	Connectivity
Load and stress	Allowing large quantities and numbers to be processed	Continuity, performance
Support (manual)	Providing the expected support in the system's intended environment (such as matching with the user manual procedures)	Suitability
Production	Test production procedures	Operability, continuity
Recovery	Testing recovery and restart facilities	Recoverability
Regression	Testing whether all components function correctly after the system has been changed	All
Security	Testing security	Security
Standards	Testing compliance to standards	Security, user-friendliness
Resources	Measuring the required amount of resources (memory, data communication, power, …)	Efficiency

4.1.2 Test levels

The test activities are performed by various testers and test teams at various times in the project, in various environments. Those organizational aspects are the reason for introducing *test levels*.

> A test level is a group of activities that is organized and managed as an entity.

Test levels state *who* is going to perform the testing and *when*. The different test levels are related to the development lifecycle of the system. It structures the testing process by applying the principle of *incremental testing* – early in the development process, parts of the system are tested in isolation to check that they conform to their technical specifications. When the various components are of satisfactory quality, they are integrated into larger components or subsystems. These in turn are then tested to check if they conform to higher-level requirements.

Often a distinction is made between low-level and high-level tests. Low-level tests are tests on isolated components, executed early in the lifecycle (left side of the multiple V-model) in a development-like environment. High-level tests are tests on the integrated system or subsystems, executed later in the lifecycle

(right side of the multiple V-model) in a (simulated) real-life environment. Chapter 13 describes in more detail the different test environments for the different test levels. Usually low-level tests are more white-box oriented and high-level tests more black-box oriented.

Table 4.2 lists some common test levels for embedded systems – it is not intended to be complete. Many organizations have their own special flavours.

Table 4.2 Test levels

Test level	Level	Environment	Purpose
Hardware unit test	Low	Laboratory	Testing the behavior of hardware component in isolation
Hardware integration test	Low	Laboratory	Testing hardware connections and protocols
Model in the loop	High/low	Simulation models	Proof of concept; testing control laws; design optimization
Software unit test, host/target test	Low	Laboratory, host + target processor	Testing the behavior of software components in isolation
Software integration test	Low	Laboratory, host + target processor	Testing interaction between software components
Hardware/software integration test	High	Laboratory, target processor	Testing interaction between hardware and software components
System test	High	Simulated real life	Testing that the system works as specified
Acceptance test	High	Simulated real life	Testing that the system fulfills its purpose for the user/customer
Field test	High	Real life	Testing that the system keeps working under real-life conditions.

4.1.3 Master test plan

If every test level would define for itself the best things to do, there would be a significant risk that some tests would be performed twice and others would be omitted. In practice, it is not unusual for the system test and the acceptance test to have the same test design techniques applied to the same system specifications – this is a waste of valuable resources. Another risk is that the total test process spends longer on the critical path than is necessary because the planning of the different test levels has not been co-ordinated.

Of course it is better to co-ordinate the various test levels: it prevents redundancies and omissions; it ensures that a coherent overall test strategy will be implemented. Decisions must be made as to which test level is most suited to testing which system requirements and quality attributes. Scarce resources, such as specialized test equipment and expertise, can be allocated to the various test levels for explicitly agreed periods. To achieve this co-ordination and manage the overall test process, an overall test plan has to be drawn up which defines the tasks, responsibilities, and boundaries for each test level. Such an overall test plan is called a *master test plan* (see Figure 4.1). Often a project manager delegates this task to a designated test manager who is responsible for all test activities.

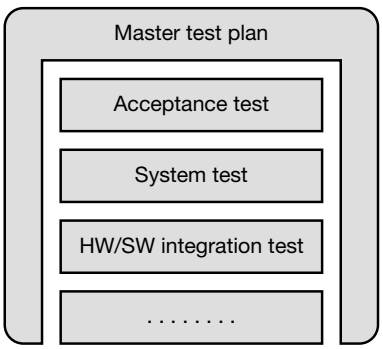


Figure 4.1 Master test plan, providing overall co-ordination of the test levels

A master test plan can be viewed as the combination of *what* has to be tested (test types) and *who* is going to perform the test activities (test levels). After completing the master test plan, for each test level the staff involved will know exactly what is expected of them and what level of support and resources they can expect. The master test plan serves as the basis for the detailed test plan of each test level. It does not need to prescribe for each test level which activities must be performed in detail – that is for the detailed test plans. The master test plan deals with decisions that concern areas where the various test levels can either help or hinder each other. Three areas are of main interest for the master test plan:

- test strategic choices – what to test and how thorough;
- allocation of scarce resources;
- communication between the disciplines involved.

These areas are dealt with in the next section under master test planning activities – respectively, "determine master test strategy," "specify infrastructure," and "define organization."

4.2 Activities

The test manager starts as early in the development process as possible with the preparations for the overall test process. This is done by developing a master test plan. It starts with the formulation of the overall objectives and responsibilities for testing and defining the scope of the master test plan. Then a global analysis of the system to be tested and the development process is carried out. The next step is to use this information and start discussions in the organization concerning which measures to take regarding testing and quality assurance to ensure a successful release of the system. This is the determination of the master test strategy and involves choosing what to do, what not to do, and how to realize it. The test manager's main responsibility is to deliver this master test strategy. To accomplish this, the required resources (infrastructure and personnel) must be defined for the test process. To manage this, communication between all involved disciplines and the required reporting must be defined – in other words, the required organization.

A good master test plan is not created in a quiet and isolated office. It is a highly political task that requires discussion along with bargaining and persuasion throughout the organization. The results of this are recorded in a document that is delivered to all stakeholders. To create a master test plan, the following activities must be completed:

1 formulate the assignment;
2 global review and study;
3 determine the master test strategy;
4 specify infrastructure;
5 define the organization;
6 determine a global schedule.

4.2.1 Formulate the assignment

The objective of this activity is to ensure that the rest of the organization has the correct expectations about what the test organization will do for them. Often a test organization is faced with the distressing situation that they are expected to perform miracles but don't get the means to do it. When an assignment is clearly formulated and well communicated, such situations become rare and can be dealt with better.

Some topics need to be discussed concerning formulating an assignment.

Commissioner

This is the originator of the assignment to draw up a master test plan – or the person (or organizational entity) who decrees that testing should be performed. This person can be thought of as the *customer* of the test team. Usually the commissioner is the general (project) manager for the system development process

who delegates his test responsibility to the test manager. Often, the user organization and product management organization have a customer role. It is important for the test manager to ensure that those concerned understand that being a customer not only means that you may make demands, but also that you have to provide the means (or at least pay for it).

Contractor

This is the person responsible for drawing up the master test plan – usually the test manager.

Test levels

These are the test levels involved in the master test plan. To be considered here are inspections, unit and integration tests (of both hardware and software), system test, functional acceptance tests, and the production acceptance tests. When certain test levels are excluded from the master test plan (e.g. not controlled by the test manager), special attention should be paid to co-ordination and communication with those activities. Also, the development of specific test tools or other components of the infrastructure should be considered.

Scope

This is defined by the restrictions and limitations of the entire test process. For instance:

- unique identification of the information system to be tested;
- interfaces with neighboring systems;
- conversion or porting activities.

Note that it is just as important to state what does *not* belong to the scope of the test assignment.

Objective

The objective of the test process can be described in terms of what will be delivered. For instance:

- services
  - providing management with advice concerning measured quality and the associated risks;
  - maintaining an environment where high priority problems and solutions can be tested immediately;
  - third line helpdesk, for extraordinary problems;
- products to be delivered by each test level.

Preconditions

These describe the conditions imposed on the test process externally, for example:

- *Fixed final date* – the date by which the system must be tested is usually already fixed by the time the test assignment is commissioned.
- *Planning* – the planning for delivery of the test basis, the test object and the infrastructure is usually fixed when the test assignment is commissioned.
- *Available resources* – the customer often sets limits on the personnel, means, budget, and time.

Assumptions

These describe the conditions imposed by the test process on third parties. They describe what is required to make it all possible, for example:

- *required support* – the test process requires various types of support concerning, for instance, the test basis, the test object, and/or the infrastructure;
- *changes in the test basis* – the test process must be informed about coming changes: in most cases this simply means participating in existing project meetings within the system development process.

During the ensuing planning process, the preconditions and assumptions are elaborated in more detail.

4.2.2 Global review and study

This activity is aimed at gaining insight into the (project) organization, the objective of the system development process, the information system to be developed, and the requirements the system should meet. It involves two activities:

- study of the available documentation;
- interviews.

Study of the available documentation

The documentation, made available by the customer, is studied. Consider here:

- system documentation, such as the results of information analysis or a definition study;
- project documentation, such as the planning for the system development process,
- the organization scheme and responsibilities, the quality plan, and (global) size estimates;
- a description of the system development method, including standards;
- a description of host and target platforms;
- contracts with suppliers.

If it is the development of a new release of an existing system, the presence and reusability of existing testware is investigated.

Interviews

Various people involved in the system development process are interviewed. Consider here:

- representatives of product marketing to gain insight into the company objectives and "selling arguments" of the product;
- user representatives to gain insight into the most appreciated functionality and the most irritating flaws of the system;
- field representatives to gain insight into the production environment of the system at intended customer sites;
- the suppliers of the test basis, the test object, and the infrastructure in order to ensure a match between the different disciplines involved.

It is also advisable to consult those who are indirectly involved, such as the accountants or lawyers, the manufacturing manager, the future maintenance organization, etc.

4.2.3 Determine master test strategy

The objective of this activity is to reach a consensus with all stakeholders about what kind of testing is (and is not) going to be carried out, related to the quality the company wishes to be asserted. This has much to do with trading off and making choices. One hundred percent testing is impossible – or at least economically unfeasible – so the company must decide how much risk they are willing to take that the system does not reach the desired quality. The test manager has the important task of explaining to all stakeholders what the impact would be of choosing not to do certain test activities, in terms of added risk to the company.

During this activity it is determined which quality characteristics of the system are more important than others. Then it is decided which (test) measures are best suited to cover those quality characteristics. For the more important quality characteristics, more thorough test design techniques can be applied – possibly more sophisticated tools may be desirable. It is also determined at which test level those measures should be carried out. In this way, the master test strategy sets goals for each test level. It prescribes globally for each level what kind of testing should be performed and how much time and resources are allocated.

The determination of the test strategy involves three activities:

- review current quality management measures;
- determine strategy;
- global estimation of the test levels.

Review current quality management measures

Testing is a part of the total quality management within the system development process. A quality system may be present which, for instance, provides for executing inspections and audits, developing a certain architecture, adhering to certain standards, and following change control procedures. When deciding which test activities are required, the other – not test related – quality management activities, and what they aim at, should also be considered.

Determine strategy

The steps to obtain a test strategy are described here briefly (a detailed description is given in Chapter 7).

- *Determine quality characteristics*. Based on risks, a set of relevant quality characteristics for the system is chosen. These characteristics are the main justification for the various test activities and they must be considered in the regular reporting during the test process.
- *Determine the relative importance of the quality characteristics*. Based on the results of the previous step, the relative importance of the various quality characteristics is determined. This is not a straightforward exercise because relative importance is subjective. Different people will have different notions of the importance of certain characteristics of the particular system. Choosing where to spend scarce resources for testing can cause a clash of interests. For the test process it is important that the stakeholders have experienced this dilemma and that each has the understanding that what is eventually decided may not be perfect but, considering everything, it is the best choice.
- *Allocate quality characteristics to test levels*. In order to optimize the allocation of scarce resources, it is indicated which test level(s) must cover the selected quality characteristics and, roughly, in which way. This results in a concise overview of the kinds of test activities that are going to be performed in which test levels in order to cover which quality characteristics. It can be presented as a matrix – see Table 7.2 for an example.

Global estimation of the test levels

For each test level, it can easily be derived from the strategy matrix which test activities must be performed. Next, a global estimation of the test effort is drawn up. During the planning and control phase of the various test levels, a more detailed estimation is performed.

4.2.4 Specify infrastructure

The aim of this activity is to specify at an early stage the infrastructure needed for the test process, especially the parts required for several test levels or those which have a relatively long delivery time. This involves three activities:

- specify required test environments;
- specify required test tools;
- determine the planning of the infrastructure.

Specify required test environments

A test environment consists of the facilities needed to execute the test, and is dependent on the system development environment and the future production environment (see Chapter 13). In general, the more it looks like the real production environment, the more expensive it is. Sometimes the testing of specific properties of the system require specialized and expensive equipment. The master test plan must define, in general terms, the different test environments and explain which test levels will be allotted which environment.

It is important to state the specific demands for the test process. Consider here, for instance, that random external triggers must be simulated or that a solution must be available to generate specific input signals.

In practice the environment is often a fixed situation and you just have to make do with this. In this case the master test plan must clearly explain the risks involved in the possible shortcomings of the test environment that has to be used.

Specify required test tools

Test tools may offer support to test activities concerning planning and control, constructing initial data and input sets, test execution, and output analysis (see section 14.1). Many tools required by testers are also required by developers, for instance tools that enable manipulation of input signals or analyze specific system behavior. Sharing of such tools and the knowledge of how to operate them should be discussed and agreed.

When a required tool is not yet available it must be specifically developed, and this should be treated as a separate development project. The master test plan must clearly indicate the dependencies between this development project and the test activities in the master test plan.

Determine the planning of the infrastructure

Responsibility for the further elaboration, selection, and acquisition or development of all the necessary parts of the infrastructure will be determined, time lines will be defined, and the agreements will be recorded. In addition, the availability of the various facilities will be outlined.

4.2.5 Define organization

This activity defines the roles, competencies, tasks, and responsibilities at the level of the entire test process. A detailed description of various forms of organization and all aspects with a role there can be found in Chapter 19. Setting up the organization involves three activities:

- determine the required roles;
- establish training;
- assign tasks, competencies, and responsibilities.

Determine the required roles

The roles needed to achieve a better match between the various test levels must be determined. They deal not with the individual test level, but with the overall test process. It is about co-ordination of different schedules, optimal use of scarce resources, and consistent collection and reporting of information. Consider especially:

- general test management and co-ordination;
- centralized control, for instance of infrastructure or defect administration;
- centralized quality assurance.

It is common that disciplines from the line organization are employed for these purposes. For instance:

- test policy management;
- test configuration management;
- methodological and technical support;
- planning and monitoring.

The determination of the test roles within the various test levels is made during the planning and control phase of the test levels concerned.

Establish training

If those who will participate in the test levels are not familiar with basic testing principles or proficient with the specific techniques and tools that will be used, they should be trained (see Chapter 18). The required training can sometimes be obtained from commercial courses, or it can be developed within the organization. The master test plan must reserve sufficient time for this training.

Assign tasks, competencies and responsibilities

Specific tasks, competencies, and responsibilities are assigned to the defined test roles. This applies especially to the tasks related to the tuning between the various test levels and the decisions to be taken. For instance:

- drawing up regulations for the products to be delivered by the various test levels;
- monitoring that the regulations are applied (internal review);
- co-ordinating the test activities common to the various test levels, such as the set-up and control of the technical infrastructure;

- drawing up guidelines for communication and reporting between the test levels, and between the various disciplines involved in the overall test process;
- setting up the methodological, technical, and functional support;
- preserving consistency in the various test plans.

4.2.6 Determine global schedule

The aim of this activity is the design of a global schedule for the entire test process. It includes all test levels (within the scope of the master test plan) and the special activities such as development of infrastructure components and training.

For each test level, the starting and finishing dates, and the deliverables, are stated. In the planning and control phase of the various test levels, the planning is elaborated in detail.

The global schedule should contain at least:

- description of the high-level activities to be carried out (phases per test level);
- deliverables of those activities;
- allocated time (man-hours) for each test level;
- required and available lead time;
- relations with, and dependencies on, other activities (within or outside the test process, and between the various test levels).

The interdependencies between the various test levels are especially important. After all, the execution phases of several test levels are mostly executed sequentially – first unit test, then integration test and system test, and finally acceptance test. Often this is exactly the critical path of the overall test process.

Testing by developers

5

5.1 Introduction

The time is gone when one person could develop a complete system alone. Nowadays, systems are developed by large teams, sometimes divided into teams per subsystem, physically separated, or even situated on different continents. The developed systems are large and complex, and that alone puts quality under pressure. The demand for quality is increasing because of the competitive market or the use of systems in safety critical situations. This explains the need for early and thorough testing. The existence of an independent test team does not mean that testing during the development stage is less important. Both teams have their own important role in the preparation of an end product with the desired quality. An independent test team, in general, executes tests based on the requirements, and their purpose is to provide confidence that the system fulfills those requirements. In contrast, the developers start testing at unit level using knowledge of the internal structure of the software. This knowledge is used again to test the integration of the different units with the purpose of delivering a stable system. According to Beizer (1990) requirements-based testing can, in principle, detect all bugs but it would take infinite time to do so. Unit and integration tests are inherently finite but cannot detect all defects, even if completely executed. Only a combination of the two approaches together with a risk-based test strategy can detect the important defects. This makes both types of tests essential in producing systems with the desired quality. The reasons why testing by developers is very important are listed below.

- Early detected defects are easy to correct. In general, the cost of fixing defects will rise in time (Boehm, 1981).
- High quality basic elements make it easier to establish a high quality system. Low quality basic elements, on the other hand, will lead to an unreliable system and this can't be solved practically by functional tests.
- Defects detected during post development stages are difficult to trace back to source.
- Defects detected during post development stages have to be corrected and this will lead to time consuming regression tests.

- Good testing during the development stage has a positive influence on the total project time.
- Straight testing of exception handling is only possible at unit level, where exceptions can be triggered individually.

Basically, quality cannot be added to the product at the end by testing – it should be built in right from the start. Checking the quality at every development stage by testing is a necessity.

Testing is not perceived by many developers as the most rewarding task during development. To keep the test effort for developers at an acceptable level, testing must be effective and efficient. To do this an integration strategy should be chosen. Integration strategies are described in section 5.2. Furthermore, all essential activities have to be planned and controlled as described in section 5.3.

5.2 Integration approach

Unit testing alone is not sufficient to get a stable, functional, correct system. Many defects are related to the integration of modules. If the requirements are not formally described, everyone has to make their own interpretation of those requirements. This is no problem as long as these interpretations are not related to interactions with other modules. Incorrect interaction between modules is often caused by these interpretations. The best instrument to detect these defects is the integration test. An integration strategy is a decision about how the different modules are integrated into a complete system. The integration covers hardware and software. It is important to decide which strategy to use because of all the dependencies between different software modules, different hardware parts, and between the software and hardware. At a certain moment, all these parts have to be ready for integration. Just when this moment happens depends on the strategy followed so the decision about the one to follow should be made as early as possible because it has a strong effect on the scheduling of project activities. Big bang, bottom-up, and top-down integration are the three fundamental strategies. However, because these three are not mutually exclusive there arises a variety of strategies through combinations of these three. The integration strategy depends on:

- availability of the integration parts (e.g. third party software or hardware);
- size of the system;
- whether it is a new system or an existing system with added/changed functionality;
- architecture.

Big bang integration

This strategy can only be successful if:

- a large part of the system is stable and only a few new modules are added;
- the system is rather small;
- the modules are tightly coupled and it is too difficult to integrate the different modules stepwise.

This strategy is basically quite simple – all modules are integrated and the system is tested as a whole.

The main advantage is that no stubs or drivers have to be used. With this strategy, it is difficult to find the causes of defects and the integration can only start if all the modules are available.

Bottom-up integration

This strategy is useful for almost every system. The strategy starts with low-level modules with the least number of dependencies, using drivers to test these modules. This strategy can be used to build the system stepwise, or first build up the subsystems in parallel and then integrate them into a complete system. The integration can start very early in the development process. This is certainly so if the planning of the project is such that the modules are also delivered in a bottom-up fashion. This approach will lead to an early detection of interface problems and these problems can be isolated rather easily and are therefore much cheaper to correct than if they are discovered only when the complete system is ready. A major disadvantage is that many drivers have to be used in carrying out this strategy. It is also very time consuming because of the iteration of tests.

Top-down integration

In this strategy, the control structure of the system takes the lead. The control structure is developed in a top-down sequence and this offers the ability to integrate the modules top-down starting with the top-level control module. At every new level, the connected modules at the corresponding level are integrated and tested. The role of the as yet non-existent modules is implemented with stubs. Changed requirements, which have impact on low-level modules, may lead to changes in top-level modules. This may lead to a (partial) restart of the integration process and its testing. Another disadvantage is the number of stubs needed to test every integration step. An advantage of top-down integration is that, even though major parts of the system are still substituted by stubs, an early 'look ad feel' of the entire system can be achieved."

Centralized integration

This type of integration is used when:

- a central part of the system is necessary for the rest of the system to function (for instance, the kernel of an operating system);
- the central part is necessary to run tests and it is too difficult to substitute this part with a stub;
- the architecture of the system is such that the central part of the system is developed first and made ready for production. After that new modules or subsystems are released to upgrade the system or add completely new functionality.

The central part of the system is tested first. The next step is the integration of the control structure. The coupled subsystems are tested in parallel according to a bottom-up or top-down strategy. The bottleneck in this approach is the integration of the central part of the system. Sometimes the only usable integration strategy is the big bang strategy because the modules of this part are often very tightly connected. This strategy can only be successful if the backbone components are tested very thoroughly. A very strong point of this approach is the combination of the different strategies and the ability to choose the most effective one for every system part.

Layer integration

This strategy is used for systems with a layered architecture. In this, interfacing only happens between the layer directly underneath and that above. Each layer is tested in isolation using the top-down, bottom-up, or big bang strategy. The next step is the integration of the layer according to a top-down or bottom-up strategy. The advantages and disadvantages are the same as for the top-down and bottom-up strategies. The integration and isolation of interfaces is much easier and therefore discovery of the causes of defects is also easier.

Client/server integration

This strategy is used for client/server architectures. The client is integrated either top-down, bottom-up, or big bang and the server is substituted by a stub and driver. For the server, the same approach is chosen – finally the server and client are integrated.

Collaboration integration

A collaboration is a collection of objects which work together to achieve a common purpose, for instance the realization of a use case. The system supports many collaborations because it has to realize many use cases. Many objects belong to more than one collaboration. This means that a clever choice of collaborations can cover the complete system . Collaboration integration is only usable for object oriented systems. This strategy is only usable if the collabora-

tions are clearly defined and cover all components and interfaces. The latter is also a disadvantage because the subtle dependencies between different collaborations can be left out of the collaboration model. This strategy is partly a big bang implementation because all the components of one collaboration are put together, and only when the collaboration is complete does integration testing start. Only a few tests are needed because the focus is on end-to-end functionality. Because of the overlap of collaborations, it is often not necessary to test every single one. Because a group of related components are tested, none or only a few stubs and drivers are needed to execute the test.

5.2.1 Application integrator

Although the unit test and the integration test are two different test levels, they are strongly related. The quality of the unit test has a strong influence on the efficiency of the integration process. The scope of these two test levels and what to do are described in the master test plan. In practice someone is still needed to co-ordinate these two levels. The person who does this is called the application integrator (AI). The AI is responsible for the progress of the integration process and for the quality of the system delivered. The AI together with the customer (the development project leader or team leader) decide on the level of quality desired. This quality level is made measurable in the form of exit criteria. To meet these exit criteria it is necessary that the integration parts possess a certain level of quality before they are used in the integration process. This desired quality level is the entry criteria for the integration part. The AI does an intake on every integration part to determine whether it meets the entry criteria. This is not another test – the deliverer of the integration part (the developer) must prove that the entry criteria are met. The system is only released when all the exit criteria are met (see Figure 5.1).

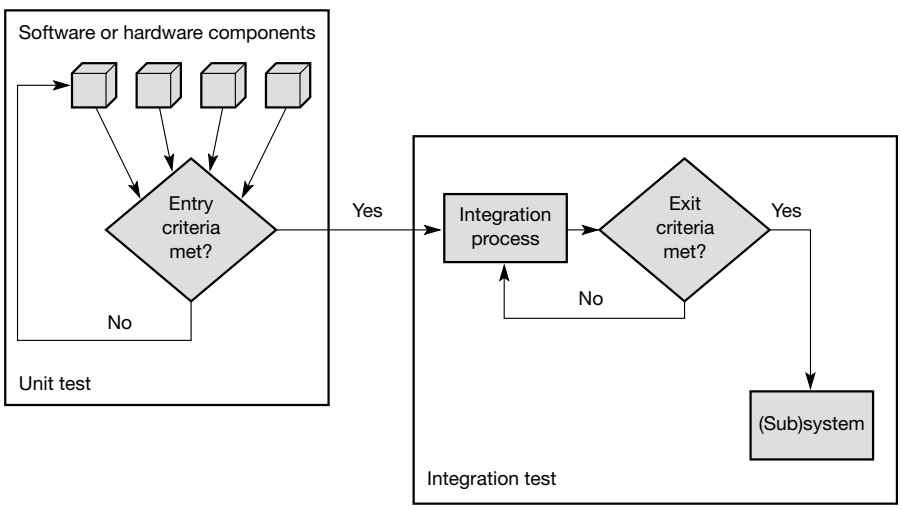


Figure 5.1
The relationship between the unit test and the integration test

The entry and exit criteria are the most important instruments for a successful integration process. The more concrete these criteria are, the better they can be used. The entry criteria give clarity about what a developer should deliver and when it is good enough to be accepted. The exit criteria give clarity about when the integration process is successfully finished. When these criteria are used in such a manner then they become powerful control instruments. These criteria may include aspects of the quality characteristics to be tested, the desired degree of coverage, the use of certain test design techniques, and the evidence to be supplied. If the criteria are detailed enough, they provide useful information for the development of the test strategy of the unit and integration tests. It is possible that integration processes are carried out in parallel – for instance, with two subsystems that are integrated into the complete system at a certain moment. The way that the AI works is very much determined by the integration strategy chosen.

The maximum span of control of the AI turns out to be ten developers – with more an additional AI is needed. More AIs also mean more possibilities for using combinations of integration strategies. Also, when hardware and software integration are part of the process, at least two AIs are needed because different knowledge is needed for the software and the hardware.

In practice this approach has shown that later tests yield a significantly smaller number of serious defects. Sometimes it is possible to start with the next test level before the integration process is finished. This is only possible if there confidence has built up in the system under construction. The developers get feedback about their work at a much earlier stage than before. This offers the opportunity to improve their performance by learning from their mistakes, although not every defect is related to mistakes made by developers. The early detected defects are much cheaper to correct. Also, this approach provides more stable modules, which has a very positive effect on the stability of the complete system.

The AI can only function well if that is the only role they have in the project. The AI has the delegated responsibility for the quality of the system and this can conflict with the responsibilities of the development project leader who has to deliver a system in time within a certain budget. Sometimes they are counterparts and, based on the information they supply, the commissioner has to decide what is of more importance – quality, time-to-market, or money.

This approach forces a conscious choice about the quality desired and the tests to carry out before delivery to the next test level. The tests carried out by developers are becoming more transparent and usually there is one person who has full responsibility for testing within the development team.

5.3 Lifecycle

The lifecycle for testing by developers is less structured and demanding than that for independent test team. However, as projects become bigger, or the risks involved are growing, the lifecycle becomes more and more important as an instrument for remaining in control. In practice this will mean that there is no longer any difference between lifecycles for independent test teams and for developers.

5.3.1 Planning and control

The planning and control phase consists of the following activities:

- formulating assignment;
- establishing test basis;
- defining test strategy;
- specifying test deliverables;
- setting up the organization;
- specifying infrastructure;
- organizing control;
- setting up schedules;
- consolidating and maintaining test plan;
- controlling the test;
- reporting.

Formulating assignment

The customer (commissioner) and the contractor are identified. The scope of the tests is determined and the preconditions are described.

Establishing test basis

All documents (functional and technical design, source code, change requests, standards) needed to fulfill the assignment are identified. The latest versions of these documents should be available and these should be up to date. These documents are the basis for designing test cases. The coding standards are used to configure the static analyzer (see Section 14.2.4.7) to detect deviations from these standards. This is extremely important when a subset of a language is used because of safety.

Defining test strategy

The test strategy dictates what, how, and to what depth tests should be executed. It is also the basis for time and resource allocation. The test strategy for unit and integration tests are based on the strategy defined in the master test plan. Functionality, performance, and maintainability are the main quality attributes tested during unit and integration tests. For functionality tests, the test design techniques "control flow test" and "elementary comparison test" are very useful. For performance, statistical usage testing is used while maintainability is covered with checklists. The unit test focuses mainly on the functionality of the unit, while the integration test focuses mainly on functionality involving interfaces and on the performance of the system or system parts.

Specifying test deliverables

In collaboration with the commissioner, it is decided what the deliverables are and how information concerning the status of the product under test and progress is made available. Test deliverables can be test cases, reports, coverage measurements, automated test scripts, etc.

Setting up the organization

The unit and integration tests require a structured organization. The primary test tasks and the responsibility for the test should be delegated. Often, the development project leader is responsible for the unit and integration tests. Unfortunately, the development project leader is also responsible for delivering on time and budget. Quality testing is time consuming and is expensive because of that. When a project is under time pressure, the first activity to cut is testing. By introducing the AI role someone else is given responsibility for testing and for the quality of the system delivered. The development project leader no longer has the power to cut testing on his own and should work with the AI to produce a progress report for the commissioner. The commissioner can use this information to decide what to do.

Specifying infrastructure

Unit tests and integration tests can make specific demands on infrastructure – for instance, coverage tools for unit testing (see section 14.2.4.10). Stubs and drivers (see section 14.2.4.5) should be developed and sometimes a simulation environment (see Chapter 13) must be built. For unique projects it is sometimes necessary to develop dedicated test tools. All these development activities should be planned and started on time otherwise they will not be available when testing must start.

Organizing control

Control is part of the regular development activities because the test activities are part of the development process. Defect management as used by an independent test team is too heavy an instrument to use during development. However, if products such as units or completely integrated systems are delivered with defects, these defects should be described in a document or a defect management tool.

Setting up schedules

The allocation of time and resources affords the opportunity to control the test process. The commissioner can be informed about progress. Delays can be recognized in time and measures taken to get back on schedule.

Consolidating and maintaining test plan

All the activities mentioned above give the information needed to write a test plan. This is a dynamic document and should be adjusted whenever the project status demands.

Controlling the test

Controlling the test is an important activity during the whole process. There are three different forms of control: control of the test process, infrastructure control, and test deliverables control. Controlling the test process is essential if the right system with the right quality is to be delivered on time.

Reporting

Periodically, and on demand, the commissioner is informed about the process and quality of the test object. The progress and quality reports are often combined. This report should provide information about:

- total number of test cases;
- number of test cases still to be executed;
- number of successfully completed test cases;
- what has been tested and which defects still have to be corrected;
- deviations from the test strategy and why they happened.

5.3.2 Preparation phase

The two main elements of the preparation phase are:

- testability review of the test basis;
- describing the infrastructure.

Testability review

Implied in testability are completeness, consistency, accessibility of the test basis, and whether the information is available to derive test cases or not. Generally, checklists are used for the testability review.

Describing the infrastructure

Integration tests are not usually executed in the real environment to avoid complications and control problems. Instead, the real environment is simulated. This simulated environment (infrastructure) must be ready on time and it should include all kinds of measures to facilitate testing. For more details on simulation see Chapter 13.

Sometimes the first stage in the development of a system is a model. This model (for instance, state transition diagrams) is used to generate code (automatically). Testing and debugging is much more effective when the models are examined at the same level of abstraction at which they were designed (Douglass, 1999). This means that testing starts before any code is written. Some modeling tools support the execution of the models within the tool environment. It would be even better if the model could be executed in a simulated environment within the tool. If this early testing has advantages for the project team, a modeling tool should be selected which support this kind of testing.

5.3.3 Specification phase

During the specification phase, test cases are derived using test design techniques. There are two major activities during the specification phase:

- specifying tests;
- creating utilities.

Specifying tests

Individual unit or integration step test cases are created on the basis of the assigned test design techniques. If no test design techniques are prescribed this does not mean that testing is unnecessary. The developer has to design and execute tests based on knowledge of the unit under test. It is possible that defects in the test basis are detected during specification documentation. These defects should be notified and clarified.

With XP (Beck, 2000), tests are specified in the form of small test programs. The test programs are specified before the actual coding of the desired software is started. The test object passes the test if the test programs do not detect any defects.

Creating utilities

To test units and integration steps it is often necessary to use stubs and drivers. These can be developed during the specification phase. If a generic architecture is used to construct the testbed, it is possible to connect the testbed to an automated test suite (see section 14.2.4.5). This test suite can now be used to construct the automated tests more easily (for more details see Chapter 15).

If a simulator is needed and has to be built specifically for the project, its development should start much earlier. The development of such a simulator takes a considerable amount of time and should be finished before test execution.

5.3.4 Execution phase

During the execution phase, the test cases are executed and the results are recorded. The actual results are compared with the expected results, and if there is a difference this is analyzed. If the difference is due to a malfunctioning of the test object, the problem should be resolved. The stop criteria for the unit and those for the integration test are a little different.

The stop criteria for the unit test is described as the entry criteria for the integration test. The unit test stops if the unit fulfills the entry criteria of the integration test.

The integration test is stopped when all the separate parts are integrated and the system under test meets the exit criteria of the integration test.

The unit developer usually carries out the unit test. This individual is familiar with the internal operation of the system and therefore can organize testing efficiently. However, the developer is more likely to overlook certain defects because of familiarity or over-confidence. A fellow developer who tests this unit can overcome this problem, although probably not able to carry out the test as fast as the unit developer. Two principles of XP can overcome these problems. The first is to test programming by building a test program before the actual development of the unit. The developed unit is considered to be correct if it passes the test program without errors. The second is to use peer programming, where two developers develop a unit together and test and correct it on the fly.

5.3.5 Completion phase

The completion phase is used to report on the status of the test object. If the quality of the test object meets the criteria, it is handed over to the next test level. The testware is adjusted where necessary and conserved for a subsequent release or similar product.

6 Testing by an independent test team

6.1 Introduction

Independent test teams are mostly occupied with high-level tests. These tests occur at the end of the development lifecycle. This is the reason why the test activities of these teams are on the critical path of the development process. This also means that some instruments which shortens the time for testing on that critical path would be very welcome – a lifecycle model is such an instrument. It makes a clear distinction between supporting activities and test execution activities. The supporting activities have the objective of keeping the time needed for test execution to a minimum. The supporting activities are kept outside the critical path by executing them in parallel with development. If this process is planned well, the test object is delivered at the same time as the test team finishes the description of the test cases.

The lifecycle described here is rather formal, because many stakeholders are involved. Most of the information needed by the test team should be provided from outside the test team. Also, not every stakeholder has the same objective and expectation as the test team. Sometimes the outcome of the test process is subject to an external audit, because of certification of the product. These are all reasons for formalizing the test process.

6.2 Planning and control phase

Objective

Co-ordination, monitoring, and control of the test process in order to provide insight into the quality of the test object within the given time and budget and with the appropriate personnel.

Procedure

The planning and control phase starts during the functional specification or design phase of the system. For planning, all kind of activities are executed and these are described below in detail. The outcome is the basis for the test plan.

The test plan should be based on the master test plan, if there is one. If none is available then the test plan is developed independently from other test levels.

The test plan contains the planning, allocated resources, and scope of the assignment. It is therefore the basis for co-ordination, monitoring, and control of the test process. An example of a test plan is shown in Appendix E.

Activities

The planning and control phase consists of the following activities:

1 formulating the assignment;
2 global review and study;
3 establishing the test basis;
4 determining the test strategy;
5 setting up the organization;
6 specifying test deliverables;
7 specifying the infrastructure;
8 organizing management and control;
9 scheduling the test process;
10 consolidating the test plan.

The following activities can be identified within the framework of the co-ordination, monitoring, and control of the test process:

11 maintaining the test plan;
12 controlling the test;
13 reporting;
14 establishing detailed schedules.

Figure 6.1 shows the dependencies between the activities of the phase planning and control.

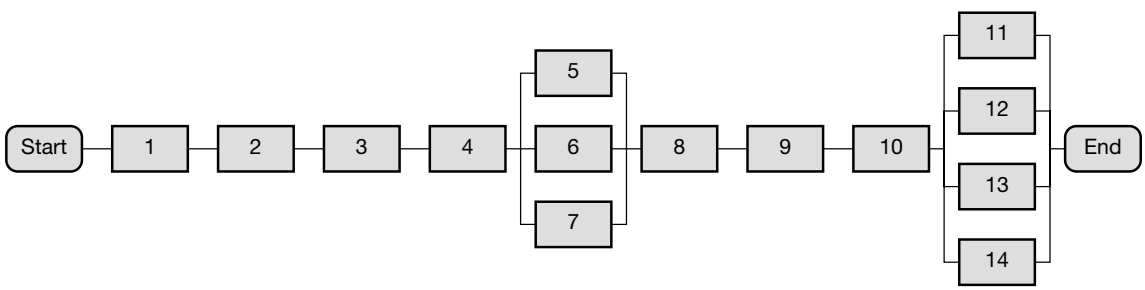


Figure 6.1 Planning and control activities

6.2.1 Formulating the assignment

Objective

The objective is to determine who the commissioner and the contractor are, what the scope and the objective of the test process is, and what the preconditions are for the test process.

Procedure

The following issues are determined.

- *Commissioner* – the provider of the test assignment and therefore also the person who should be informed about progress and problems, and who should agree with any readjustments.
- *Contractor* – the person who is responsible for carrying out the test assignment.
- *Scope* – what the test object is, but also what is not subject to test. For example:
  - unique identification of the system to be tested;
  - interfaces with adjoining systems.
- *Objective* – the expected output of the test process. For example:
  - the products to be delivered;
  - the quality characteristics to be tested: for example, functionality, maintainability, and performance.
- *Preconditions* – there are two types of preconditions. The first is those imposed on the test process (external). The second comprises those imposed from within the test process (internal).
  Examples of external preconditions include:
  - *fixed deadline*: the deadline for finishing testing is fixed, because the start of production and/or product introduction is already scheduled;
  - *schedules*: the test assignment is often given if the development process is already on the move – as a result the schedules for the delivery of the test basis, the test object, and the infrastructure are already set;
  - *allocated resources*: the commissioner often sets limits on the human resources, budget, and time.

  Examples of internal preconditions include:
  - *support regarding the test basis*: there must be someone available to answer questions concerning uncertainties about the test basis;
  - *support regarding the test object*: the immediate support of the development team is necessary when defects block testing. This type of defect must be repaired with high priority because they are a serious threat to progress;
  - *changes in the test basis*: the test team must be informed immediately of any changes to the test basis – in most cases, this simply means linking up with the existing procedures in the system development process.

6.2.2 Global review and study

Objective

The objective of this activity is to gain insight into the available system and project documentation, the demands made on the system in terms of functionality and quality, the organization of the system development process, the available knowledge and experience in the field of testing, and, with respect to the acceptance test, the user demands.

Procedure

The procedure consists of two activities.

- *Studying the available documentation*. If the test process concerns a maintenance test, there will also be an investigation into the presence and usefulness of existing testware.
- *Conducting interviews*. The commissioner, marketing manager (representing the customer), and head of development are interviewed to provide the necessary background information about the test object and the requirements it has to fulfill.

6.2.3 Establishing the test basis

Objective

The objective of this activity is to determine what the test basis and the basic documentation are.

Procedure

Select the relevant documentation and determine the status of the documents and their delivery schedule. To be able to perform a test properly and to complete the test assignment successfully, it is necessary to know the requirements with respect to the functionality and quality of the system. All the documentation that describes these requirements is included in the test basis.

6.2.4 Determining the test strategy

Objective

The objective of this activity is to determine what to test, how to test it, and with what coverage.

Procedure

The assignment is converted into the concrete approach of a test process. The procedure consists of the following activities.

- *Strategy development*. A test strategy describes the choices which have to be made in respect of which parts of the system have to be tested thoroughly and which less thoroughly (for further details see Chapter 7).

- *Drawing up a budget*. Based on the strategy, and taking into account the available resources, a reliable budget is drawn up for the test process. Test processes have many uncertainties and it may be useful to anticipate this by incorporating an "unforeseen" item. This item generally amounts to between 10 and 15 percent of the entire budget.

6.2.5 Setting up the organization

Objective

The objective of this activity is to determine how the organization of the test process will be set up: roles, tasks, authorities, responsibilities, hierarchy, consultation structures, and reporting lines. The need for training will also be considered. More details on the organization of a test team can be found in Chapters 17, 18, and 19.

Procedure

- The specific tasks, authorities, and responsibilities in the test process are described and assigned to the test roles. The tasks are related to the activities described in this lifecycle model.
- The relationship between the various test functions, the internal relationships within the test team, and the relationships with the other parties involved in the system development process are described.
- When it has been decided which test roles should be carried out in the test process, the staff required are recruited, taking into consideration their skills and the knowledge and skills of each test role (see Chapter 17). One person can perform more than one role.
- Establishing training courses if knowledge of structured testing or automated testing is not sufficient.
- Hiring external personnel if knowledge is not available within the organization or it is too time consuming to train current employees.
- Establishing reporting lines.

6.2.6 Specifying test deliverables

Objective

The objective of this activity is to specify the products to be delivered by the test team.

Procedure

Specifying the test deliverables involves two activities.

- *Establishing the testware*. Testware is defined as all test documentation that is produced during the test process and can be used for future releases, and which therefore must be transferable and maintainable. In regression tests, for example, use is often made of existing testware. Examples of testware are test plans, logical and physical test design, test output.

- *Setting up standards*. Standards and agreements on the naming of the test deliverables are determined. When possible, templates are made for the various documents or existing templates are used.

6.2.7 Specifying the infrastructure

Objective

The objective of this activity is to determine at an early stage the required infrastructure for the test process.

Procedure

The procedure consists of three activities.

- *Specifying the test environment*. The required test environment is described. It contains the facilities needed to perform the test (see Chapter 13).
- *Specifying the test tools*. The required test tools are described. These offer support to the test activities with respect to planning and control, carrying out tests, and analyzing the results (see Chapters 14 and 15).
- *Establishing the infrastructure schedule*. For all the required components of the infrastructure, it is determined who is responsible for their detailing, selection, and acquisition – agreements must be recorded. A schedule is also set up, specifying the times when items must be made available to the test team.

6.2.8 Organizing management and control

Objective

The objective of this activity is to describe how management and control of the test process, the infrastructure, and the test deliverables will be arranged.

Procedure

The procedure consists of the following activities:

- specifying the test process control;
- specifying the infrastructure control;
- specifying the test deliverables control;
- specifying defect procedure.

These are described in detail in Chapter 20.

6.2.9 Scheduling the test process

Objective

The objective of this activity is to give a detailed description of time, money, and personnel needed for the test activities.

Procedure

The procedure consists of two activities.

- *Setting up an overall schedule*. Working on the basis of the established (hours) budget, the available resources, and the delivery schedules of the various system parts and documentation, an overall schedule is set up for the test process. Both the recruited personnel and the products are assigned to the activities to be carried out in the test process.
- *Setting up a financial schedule*. The financial consequences are scheduled in terms of personnel and infrastructure.

6.2.10 Consolidating the test plan

Objective

The objective of this activity is to record the results of the activities carried out so far, and to acquire formal approval from the commissioner.

Procedure

The procedure consists of the following activities:

- identifying threats, risks, and measures;
- establishing the test plan;
- establishing the change procedure for the test plan;
- consolidating the test plan.

Possible threats for the test process in the test plan must be identified. Threats can be related to the following:

- *Feasibility* – to what degree are the proposed test plans and schedules of the various suppliers feasible and realistic?
- *Testability* – to what degree is the expected quality of the test basis sufficient for the completion of the test process?
- *Stability* – to what degree will the test basis be subject to changes during the test process?
- *Experience* – to what degree is the test team's experience or knowledge level adequate for carrying out the test process properly?

The test plan lists the measures that have been taken for each risk. These include preventive measures taken to avoid risks, but also any detective measures to identify risks early.

Establishing the test plan

The results of the activities carried out so far are recorded in the test plan. A test plan contains the following sections (see Appendix E):

- assignment
- test basis

- test strategy
- planning
- threats, risks, and measures
- infrastructure
- test organization
- test deliverables
- configuration management
- *Appendices*
  - change procedure for the test plan
  - justification of the (hours) budget.

A management summary that outlines the strategy, schedules, budget, threats, risks, and measures is optional.

Establishing the change procedure for the test plan

A change procedure is set up with regard to the approved test plan. This procedure details both the criteria and the required authority for changes.

Consolidating the test plan

The test plan is submitted to the commissioner for approval. It is advisable to record the approval formally through the signatures of both the test manager and the commissioner. In addition, a presentation for the steering group and other parties involved may be helpful in obtaining approval and, maybe even more important, support within the organization.

6.2.11 Maintaining the test plan

Objective

The objective of this activity is to keep the test plan and the overall schedule up to date.

Procedure

Maintenance of the test plan is carried out at a time when changes that lead to adjustment of the test plan in accordance with the established criteria are introduced.

- *Readjusting the test plan and/or test strategy*. Changes to the test plan affect practically all activities carried out in the test process. The test strategy, in particular, is subject to change. A good reason to change the strategy would be that certain tests find many more, or fewer, defects than expected. It is then decided to specify extra tests and carry them out, or to carry out planned tests only in part, or even to cancel them altogether.

  The changes are recorded in a new version of the test plan or in a supplement, which must also be submitted to the commissioner for approval.

It is the responsibility of the test manager to communicate clearly the consequences of the changes to the commissioner.

- *Maintaining the schedule*. The most common and expected changes are those regarding the schedule. Reasons for change can be related to the availability of test basis, test object, infrastructure, test personnel, etc. Also the quality of the test basis or system under test can give cause to adjust the schedule.

6.2.12 Controlling the test

Objective

The objective of this activity is to control the test process, infrastructure, and test deliverables, in order to be able to provide constant insight into the progress of the test process and the quality of the test object.

Procedure

In accordance with the procedures established in the test plan, the test process, infrastructure, and test deliverables are subject to control.

6.2.13 Reporting

Objective

The objective of this activity is to provide the organization with information about the progress of the test process and the quality of the system under test.

Procedure

Periodically, and ad hoc on request, reports are provided on the progress of the test process and the quality of the test object. The test plan lists the form and the frequency of reports (see Chapter 20). The progress and quality reports contain data relating to the most recent reporting period and the accumulated data from the entire test process. Ad hoc reports are tailor-made at the request of the commissioner. The reports may contain the following elements:

- How much of what was indicated in the test plan has been tested?
- What still needs to be tested?
- Can any trends be recognized with respect to the quality of the test object and the defects found?

6.2.14 Establishing detailed schedules

Objective

The objective of this activity is to set up and maintain the detailed schedules for the various phases: preparation, specification, execution, and completion.

Procedure

The detailed schedule should include at least the following aspects for each phase:

- activities to be carried out;
- links with and dependencies on other activities (inside or outside the test process);
- time allocated per activity;
- required and available overall project time;
- products to be delivered;
- personnel involved.

6.3 Preparation phase

Objective

The most important objective of the preparation phase is to determine if the test basis has sufficient quality for the successful specification and execution of the test cases (testability).

Preconditions

The test basis should be available and fixed.

Activities

The following activities are distinguished in the preparation phase:

1 testability review of the test basis;
2 defining test units;
3 assigning test design techniques;
4 specifying the infrastructure.

Figure 6.2 shows the dependencies of these activities.

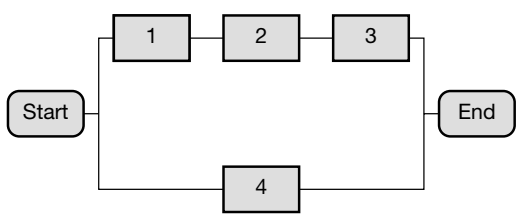


Figure 6.2 Preparation phase activities

6.3.1 Testability review of the test basis

Objective

The objective of this review is to establish the testability of the test basis. Testability here means the completeness, consistency, and accessibility, and also convertibility into test cases.

Procedure

The testability review is described briefly below. A more extensive description is given in Chapter 8.

- *Selecting relevant documentation*. The test basis is collected as listed in the test plan. If any changes have been introduced in the meantime, the test plan must be adjusted.
- *Drafting checklists*. In relation to the test strategy, checklists are prepared for the various subsystems. These checklists function as guides for the assessment of the test basis.
- *Assessing documentation*. The test basis is assessed according to the checklist. If the basis is not sufficient, the commissioner and the provider of the test basis are informed.
- *Reporting*. The results and recommendations of the assessment are reported to the commissioner and the provider of the test basis.

6.3.2 Defining test units

Objective

The objective of this activity is the division of the subsystems into independent testable units.

Procedure

The procedure regarding the definition of test units involves two activities.

- *Determining test units*. Functions within a subsystem which have a logical coherence or high dependency are grouped into one test unit. Very large functions are not combined this way to avoid inconveniently large test units.
- *Realize a test unit table*. For each subsystem it is indicated which test units it contains. This subdivision is visualized in a test unit table such as that shown in Table 6.1.

Table 6.1
Sample test unit table

System	Test unit
Subsystem A	Test unit A1
	Test unit A2
	Test unit A3
	etc.
Subsystem B	Test unit B1
	Test unit B2

6.3.3 Assigning test design techniques

Objective

The objective of this activity is to use the test strategy to allocate test design techniques to the test units.

Procedure

The table drawn up in the previous activity (Table 6.1) is extended using the allocated test design techniques (see Table 6.2). It is possible that more than one technique is allocated to one test unit.

Table 6.2 Sample test table extended with allocated test design techniques

System	Test unit	Test design technique
Subsystem A	Test unit A1	Classification tree method
	Test unit A2	State transition testing
	Test unit A3	Elementary comparison test
	etc.	Elementary comparison test
Subsystem B	Test unit B1	Statistical usage test
	Test unit B2	Statistical usage test
		State transition testing

6.3.4 Specifying the infrastructure

Objective

The objective of this activity is to realize a detailed description of the infrastructure required.

Procedure

The description of the test infrastructure in the test plan is detailed when necessary and, together with suppliers' agreements, a delivery schedule is made. If some parts of the infrastructure have to be developed in-house, a development plan is established.

6.4 Specification phase

Objective

The objective of the specification phase is to build up a test set using the allocated test design techniques.

Preconditions

The following conditions should be met before the specification phase can be started:

- the test basis should be available and fixed;
- the delivery schedule of the test object and the infrastructure are available for the creation of the test scenario.

Activities

The following activities are distinguished in the specification phase:

1 deriving test cases;
2 drafting test scripts;
3 producing a test scenario;
4 specifying the entry check of the test object and infrastructure;
5 installing the infrastructure.

Figure 6.3 shows the dependencies between the activities of the specification phase.

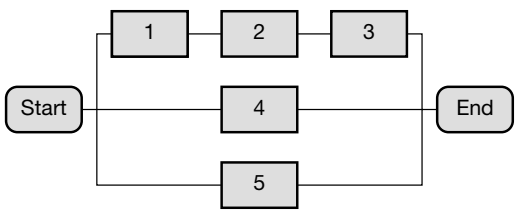


Figure 6.3 Specification phase activities

6.4.1 Deriving test cases

Objective

The objective of this activity is to derive test cases for each test unit based on the allocated test design techniques.

Procedure

The test cases are derived using the allocated test design techniques. It should be determined in parallel if the various test cases can be executed independently or influence the outcome of each other. The test design is prepared according to the standards described in the test plan. During the test design phase the shortcomings of the test basis can be detected. These defects are reported using the defect management procedures described in the test plan.

6.4.2 Drafting test scripts

Objective

The objective of this activity is to convert the test cases described in the test design into executable, concrete test actions. This includes establishing the sequence of actions and conditions for their execution in test scripts.

Procedure

The test cases are converted into executable and verifiable test actions ordered in the correct sequence. Ideally the actions are independent so that the situation where a failed action blocks a large part of the script is avoided.

The test script should describe at least the preconditions and the execution actions.

6.4.3 Producing a test scenario

Objective

The objective of this activity is to record the sequence in which the test scripts will be executed in a test scenario.

Procedure

The test scenario is the roadmap for test execution. It describes the execution sequence of the test scripts and in doing so becomes the control instrument for test execution. The test scenario can be extended by the allocation of test scripts to individual testers. The interdependencies between the different test scripts should be kept to a minimum.

To detect the most important defects, the test scripts that relate to the most crucial parts of the system, as recorded in the test strategy, are executed during the initial phase of the test execution if possible.

The test scenario must be a flexible living document. Many things can cause change in the test scenario – for instance blocking faults, malfunctioning of the infrastructure, absence of testers, etc.

6.4.4 Specifying the entry check of the test object and infrastructure

Objective

The objective of this activity is to describe how the entry checks of the test object and the infrastructure are to be carried out.

Procedure

A checklist is prepared showing all the items that should be delivered to the test team. This list is used to perform the test object entry check to determine if it is complete and whether the test can start or not.

A similar activity is carried out with respect to the specifications of the test infrastructure.

A pretest test script is prepared. The execution of this test script determines whether the test object is stable enough to start test execution. Stable, in this context, means that the test script is executed with no defects detected.

6.4.5 Installing the infrastructure

Objective

The objective is to install the infrastructure according the specifications.

Procedure

The infrastructure is installed parallel to the other activities of the specification phase. Execution of this activity usually consists of the following elements:

- solving bottlenecks and problems, and recording any measures taken in new agreements;
- entry check of the infrastructure;
- installation check;
- trial run.

6.5 Execution phase

Objective

The objective of the execution phase is to execute the specified test scripts in order to gain insight into the quality of the test object.

Preconditions

The infrastructure is installed and the test object is delivered to the test team.

Activities

The following activities are distinguished in the execution phase. They are summarized in Figure 6.4:

1 entry check of test object/infrastructure;
2 executing (re)tests;
3 comparing and analyzing the test results;
4 maintaining the test scenario.

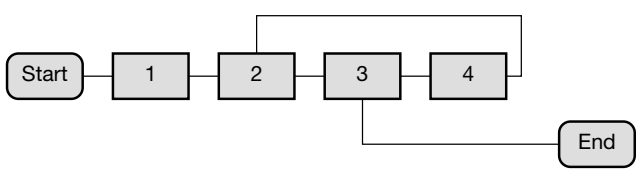


Figure 6.4 Execution phase activities

6.5.1 Entry check of test object/infrastructure

Objective

The objective of this activity is to determine whether the delivered components of the test object and the infrastructure function are ready to use for testing.

Procedure

The entry check of the infrastructure is only carried out if this activity is not executed during the specification phase. It is preferable to do this in the specification phase because if problems are detected there is enough time to solve them before test execution starts.

The completeness of the delivered test object is checked using the prepared checklists. Missing items are reported and a decision made whether to continue the entry check or not.

If the test object is successfully installed and the infrastructure ready to use, the prepared test cases of the entry check are executed. Defects are reported immediately and should be corrected with the highest priority. A successful completion of the entry check is a precondition to start test execution. In other words, the test execution phase does not start until the entry check is completed successfully.

6.5.2 Executing (re)tests

Objective

The objective of this activity is to obtain test results to assess the quality of the test object.

Procedure

The test scripts are executed in the sequence specified in the test scenario. Discipline is very important throughout the execution of the test. The tests should be carried out as they are described in the scenario and the test scripts. If the testers deviate from the test scripts, there is no guarantee that the test strategy will be carried out correctly. As a consequence it is not possible to state if the risks have been covered as intended.

In addition to test scripts, checklists are used to carry out static tests.

6.5.3 Comparing and analyzing the test results

Objective

The objective of this activity is to detect unexpected behavior of the test object and analyze the cause of this.

Procedure

The actual results are compared with the expected results and the result of this comparison is recorded. If there is difference, this should be analyzed. There are various causes of differences:

- a test execution error, which means that the test concerned must be executed again;
- a test design error;
- a programming error;
- shortcomings in the test environment;
- an inconsistency or obscurity in the test basis.

Only if a difference is due to a malfunctioning of the system is this defect reported, according to the defect procedure described in the test plan.

6.5.4 Maintaining the test scenario

Objective

The objective of this activity is to keep the test scenario up to date so that it is clear at all times which test scripts must be executed and in what order.

Procedure

During the execution of the (re)tests, problems may be detected that have consequences for the execution of the tests. First, in connection with possible test defects, it has to be determined if the testware must be adapted and the test rerun. The rerun of the test is included in the test scenario. Any rework with respect to the testware is initiated. Second, defects nearly always lead to the inclusion of a retest in the test scenario. It is important, however, to determine how the retest should be executed. Full or partial re-execution of a test script depends on the following:

- the severity of the defects;
- the number of defects;
- the degree to which the previous execution of the test script was disrupted by the defects;
- the available time;
- the importance of the function.

Maintaining the test scenario is of great importance. Not only does it provide insight into the test scripts that still have to be executed, but it also forms the basis for the changes that have to be introduced into the detailed schedule of the execution phase and the overall schedule of the test process as a whole.

6.6 Completion phase

Objective
There are several objectives in the completion phase:

- archiving the testware, so that it can be reused for subsequent tests;
- obtaining experience figures for the benefit of better control of future test processes;
- finishing the final report, informing the commissioner of the course of the test and discharging the test team.

Preconditions
The test execution must be completely finished, including retests.

Activities
The completion phase consists of the following activities. These are summarized in Figure 6.5:

1 evaluating the test object;
2 evaluating the test process;
3 archiving testware;
4 discharging the test team.

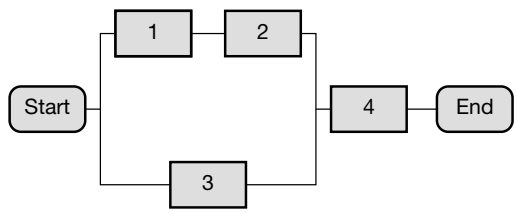


Figure 6.5 Completion phase activities

6.6.1 Evaluating the test object

Objective
The objective of this activity is to evaluate the quality of the test object in order to provide the final release advice.

Procedure
On basis of the executed tests, the test reports, and the (status of the) registered defects, the final release advice is drawn up. It is important to indicate which defects are still unsolved ("known errors") and which risks are related to these defects. This means that any alternatives should be pointed out, such as postponement, providing a lower functionality level, etc.

6.6.2 Evaluating the test process

Objective

The objective of this activity is to obtain insight into how the test process went and to collect experience data for future test processes.

Procedure

After the execution phase, the test process is evaluated to gain insight into the strong and weak points of the executed process. This evaluation can be the basis for improvements. The results of the evaluation are reported. The final report should include at least the following aspects.

- *Evaluation of the test object*. The results of the evaluation of the test object are displayed. It is particularly important to list any unsolved defects and the related risks.
- *Evaluation of the test process*. The results of the evaluation of the test process are displayed. The following subdivision can be made:
  - evaluation of the test strategy;
  - to what degree the chosen test strategy has been deviated from;
  - was the chosen test strategy the correct one: which components of the system have been tested too much or too little?
- *Scheduling versus realization*.
  - to what degree has the schedule been realized?
  - have any structural deviations been detected?
- *Evaluation of resources, methods, and techniques*.
  - to what degree have the selected resources been used?
  - have the selected methods and techniques been used correctly and are there any defects relating to the methods and techniques?

6.6.3 Archiving testware

Objective

The objective of this activity is to select and update the testware created to be usable for future tests.

Procedure

The testware is collected and adjusted when necessary. It is stored and a backup is made. If a new release is planned, the testware is handed over to the project leader of the new release. The archiving of testware offers the opportunity to reuse test scripts and to achieve the same coverage with less effort the next time.

6.6.4 Discharging the test team

Objective

This is the formal completion of the test process and provides the test team with a discharge statement.

Procedure

On the basis of the evaluation report and the testware that has been handed over, the commissioner is asked to officially end the test process and to provide the test team with a discharge. After receiving the discharge, the test team is disbanded.

PART III

Techniques

Introduction

A technique basically describes how a certain activity must be performed. Many testers will automatically think of techniques as "testing a system," meaning exercising the system and evaluating its behavior. That is just one type of technique. In principle, for each activity that is identified techniques can be developed to support that activity. This part provides a selection of useful techniques that are applicable to various phases of the test lifecycle.

It starts off (Chapter 7) with the technique for executing a risk-based test strategy in the planning and control phase. This technique forces the organization to make choices as to what to test and what not to test. It is a very important technique that helps test managers to ensure that testing is focused on the "right things" and that the organization has realistic expectations about what testing can achieve.

The testability review technique (Chapter 8) is applied in the preparation phase. A testability review can save much time and effort from being wasted in trying to design test cases "on quicksand." It addresses the issue "Do we have sufficient and reliable information as a basis for designing our tests?"

Chapter 9 describes how to organize and perform formal inspections. This is a generally applicable technique for evaluating and improving the quality of documents. It can be applied as a testability review but is applicable to all kinds of system documentation throughout the development lifecycle.

Chapter 10 describes some techniques that are commonly applied to the analysis of safety critical aspects. These techniques are also applied to analyzing the vulnerability of the system in case of malfunctioning components. Since safety analysis is often organized separately, but parallel to, other testing activities, this chapter also describes a separate lifecycle for safety analysis and relates that to the testing lifecycle.

The most substantial chapter concerns test design – how to derive test cases from certain system specifications. Chapter 11 offers a variety of test design techniques that can be used in different circumstances for different purposes. It provides a good starting set of techniques, which will be sufficient for many test projects. Nevertheless, many more test design techniques have been invented and the reader is encouraged to peruse the available literature and publications on this subject. The introduction to this chapter explains the general principles of test design techniques and in what respects they are different and achieve different goals.

Finally Chapter 12 provides a variety of checklists that can be used throughout the complete testing lifecycle.

Risk-based test strategy

7

7.1 Introduction

Developing a risk-based test strategy is a means of communicating to the stakeholders what is most important about this system for the company, and what this means for the testing of this system. "Just test everything" is either theoretically impossible, or at least economically unfeasible – it would be a waste of resources (time, money, people, and infrastructure). In order to make the best possible use of resources, it is decided on which parts and aspects of the system the test emphasis should fall. The risk-based test strategy is an important element in a structured test approach and contributes to a more manageable test process.

Priorities must be set and decisions made about what is important and what isn't. Now this sounds very easy and straightforward, but it has a few problematic questions:

- what do we mean by "important?"
- what are the things that we evaluate to be important or not?
- who makes these decisions?

What do we mean by important? Isn't it a very subjective concept? With a risk-based test strategy, evaluating importance is about answering the question: "How high is the (business) risk if something is NOT OK in this area?" A risk is defined as the chance of a failure occurring related to the damage expected when it does occur. If a system is of insufficient quality, this may imply high damage to the organization. For instance, it could cause loss of market share, or the company may be obliged to call back millions of sold products to replace the faulty software. Therefore this situation forms a risk for the organization. Testing helps to cover such risks by providing insight into the extent to which a system meets the quality demands. If certain areas or aspects of the system imply high risks for the organization, then more thorough testing is obviously a solution. Of course, the reverse also holds – no risk, no test.

What are the things that we evaluate as important or not? In the test strategy, the relative importance of two kinds of things are evaluated – the subsystems and the quality attributes. Basically, statements such as "the signal

decoding subsystem is more important than the scheduling subsystem" and "usability is more important than performance" are made. A subsystem can be any part of a system, which is conveniently treated separately for the purpose of strategy decisions. Usually the architectural decomposition of the system is used for this purpose. Quality attributes describe types of required system behavior, such as functionality, reliability, usability, etc. Standards for the definition of quality attributes exist, such as ISO 9126 (see www.iso.ch).

Who makes these decisions? The task of establishing a risk-based strategy is much like a diplomatic mission. Many different personnel, or departments with different interests, are involved – sales, maintenance, personnel, developers, end users, etc. They all have their own ideas about what is important. If they are not listened to, they can make things difficult for the test project, claiming "I don't agree and I am going to fight it." Getting people involved in the decision-making process, creates higher commitment and decreases the risk of political trouble later in the project. This doesn't mean that the perfect test strategy is the one that satisfies everyone's wishes. That would probably mean that everything imaginable was considered important by someone and thus everything has to be tested after all. A better definition of the best test strategy is "that which satisfies no one completely, but on which everyone agrees that, all things considered, it is the best compromise."

Now it is true that establishing a test strategy involves much negotiating and making acceptable choices, but some things are just not negotiable. The company may impose compliance with certain *standards* on the development and testing process. Some industries are required to comply to certain industry standards for certification of their product. For instance the DO-178B standard (RTCA/DO-178B, 1992) is required for the safety critical software in the avionics industry. Many standards exist that can help an organization to fill in parts of their test strategy (Reid, 2001). For instance the software verification and validation standard IEEE 1012 relates integrity levels of software to required test activities. The process for determining such integrity levels is defined by the ISO 15026 standard. Other useful standards are IEEE 829 which defines a broad range of test documentation, BS7925-1 which is a software testing vocabulary, and BS7925-2 which defines and explains several test design techniques.

7.2 Risk assessment

Developing a test strategy requires insight into risks. What will the consequences be when the authentication module does not work correctly? What will the damage be when the system shows insufficient performance? Unfortunately, risk assessment in our industry is not a science. Risk cannot be calculated and expressed in absolute numbers. Nevertheless, this section may help to give the required insight into risks to enable the development of a sound test strategy.

For the purpose of test strategy, risks are assessed on the basis of quality characteristics and subsystems. In order to do that, separate aspects of risk are analyzed which are derived from the well-known equation (Reynolds, 1996):

$$\text{Risk} = \text{chance of failure} \times \text{damage}$$

where the chance of failure is related to aspects including frequency of use and the chance of a fault being present in the system. With a component that is used many times a day by many people, the chance of a fault showing itself is high. In assessing of the chance of faults occurring, the following list (based partly on (Schaefer, 1996)) may be helpful. It shows the locations where faults tend to cluster:

- complex components;
- completely new components;
- frequently changed components;
- components for which certain tools or techniques were employed for the first time;
- components which were transferred from one developer to another during development;
- components that were constructed under extreme time pressure;
- components which had to be optimized more frequently than usual;
- components in which many defects were found earlier (e.g. in previous releases or during earlier reviews);
- components with many interfaces.

The chance of failure is also greater for:

- inexperienced developers;
- insufficient involvement of user-representatives;
- insufficient quality assurance during development;
- insufficient quality of low-level tests;
- new development tools and development environment;
- large development teams;
- development teams with suboptimal communication (e.g. owing to geographical spread or personal causes);
- components developed under political pressure, with unresolved conflicts in the organization.

Next the possible damage must be estimated. When the system fails (does not meet its quality requirements), what will be the damage to the organization? This damage can take the form of cost of repair, loss of market share caused by negative press, legal claims, foregone income, etc. An attempt should be made to translate each form of damage into money terms. This makes it easier to express the damage in numerical terms and to compare the related risk with other assessed risks.

The required information for risk assessment is usually obtained from different sources. Some people know how the product is used in its intended environment, about frequency of use and possible damage. These include end users, support engineers, and product managers. Project team members such as architects, programmers, testers, and quality assurance staff, know best what the difficulties are in developing the product. They can provide information useful in assessing the chance of a fault.

Because of the complexity of the matter, it is impossible to assess risks with complete objectivity and accuracy. It is a global assessment resulting in a relative ranking of the perceived risks. It is therefore important for the risk assessment to be carried out not only by the test manager, but also by as many of the other involved disciplines and stakeholders in the project. This not only increases the quality of the test strategy, but has the added advantage that everyone involved has a better awareness of risks and what testing can contribute towards handling them. This is extremely important in "setting the right expectations with regard to testing." It must be understood that testing is just one of the ways of managing risks. Figure 7.1 shows the wide range of risk management options (Reynolds, 1996). The organization must decide which risks it wants to cover with which measures (including testing) and how much it is willing to pay for that.

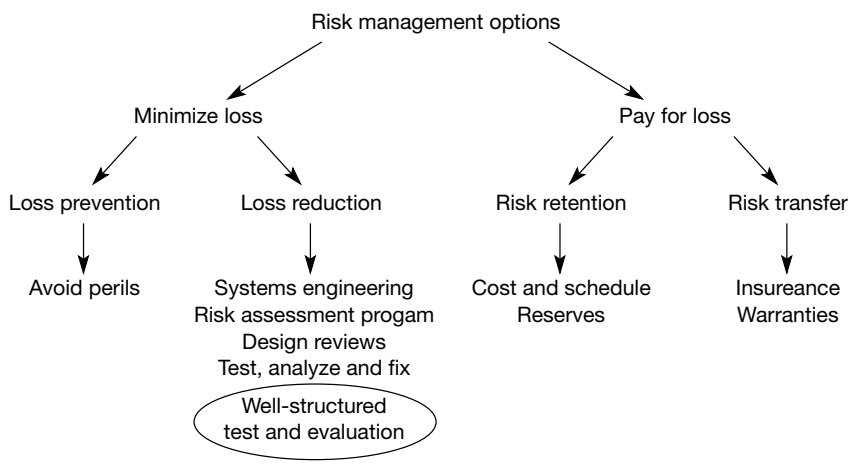


Figure 7.1
Treatment of risks

7.3 Strategy in master test planning

The objective of a test strategy in master test planning is to get organization-wide *awareness* of the risks that need to be covered and *commitment* on how much testing must be performed where and when in the development process. The following steps must be taken:

- selecting the quality characteristics;
- determining the relative importance of the quality characteristics;
- assigning the quality characteristics to test levels.

7.3.1 Selecting the quality characteristics

In a co-ordinated effort with all parties involved, a selection of quality characteristics is made on which the tests must focus. Major considerations here are business risks (see section 7.2), and organizational and international standards and regulations. The selected quality characteristics must be addressed by the tests that are going to be executed. The tests are expected to report on those quality characteristics – to what extent does the system meet the quality requirements, and what are the risks of not conforming to them.

A starting set of quality characteristics that covers the possible quality demands must be available. International standards, such as ISO 9126, provide a useful basis for the organization to produce its own set of definitions that suits the specific culture and kind of systems it develops. Because many different people are going to discuss the importance of a particular quality characteristic and what to do about it, it is important that everybody is on the same wavelength. This can be promoted by discussing the following topics for each quality characteristic and documenting the results.

- *Clarify the quality characteristic*. For instance, with "performance" do we mean how fast it responds or how much load it can handle? In the case of both aspects being considered relevant, then defining a separate quality characteristic for each will prevent confusion.
- *Provide a typical example*. Formal definitions, especially when copied from international standards, can be too academic for some involved in the development of a test strategy. It helps to give an example of what this quality characteristic could mean for this specific system.
- *Define a way of measuring the quality demand*. During testing it must be analyzed to what extent the system meets quality requirements. But how can this be established? Some time should be spent on examining how to establish how well the system behaves with regard to a specific quality characteristic. If this cannot be defined, then it cannot be expected that the test organization reports anything sensible and reliable about this.

7.3.2 Determining the relative importance of the quality characteristics

After selecting the initial set of quality characteristics, the importance of each in relation to the others must be discussed. Again, the assessed risks are the basis for deciding how important a quality characteristic is for this system. The result of this discussion can be described in a matrix, where the importance of each quality characteristic is indicated as a percentage. An example of this is given in Table 7.1, where a typical organization-specific set of definitions is used.

Table 7.1 Matrix of relative importance of quality characteristics

Quality characteristic	Relative importance (%)
Connectivity	10
Efficiency (memory)	–
Functionality	40
Maintainability	–
Performance	15
Recoverability	5
Reliability	10
Security	–
Suitability	20
Usability	–
Total	100

The objective of such a matrix is to provide a general picture and a quick overview. The quality characteristics with the high percentages will get priority attention in the test process, while those with low percentages will get a little attention when time allows. The advantage of filling in the matrix is that the organization stakeholders are forced to abandon a rather generalized attitude and decide in concrete terms about what is important.

There is a natural tendency to have a cautious opinion when it comes to quality. Most people don't like to say that something is "not important." This may lead to a matrix filled with lots of small numbers, that obscures the general overview. Therefore, a minimum of five percent should be stipulated as a guideline. When a quality characteristic scores zero, this does not necessarily mean that it is totally irrelevant, but that it is not wise to spend scarce resources on it.

7.3.3 Assigning the quality characteristics to test levels

The previous steps have identified which quality characteristics are most important for a system and should form the focus of the test process. Now, the whole test process consists of many test activities performed by various testers and test teams at various times and in various environments (see section 4.1.2). In order to optimize the allocation of scarce resources, it must be discussed which test level(s) must cover the selected quality characteristics and, roughly, in which way. This results in a concise overview of the kinds of test activities to be performed in which test levels in order to cover which quality characteristics.

This can be presented as a matrix with the test levels as rows and the quality characteristics as columns. Each intersection indicates how this quality characteristic is covered in this test level using the following symbols:

++	the quality characteristic will be covered thoroughly – it is a major goal in this test level
+	this test level will cover this quality characteristic
(empty)	this quality characteristic is not an issue in this test level

Table 7.2 provides an example of such a strategy matrix. This example shows, for instance, that functionality is the most important quality characteristic (which it often is) and must be tested thoroughly in the unit test and the system test. The hardware/software integration test will rely on this and merely perform shallow testing of functionality. It will focus more on testing how the components interact together technically and with the environment (connectivity), and how the system recovers when a component fails (recoverability).

Table 7.2 Example of a strategy matrix of a master test plan

	Functionality	Connectivity	Reliability	Recoverability	Performance	Suitability
Relative importance (%)	40	10	10	5	15	20
Unit test	++			+		
Software integration test	+	++				
Hardware/software integration test	+	++		++		
System test	++		+		+	
Acceptance test	+			++		++
Field test			++		++	

Again, the strength of this matrix is that it provides a concise overview. If desired, more detailed information can be provided, in plain text. For instance, the required use of specific tools or techniques, by adding footnotes to the matrix.

7.4 Strategy for a test level

The objective of a test strategy for a particular test level is to make sustained choices as to what to test, how thoroughly, and which test techniques to apply. The test strategy must explain to all the parties involved why testing the system in this way is the best possible choice, considering time pressure and scarce resources. The choices are not arbitrary but can be related to what the organization finds most important (in terms of business risks). Properly executed, the

development of a test strategy is not just a technical exercise, but also has a highly political aspect – it assures the stakeholders that testing is not a goal in itself but is for the benefit of the whole organization. At the same time, it makes them aware that they will not get the perfect test but the best test considering the circumstances.

Ultimately the test strategy results in defining specific test techniques applied to specific parts of the system. Each can be seen as a concrete test activity and can be planned and monitored separately. This makes the test strategy an important management tool for the test manager.

The following steps must be taken:

1 selecting the quality characteristics;
2 determining the relative importance of the quality characteristics;
3 dividing the system into subsystems;
4 determining the relative importance of the subsystems;
5 determining test importance per subsystem/quality characteristic combination;
6 establishing the test techniques to be used.

If a master test plan, including a test strategy, is available, then the test strategy for a particular test level must be based on this. In that case, steps 1 and 2 will be a relatively easy translation of what has already been discussed and decided on a global level. The test strategies for the various test levels can be seen as the more concrete refinements of the global goals that were set in the master test strategy. If a test strategy for the master test plan is not available, the test manager of a particular test level is obliged to initiate and co-ordinate discussions about business risks and quality characteristics, just as would be done for a master test plan.

7.4.1 Selecting the quality characteristics

A list of relevant quality characteristics is determined. The same instructions apply as for the corresponding step in the strategy development for the master test plan.

7.4.2 Determining the relative importance of the quality characteristics

The importance of each selected quality characteristic in relation to the others is determined. The results are described in a matrix, where the importance of each quality characteristic is indicated as a percentage. Again, the same instructions apply as for the corresponding step in the strategy development for the master test plan.

Table 7.3 is an example of a matrix showing the relative importance of the quality characteristics – it is based on the results from the master test plan (see Table 7.2). The numbers in this example differ from those in the master test plan because not all quality characteristics are tested at this particular test level, however the numbers still have to total 100 percent. Also, if two characteristics

have the same relative importance at the level of the master test plan, they do not necessarily have to be tested with the same relative importance at this test level.

Quality characteristic	Relative importance (%)
Functionality	40
Performance	25
Reliability	10
Suitability	25
Total	100

Table 7.3 Matrix of relative importance of quality characteristics for a particular test level

7.4.3 Dividing the system into subsystems

The system is divided into subsystems that can be tested separately. Although the term "subsystem" is used here and in the rest of this chapter, one can also think of "component" or "functional unit" or other such terms. The bottom line is that quality demands are not the same for individual parts of the system. Moreover, the various subsystems may differ in terms of risks for the organization.

In general, this dividing step follows the architectural design in which the systems components and their relationship are already specified. Deviations from this in the test strategy should be clearly motivated. Examples of alternative criteria for division are the extent of risk, or the order of release by the developer. If, for instance, a data conversion is part of the implementation of the new product, then the conversion module can be treated as a separate subsystem.

The total system is added to the list of subsystems. This serves to indicate that some quality demands can be evaluated only by observing the behavior of the complete system.

7.4.4 Determining the relative importance of the subsystems

The relative importance of the identified subsystems is determined in much the same way as for quality characteristics in step 2. The risks for each subsystem are weighted and the results documented in a matrix (see Table 7.4 for an example). It is not about exact percentages in this matrix, but rather of getting an overview of how important the different subsystems are. It should not express the personal view of the test manager, but rather of the people who know how the product is used in its intended environment (such as end users and product management). This step is useful in helping to force these people to form an opinion about what they deem to be important.

Table 7.4 Matrix of relative importance of subsystems

Subsystem	Relative importance (%)
Part A	30
Part B	10
Part C	30
Part D	5
Total system	25
Total	100

7.4.5 Determining the test importance per subsystem/quality characteristic combination

This step refines the strategy by combining the assessments of quality characteristics and subsystems. For example, a refinement may be that performance is an important quality characteristic (rating 25 percent) but that this holds predominantly for subsystem B (which, for instance, takes care of routing image data) and not at all for subsystem D (which, for instance, logs statistical data). This kind of information helps to pinpoint areas where testing should focus even more closely.

The result of this refinement step can be presented as a matrix with the quality characteristics as rows and the subsystems as columns. Each intersection indicates how important this quality characteristic is for this subsystem using the following symbols:

- ++ the quality characteristic is predominant for this subsystem
- + the quality characteristic is relevant for this subsystem
- (empty) this quality characteristic is insignificant for this subsystem

Table 7.5 Matrix of relative importance per subsystem and quality characteristic

Relative importance (%)		Part A	Part B	Part C	Part D	Total system
	100	30	10	30	5	25
Functionality	40	++	+	+	+	+
Performance	25	+	++	+		+
Reliability	10		+			++
Suitability	25	+		+	+	++

Table 7.5 gives an example of such a matrix. It is emphasised once more that test strategy development is not a mathematical exercise. The purpose is to assist in choosing where to put in the major efforts and which areas to ignore or touch briefly. The strength of a matrix such as shown in Table 7.5 is that it provides a

concise overview of what the organization finds important and where testing should focus. It reflects the choices that the organization has made during the complex process of analyzing risks, weighing consequences, and allocating resources.

7.4.6 Establishing the test techniques to be used

The final step involves the selection of the test techniques that will be used to test the subsystems on the relevant quality characteristics. The matrix that resulted from the previous step (see Table 7.5) can be regarded as a "contract" – the test team is expected to organize and execute a testing process that covers what the organization has defined to be important. There will be rigorous testing in one area and more global testing in others, in accordance with what is specified in the matrix.

Usually, most of the system's functions are tested by means of test cases specifically designed for that purpose (dynamically explicit testing). To this end, many test design techniques are available (see Chapter 11). Testing can also be done by analyzing metrics during the development and test processes (dynamically implicit testing) or by assessing the installed measures on the basis of a checklist (static testing). For instance, performance can be tested *during* the testing of functionality by measuring response times with a stopwatch. No explicit test cases were designed to do this – it is tested *implicitly*. Also, security, for instance, can be tested statically by reviewing the security regulations.

It is the responsibility of the test manager to establish the set of techniques that "can do the job." Where the matrix shows a high importance, the use of a rigorous technique that achieves a high coverage, or the use of several techniques, is an obvious choice. A low importance implies the use of a technique with lower coverage, or dynamically implicit testing. Just which techniques are the best choice depend on various factors, including:

- *Quality characteristic to be tested.* For instance, a technique may be very good at simulating realistic usage of the system, but very poor at covering the variations in functional behavior.
- *Area of application*. Some techniques are for testing the human machine interaction, while others are better suited to test all functional variations within a certain component.
- *Required test basis*. Some techniques require that a specific kind of system documentation is available, for instance state transition diagrams or pseudocode or graphs.
- *Required resources*. The application of a technique requires a certain allocation of resources in terms of human and machine capacity. Often this also depends on the "required knowledge and skills."
- *Required knowledge and skills*. The knowledge and skills of the testing staff can influence the choice of techniques. The effective use of a technique sometimes requires specific knowledge and/or skills of the tester. In depth domain expertise can be a prerequisite, and some techniques require a trained (mathematical) analytical talent.

The introduction to Chapter 11 provides more detailed information about such aspects. The test manager's experience and knowledge of the various test design techniques and their strengths and weaknesses is vital in establishing the required test techniques. It is not unusual to find that available test techniques must be tuned or combined into new test techniques that suit a specific test project. The selection and tuning of the techniques must be done early in the test process, allowing for the test team to be trained in required areas.

The result of this step is the definition of the techniques to be used in the test on each subsystem. See Table 7.6 for an example. Each + in the table signifies that the corresponding technique will be applied to this particular part of the system. From a management point of view this is a very useful table because each + can be treated as a separate entity that can be planned and monitored. Each can be assigned to a tester who is then responsible for all necessary activities, such as designing test cases and executing tests. For large systems this table is usually further detailed by subdividing the system parts into smaller units that can be properly handled by a single tester. Optionally – particularly in large test projects – the test strategy is not specified in such detail until the preparation phase.

Table 7.6 Establishing which techniques to apply to which part of the system

Applied test technique	Part A	Part B	Part C	Part D	Total system
W	+	+	+		
X		+	+		
Y	+			+	+
Z					+

In order to ensure that the most important tests are executed as early as possible, this step also determines the order of priority of the established tests (technique–subsystem combination).

7.5 Strategy changes during the test process

The test manager must not have the illusion that the test strategy will be developed once and then remain fixed for the duration of the project. The test process is part of a continuously changing world and must react properly to those changes.

In the real world, development and test projects are often put under pressure, especially in the later stages of a project. Suddenly the schedule is adjusted – usually the available time is reduced. The test manager is asked to perform fewer or shorter tests. Which tests can be canceled or carried out in less depth?

Using the test strategy as a basis, the test manager can discuss these topics with the stakeholders in a professional way. When changed circumstances require that testing should be other than previously agreed, then the expectations of just what testing should achieve must also change. Strategy issues must be re-evaluated – has our notion of which aspects are more important than others changed? Are we willing to accept the increased risk if testing is reduced in a certain area?

A similar situation arises when the release contents change – for example, extra functionality may be added to the system, or the release of parts of the system delayed, or the product is now also targeted for another market segment with different requirements and expectations. In such cases the test strategy must be re-evaluated – is the testing, as it was planned, still adequate for this new situation? If the relative importance of the quality characteristics or the subsystems changes significantly, the planned tests must be changed accordingly.

Another reason to change the test strategy can be because of the results of the testing itself. When testing of a certain part of the system shows an excessive number of defects, it may be wise to increase the test effort in that area – for instance, add extra test cases or apply a more thorough test technique. The inverse is also true – when excessively few or only minor defects are found, then it should be investigated if the test effort may be decreased. In such situations a major discussion issue should be "what is the risk as we perceive now through testing, and does this justify a decision to increase or decrease testing effort?"

7.6 Strategy for maintenance testing

The technique described above for developing a test strategy can be used for a newly developed system without reservation. A legitimate question to ask is to what extent the steps described are still useful for a maintenance release of a system that has been developed and tested before.

From the perspective of test strategy, a maintenance release differs mainly in the *chance* of a fault happening. During maintenance there is a risk that faults are introduced with changes in the system. Those intended changes of system behavior must, of course, be tested. But it is also possible that the system, which used to work correctly in the previous release, doesn't work in the new release as a side effect of the implemented changes. This phenomenon is called *regression*. In maintenance releases, much of the test effort is dedicated to testing that previous functionality works correctly – this is called regression testing. Because the chance of a fault occurring changes when the product enters the maintenance stage, so does the associated risk. This in turn changes the relative importance of the subsystems. For example, a subsystem had a high importance when it was newly developed, because of its business critical nature. The subsystem is unchanged in the maintenance release, so the associated risk is now low and the subsystem may be given a low importance for this maintenance release.

Therefore, in developing a test strategy it is often useful to substitute the concept of "subsystem" with "change." These changes are usually a set of "change requests" that will be implemented, and a set of "known defects" that are corrected. For each change, it is analyzed which system parts are modified, which may be indirectly affected, and which quality characteristics are relevant. Various possibilities exist for testing each change – depending on the risks and the desired thoroughness of the test:

- a limited test, focused only on the change itself;
- a complete (re)test of the function or component that has been changed;
- a test of the coherence and interaction of the changed component with adjacent components.

With every implementation of change, the risk of regression is introduced. Regression testing is a standard element in a maintenance test project. Usually a specific set of test cases is maintained for this purpose. Depending on the risks and the test budget available, a choice has to be made to either execute the full regression test set or to make a selection of the most relevant test cases. Test tools can be used very effectively to support the execution of the regression test. When the regression test is largely automated, selecting which regression test cases to skip is no longer necessary because the full regression test set can be executed with limited effort.

The decision to formulate the test strategy in terms of change requests instead of subsystems depends largely on the number of changes. When relatively few are implemented, the testing process is better managed by taking those changes as basis for risk evaluations, planning, and progress tracking. When subsystems are heavily modified by many implemented changes, they can be treated in the same way as when they were newly developed. Then it is usually preferable to develop the test strategy based on subsystems. If it is decided to formulate the test strategy in terms of change requests, then the steps for developing a test strategy are as follows:

1 determining changes (implemented change requests and corrected defects);
2 determining the relative importance of the changes and regression;
3 selecting quality characteristics;
4 determining the relative importance of the quality characteristics;
5 determining the relative importance per change (and regression)/quality characteristic combination;
6 establishing the test techniques to be used.

Table 7.7 shows an example of the matrix that results from the first two steps. The other steps are similar to those described in section 7.4.

Changes/regression	Relative importance (%)
Change request CR-12	15
Change request CR-16	10
Change request CR-17	10
Defect 1226, 1227, 1230	5
Defect 1242	15
Defect 1243	5
…	30
Regression	10
Total	100

Table 7.7
Matrix of the relative importance of changes and regression

The example shows that regression has been assigned a value of 10 percent. It should be noted that this reflects the importance that is allocated to the issue of regression. It does not mean that 10 percent of the test process should be regression testing. In practice the total effort of regression testing is much larger than that. The reason is that, usually, a relatively small amount of the system functionality is changed while a much larger part remains unchanged. As noted earlier in this section, automation of the regression test can be a tremendous help in freeing scarce time and resources for maintenance testing.

8 Testability review

8.1 Introduction

During the planning phase, the test team determines which documentation forms the test basis. The main activity of the preparation phase is the testability review of the test basis. Testability means the completeness, unambiguouity, and consistency of the documentation that forms the test basis. High testability is a prerequisite for successful progress during specification of the test. The review has the purpose of determining if the quality of the documentation is sufficient to serve as a basis for testing.

The documentation that forms the test basis is the first deliverable of the design process. It is the first opportunity to test. The sooner defects are found the simpler and cheaper it is to correct them. Defects not detected and corrected in the documentation will lead to major quality problems later in the development process. As a consequence, much time consuming and expensive rework must be done to solve these problems and meeting the release date is threatened.

8.2 Procedure

A testability review contains the following steps:

- selecting the relevant documentation;
- composing a checklist;
- assessing the documentation;
- reporting the results.

8.2.1 Selecting relevant documentation

The test plan should identify the documentation to be used for deriving test cases. However, the preparation of the test plan and the preparation phase can have some overlap. Also, if the test plan is already finished, changes may have occurred. Therefore the testability review starts with a formal identification of the test basis and the actual collection of the documentation.

8.2.2 Composing a checklist

For the testability review, checklists should be used. The checklists depend on the test design techniques to be used. The test plan should provide information about the test design techniques to be used and which system parts they should be applied to.

Chapter 12 provides examples of checklists for the various test design techniques along with other checklists. These can be combined into a single checklist to prevent the parts of the test basis from being reviewed more than once. It is probably necessary to compose a new checklist for each organization, each project, and each test type.

8.2.3 Assessing the documentation

With the composed checklist, the test team assesses the documentation and raises a defect report for every one detected. The early detection of the defects creates the opportunity to improve the quality of the test basis even before system development is started. It also gives a thorough insight into the nature and size of the system, which can lead to adjustment of the test plan.

8.2.4 Reporting the results

Based on the detected defects, the test team delivers a testability review report. This report provides a general summary with regard to the quality of the documentation. The possible consequences of low quality should also be described. The following sections are distinguished in the testability review report:

- *Formulating the assignment*. Identification of the test basis, and descriptions of the commissioner and the contractor (i.e. those responsible for executing the assignment). The test manager is likely to be the contractor.
- *Conclusion*. The conclusion regarding testability of the reviewed documentation and any related consequences and/or risks – has the test basis sufficient quality to ensure that designing tests will be useful?
- *Recommendations*. Recommendations regarding present documentation and any structural recommendations to improve the quality of future documentation.
- *Defects*. The detected defects are described or references are made to the corresponding defect reports.
- *Appendix*. The checklists which were used.

8.2.5 Follow-up

The testability review should not result in a statement by the test team that it is impossible to test the system. The consequence of a serious lack of quality in the test basis is that there isn't enough information to apply the demanded test design techniques. The recommendation depends on the risk involved in the system part to which the documentation is related. If there is a low risk, a recommendation could be to apply a less formal test design technique. A high risk

almost automatically means that the quality of the documentation should be raised and rework on the documentation becomes essential. What to do if there is hardly any documentation to form the test basis is described in the next section.

8.2.6 Non-ideal test basis

Sometimes the requirements are not yet available. Innovative ideas for new market are one of the causes of the lack of requirements. In these kinds of projects, the requirements evolve during the development of the product. Another reason for a lack of requirements is a hangover from the past when systems were small and most were developed and tested by the same person. Nowadays the realization of embedded systems involves a team effort and the description of requirements is essential to manage the project and to deliver the right information to the test team.

In these situations it is a waste of time to apply a testability review. The preparation phase is then merely a period of gathering the right information. The test strategy informs the test team about identified risks in relation to subsystems and test design techniques. This kind of information can be used as a guideline for gathering the right information to serve as a test basis. The information can be gathered by interviewing developers and stakeholders – also by being a member of the design process.

9 Inspections

9.1 Introduction

Inspection is a formal evaluation technique in which a person or group *other than the author* examines software requirements, design, or code in detail to detect faults, violations of development standards, and other problems. Fagan is seen by many as the founder of this technique and has produced several publications on this subject, (for instance Fagan, 1986). Inspections have proved to be a very cost effective way of finding defects.

The objective of inspections is:

- to verify that the software is meeting its specification;
- to verify if the software conforms to the applicable standards;
- establish an incidental and structural improvement of the quality of products and process.

Defects found by inspection should be fixed depending on their severity, just like other defects.

Inspection can be introduced much earlier than dynamic testing. The test design can only start if the requirements are ready. These requirements can be subject to inspection to verify whether they have the desired quality for use in development and test design or not.

Preparation is the most important phase of the inspection. During this phase, the assessors evaluate the specifications. One or more roles are assigned to each assessor and therefore each is able to assess the specifications within their own remit. Each assessor will detect unique defects, which couldn't be detected by the other assessors because of their different roles.

Invoking a causal analysis meeting may increase the value of inspections. During such a meeting the causes of a detected defect are determined and this should prevent such defects in the future.

It is important that the person who organizes and steers the inspections has some degree of independence. In order to achieve a high degree of effectiveness and efficiency, this person (the moderator) should have a deep knowledge of the inspection technique.

9.1.1 Advantages

The use of inspections of products of the development has various advantages.

- Defects detected early can be corrected relatively cheaply.
- The detection rate of defects is quite high because a focused and dedicated team carries out the assessment.
- The evaluation of a product by a team encourages the exchange of information between its members.
- Inspections are not restricted to design documents but can be used for all documentation deliverables of the development process, as well as the test process.
- The inspection stimulates awareness and motivation regarding development of quality products.

Inspection is an eminently suitable technique for improving the quality of products. This applies initially to the assessed products themselves. The feedback from the inspection process to development processes enables process improvements.

9.2 Procedure

Inspections involve the following steps:

- entry check
- organizing the inspection
- kick-off
- preparation
- defect registration meeting
- causal analysis meeting
- performing rework
- follow-up
- checking on exit criteria.

9.2.1 Entry check

The moderator performs an entry check on the product, based on the entry criteria. The entry criteria give the product author clear guidelines concerning when the product is accepted for inspection. The entry criteria should avoid unnecessary starts of inappropriate products. Examples of entry criteria are:

- the product must be complete;
- the reference documentation must be approved;
- the references must be correct and up to date;
- the initial quick scan executed by the moderator detects no more than a specified number of defects;
- the document must be spell-checked;
- the document meets acceptable standards.

9.2.2 Organizing the inspection

The moderator organizes an inspection if the product passes the entry check. A team must be gathered and roles assigned to the participants – roles must be strongly matched to the interests and expertise of the participants. Some examples of roles are (Gilb and Graham, 1993):

- *User*: concentrates on the user or customer point of view;
- *Tester*: concentrates on testability;
- *System*: concentrates on wider system implications;
- *Quality*: concentrates on all aspects of quality attributes;
- *Service*: concentrates on field service, maintenance, supply, installation.

This list is not exhaustive and not every inspection should involve these roles.

All essential documents such as the product, reference documents, and standards should be distributed to the team. Dates for preparation and meetings must be set.

9.2.3 Kick-off

The kick-off meeting is optional and is usually organized for the following reasons.

- The inspection is to be carried by personnel who have no experience with this technique. The moderator can give a short introduction to the technique and describe the roles of the participants.
- If the product is complex it is very helpful if the author of the products gives an introduction.
- If the inspection procedures are changed, a kick-off meeting is useful for informing the participants.

9.2.4 Preparation

The only activity of the preparation phase is the detection of unique defects. Every participant has their own role and the product is inspected from that point of view. Although it is more likely that they detect defects to do with their expertise, everyone should register all defects detected. All related documents are subject to the inspection.

9.2.5 Defect registration meeting

In addition to logging detected defects, the defect registration meeting has the purpose of detecting new defects and exchanging knowledge.

The moderator, participants, and author participate in this meeting. The moderator is responsible for listing all defects. Meanwhile, the author registers them all in a defect report. To avoid wasting time, cosmetic defects and possible solutions are not discussed. The defect registration meeting should be limited – don't try to handle an 80-page document in a single day's meeting.

9.2.6 Causal analysis meeting

This must be held in a constructive atmosphere soon after the defect registration meeting. It is constructed in the form of a brainstorming session. The focus must be on the root causes of defects rather than on the defects themselves. Based on the root causes, minor improvement suggestions should contribute to a sustained improvement of the process.

9.2.7 Performing rework

The author must solve all registered defects in the assessed product. All other defects in related documents are registered as change proposals. The updated document is delivered to the moderator.

9.2.8 Follow-up

The moderator must check if the author has solved all the defects. The moderator does not check whether the defects are solved correctly or not. As feedback, the document can be sent to the team members who can check if the defects have been solved satisfactorily.

9.2.9 Checking on exit criteria

The final inspection step is the formal decision as to whether product meets the exit criteria. Examples of such criteria are:

- rework must be completed;
- the new document is subject to configuration management;
- the change requests for the related documents are processed according to the procedures;
- the inspection report is handed over to, and accepted by, quality management.

During the whole process of inspection, data can be collected for measurement purposes. These can be used to demonstrate the benefit of early inspection.

Safety analysis 10

10.1 Introduction

Safety is the expectation that a system does not, under defined conditions, lead to a state in which human life is endangered.

Many embedded systems are used in safety critical situations. Malfunctioning can have serious consequences such as death, serious injury, or extensive environmental damage. The design and development process for these systems must have some built-in measures that make it very unlikely for safety to be threatened.

The best way to handle the safety requirements of a system is to start monitoring it in the design phase. In this chapter, two safety analysis techniques are described FMEA (Failure Mode and Effect Analysis) and FTA (Fault Tree Analysis).

In safety analysis, severity categories are used (see Appendix A). These categories facilitate the classification of any risks identified. This classification can be used to formulate a safety strategy in which different measures are taken for different risk categories.

Safety analysis is about the cause–effect relationship (see Figure 10.1). The effect is always related to the endangerment of life.

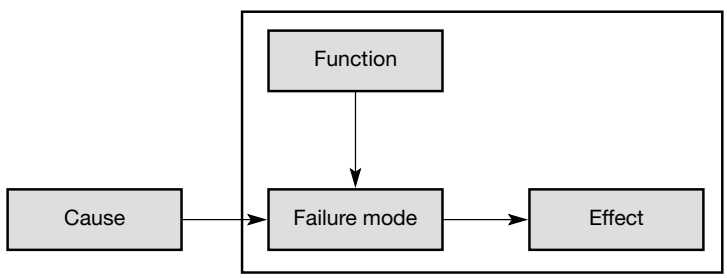


Figure 10.1
The relationship between cause, function, failure mode, and effect. The boxed section can be analyzed by FMEA, and the whole is analyzed by FTA.

The causes of failure are varied:

- fault (hardware and software) – an imperfection or deficiency in the system which may, under some operational conditions, result in a failure;
- wear and tear (hardware);
- environmental conditions such as electromagnetic interference, mechanical, and chemical disturbance (hardware).

A *failure mode* describes the way in which a product or process could fail to perform its desired function as described by the needs, wants, and expectations of internal and external customers. A *failure* is the inability of a system or component to fulfill its operational requirements. Failures may be systematic or due to physical change. An *effect* is an adverse consequence caused by a failure mode.

10.2 Safety analysis techniques

10.2.1 Overview

First the scope of the analysis must be defined. Should it be done on module, subsystem, and/or system level? To enable a more concrete discussion about "how safe" something should be, a risk classification table is constructed (see Table A.4). Second the analysis techniques FMEA (see section 10.2.2) and FTA (see section 10.2.3) can be applied. In section 10.2.4 an overview of variants of safety analysis methods is given. The analysis techniques are executed by small teams made up of a safety manager, designers, experts, and a tester. The safety manager is responsible for achieving the appropriate safety levels within the system. The tester is part of this team to bring a criticizing attitude and neutral position. Based on the results of the analysis, the safety manager might change the project plan and the designer may need to change the product design.

10.2.2 Failure mode and effect analysis

FMEA is a forward analysis method which determines the effect of a failure mode (corrupt data or unexpected behavior) on the system. In this context system means equipment as an end product (so it includes both software and hardware). The technique is used early in the design process when it is easy to take action to overcome any problems identified, thereby enhancing safety through design. FMEA comprises three steps:

- identifying potential failure modes;
- determining the effects of these potential failure modes on the functioning of the system;
- formulating actions to diminish the effects and/or failure modes.

The proper use of FMEA can have the following benefits:

- highly improved safety of the system;
- the ability to track risks throughout the development lifecycle;
- early identification of potential safety hazards;
- documented risks and actions to reduce them;
- minimizes late changes and related costs;
- produces a highly reliable input for the test strategy.

A typical FMEA is carried out as follows.

1 The system and its functions are described. The design and the requirements can also be used. If no documentation is available giving the relationships between the different parts of the system (flow diagram or block diagram), this should be prepared. Having the lead designer participate in the FMEA can compensate for the lack of documentation.
2 Potential failure modes are identified. Lutz and Woodhouse (1999) identified two categories of failure modes for software: event and data failure modes. The data failure modes are:

   - missing data (e.g. lost message, no data because of hardware failure);
   - incorrect data (e.g. inaccurate, spurious),
   - timing of data (e.g. old data, data arrives too soon for processing);
   - extra data (e.g. data redundancy, data overflow).

   The event failure modes are:

   - halt/abnormal termination (e.g. hung or deadlocked);
   - omitted event (e.g. execution continuous while event does not take place);
   - incorrect logic (e.g. preconditions are not accurate, event is not triggered as intent);
   - timing/order (e.g. event occurs at the wrong moment, or the ordering of events is wrong).

3 Every function under investigation is described in respect of what the effect is of the potential failure modes. In this stage of FMEA, there is no discussion about whether the failure mode is possible. The first step is to describe the direct effect at the function under investigation, and the next is the description of the effect at system level. The effects are classified according to their related risks (see Appendix A).

   The prediction of the system effect is based mostly on assumption, because the system is not yet fully understood or it is incomplete in design or detail. In these situations it is not desirable to discuss this effect. Staff who carry out the FMEA must be aware of this situation.
4 For the serious risks (i.e. the risks classified in the first three risk categories in Appendix A) the causes of the effects are identified, and for every cause the measures to be taken are described.

5 The risks identified are monitored during the development process and tests should be developed to give confidence in the measures adopted to handle them.

10.2.3 Fault tree analysis

FTA is used for identifying causes of failures. It is a technique for analyzing the design in respect of safety and reliability. System failure is taken to be at the top of the fault tree, and the next step is to consider which unwanted behavior (of parts of the system) is responsible for the malfunctioning of the system. In this context, a system failure is an incorrect event, incorrect data, or unexpected data or behavior. Section 10.2.4 will discuss a couple of variants of this analysis technique.

Figure 10.2 shows the symbols used to build a fault tree. As stated earlier, the failure of the system is placed at the top of the fault tree. The next step is to determine what caused this failure. Every subsequent step looks at the cause of failure in the previous step. This analysis leads to a reason for the system failure. Safety requirements are the baseline for deciding what is unexpected or unwanted system behavior. FTA is a job for an expert team.

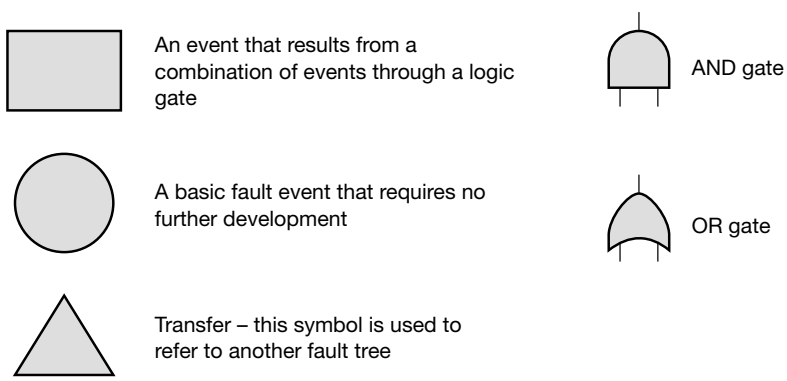


Figure 10.2
FTA notation

Figure 10.3 depicts an FTA showing the possible causes of a pacemaker failure. At the top is the unsafe condition, pace too slow. This is a potential life threatening malfunction. It can result from a number of faults. In this case, a time-base fault or a shutdown failure causes the pacemaker to run too slowly. A watchdog failure and a crystal failure cause the time-base fault. By determining for every event the conditions causing that event, a fault tree is created. If an event can be caused by only two or more conditions occurring together, these conditions are connected by an AND gate. If either of these conditions can cause an event, then they are connected by an OR gate.

A system in which the conditions of a hazard are connected by only OR gates is very sensitive to disturbance. Just one failure in the system leads to a cascade of events and an unsafe situation – such failures are called single-point failures.

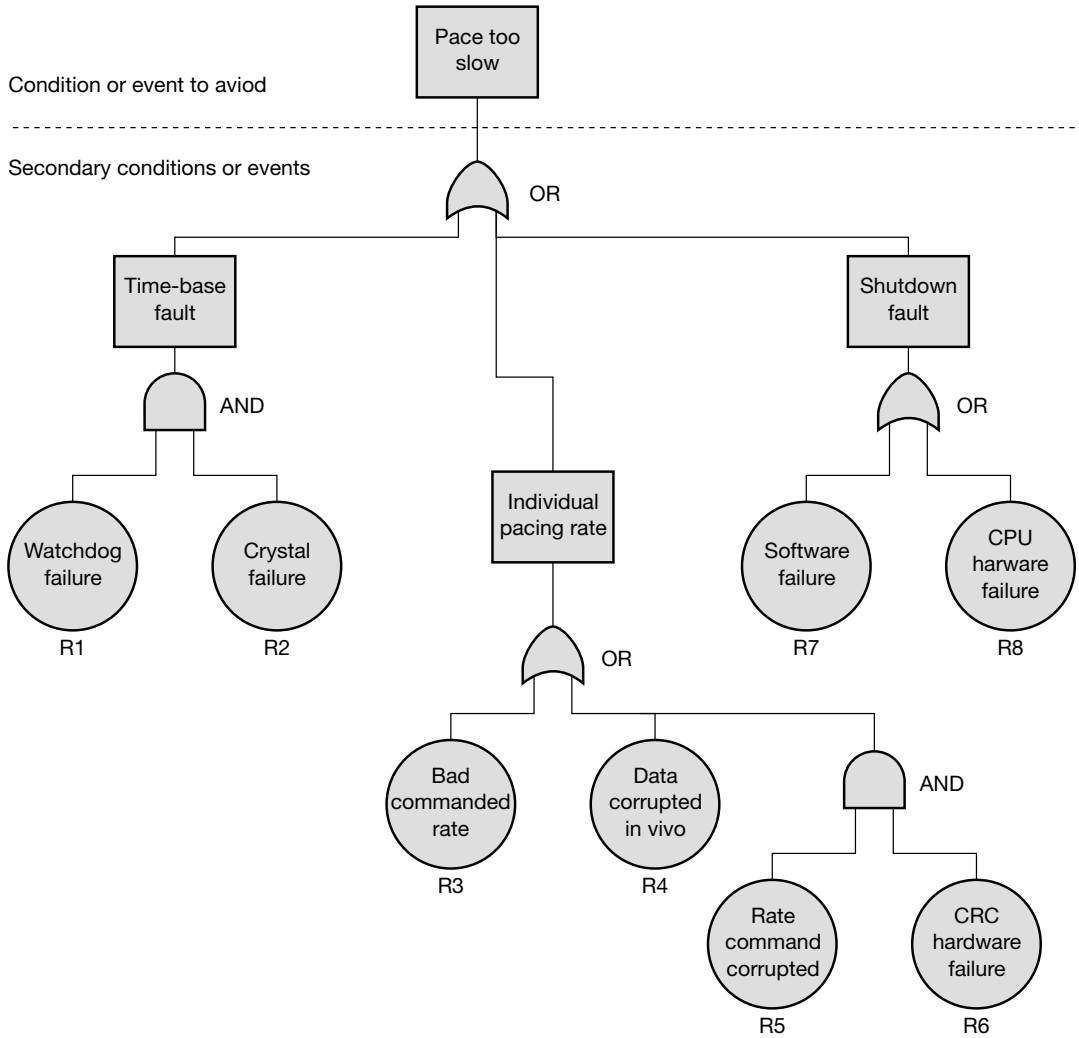


Figure 10.3 Subset of a pacemaker FTA (Douglass, 1999)

A fault tree can also be used to find common mode failures. These failures affect many parts of the system. In a fault tree, these failures are the fundamental cause of several conditions.

It is not always necessary or desirable to have a complete breakdown structure for the top hazard. For example R1 (watchdog failure) in Figure 10.3 can be used as a top hazard in another fault tree. By using the fault tree in this manner, this technique can be used on complex systems and at both high- and low-level system breakdowns.

10.2.4 Variants on the techniques

The FMEA technique takes one failure mode at a time – there are similar techniques that take more than one failure mode at a time into account.

The FTA technique is also used in a slightly different way to determine the relevance of the outcome of the FMEA. FMEA describes the effect at system level. If the effect is classified in one of the three highest risk classes (as an example), then safety measures must be taken. Before starting out with this, it is wise to determine whether or not it is possible for the effect to occur. If it is possible then the effect of this failure mode is analyzed with FTA. If it turns out to be possible for the effect to occur, then measures must be taken.

Another variant is the determination of the probability of occurrence of undesired behavior with FTA. All possible causes of a top event, and their probability of occurrence, are determined. Using these probabilities, is it possible to calculate the probability of occurrence of the top event. The probabilities at OR gates are added (be aware that this is not simply the same as adding two numbers together) and the probabilities at AND gates are multiplied. Calculating the probabilities in the fault tree bottom-up, it is possible to get the probability of occurrence of the top event.

A third variant of FTA classifies every event in a risk class. The risk class defines the development method, the way it is built and monitored. The top event of Figure 10.3 is a life threatening situation. This event is classified in the highest risk class. In the fault tree, every event related to an OR gate is classified in the same class as the event at the next higher level. In the case of an AND gate there are three possibilities.

- The event is connected to a monitoring function; the function watches over another function – then the event is classified in the same risk class as the next higher level.
- The event is connected to a function that is monitored by another function; then the function is classified in a lower risk class than the next higher level.
- The event is not connected to a monitoring function, so it will be classified in the same risk class as the next higher level.

10.2.5 Other applications of the techniques

FMEA and FTA are techniques normally only presented and used for safety analysis. However, they are also very useful during the test strategy determination. FMEA and FTA can then be effectively applied to identify the weak spots and risky parts of the system (see section 7.2). Usually the "determine the relative importance of subsystems" step is not too difficult, even without techniques such as FMEA and FTA. However, deciding on a more detailed level just which functions or units should be tested more thoroughly is often more an educated guess than a well-founded decision. This can be much improved by applying FMEA and FTA.

10.3 Safety analysis lifecycle

10.3.1 Introduction

Building a safety critical system involves dealing with the law and certification authorities. Some of the regulations and/or requirements related to safety critical systems are very strict. In order to fulfill these requirements there has to be a well-structured process with clear deliverables. With safety in mind, the British Ministry of Defence has developed a standard for safety management called MOD-00-56 (MOD, 1996a, 1996b). Part of this standard is a description of a structured process for developing and implementing a safety critical system. The process shares some products and activities with the test process. Also included in the safety process are a couple of test activities. If the shared activities and products are not well co-ordinated, then both processes can frustrate each other. The result is a product that is not certifiable or one that offers incorrect functionality.

The safety analysis lifecycle (see Figure 10.4) is a structured description of an activity diagram and data flow. The objective of this process is to move from some global requirements to a system certified for safety related use. All the designs, decisions, and results of analyses are stored centrally in a hazard log. This forms the basis for the safety case and this case is delivered specially for certification.

- *Safety requirements*. These requirements are part of the test base for safety verification. The establishment of the safety requirements is the first activity of the safety process. These requirements are a translation of law and other restrictions and will not change during the rest of the process.
- *Safety project program*. This is the project statement with time, cost, and resource estimations.
- *Hazard identification*. This activity has the objective of identifying the parts of the system in which malfunctioning can have serious consequences for the working of the system. FTA and FMEA are the techniques most often used.
- *Risk estimation*. All identified hazards are analyzed to see what the effect on the system can be and what the consequences would be. Based on this classification, it is estimated what the risk is and whether or not corrective action has to be taken.
- *Safety compliance assessment*. This assessment has the objective of finding whether or not all the necessary measures are taken, and taken in the right way.
- *Safety verification*. The system is tested to see whether or not it functions according to the safety requirements.

A detailed description of this process and all the activities can be found in MOD 00-56 (Parts 1 and 2) (MOD, 1996a, 1996b).

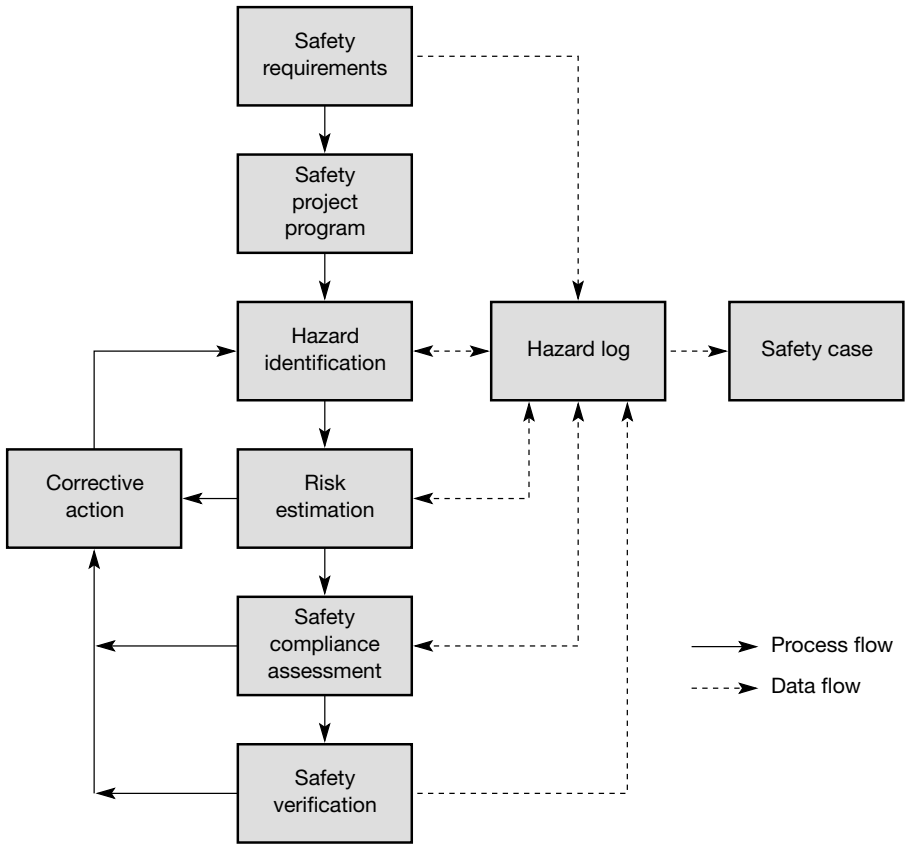


Figure 10.4
Safety lifecycle based on MOD-00-56 (MOD, 1996a, 1996b)

10.3.2 Test base

The final design is part of the test base for the "regular" test and also for the safety tests. The final design is realized during an iterative process (see Figure 10.5).
The process starts with a global design. Based on this, hazard identification and risk estimation is executed. The outcome of the risk estimation can be that corrective actions are necessary – otherwise the global design is worked out in more detail and the hazard identification and risk estimation are executed again. This process goes on until no more corrective actions are needed and the detailed design is ready.

This final design, together with the requirements (functional, performance, usability, etc.), is the test base for the regular test. The delivery of the final design means that the test group can start with the preparation and execution phases of the test process. The final design is judged on its testability during the preparation phase. The process will be smooth if this activity is included in the process of realization of the design.

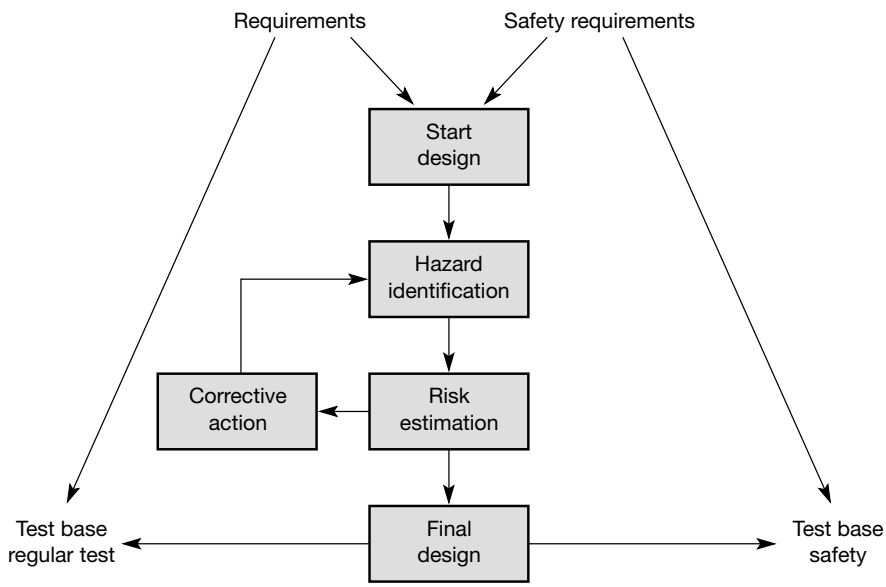


Figure 10.5
The realization of the final design and its relation to the test and safety process

The final design and the safety requirements are the test base for the safety compliance assessment and the safety evaluation. These activities can be prepared by the same test group who do the regular test or by a separate group. The test plan is sometimes subject to audit and the safety manager should always accept the test plan.

10.3.3 Test activities

Test execution in the regular test process and the safety project will produce defects. These should be analyzed and sometimes corrected. The correction of a safety defect can influence the functioning of the system and vice versa, a correction of a functional defect can influence safety. In order to overcome this problem, the impact analysis and corrective actions should be centralized (see Figure 10.6).

A corrective action for a safety defect can lead to a functional retest. A corrective action related to a defect related to a quality attribute will, more often than not, lead to a safety verification.

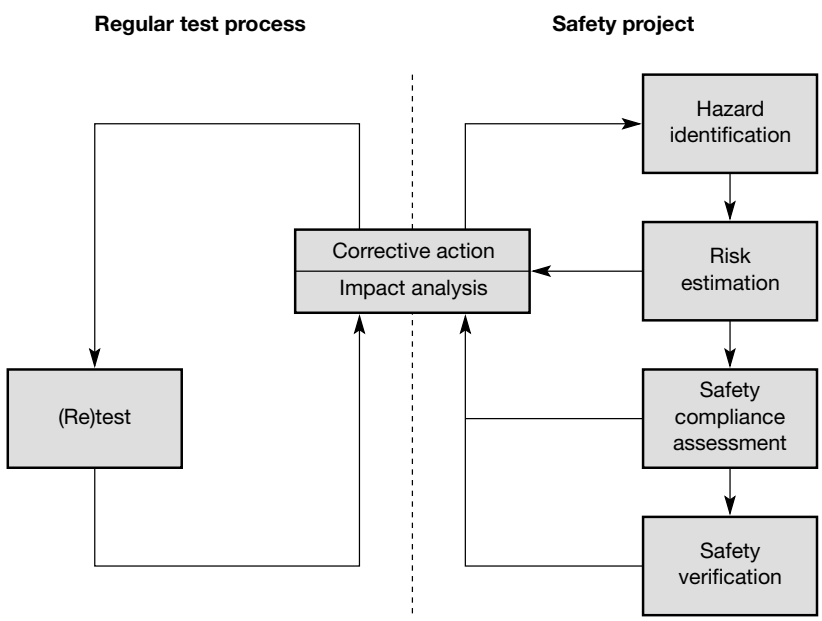


Figure 10.6
Centralized impact analysis and corrective action

Test design techniques 11

11.1 Overview

11.1.1 Introduction

Structured testing implies that it has been well thought through just which test cases are needed to accomplish the required goals, and that those test cases are prepared before actual test execution. This is in contrast to just "making up smart test cases as you go along." In other words, structured testing implies that test design techniques are applied. A test design technique is a standardized method of deriving test cases from reference information.

This chapter first explains the general principles of test design techniques – the steps involved in using one and the resulting test products are described. This is followed by a brief discussion of the advantages of this structured way of deriving test cases.

An abundance of test design techniques is available, published in books and magazines, and presented at conferences. This chapter discusses some characteristics of such techniques to provide insight into how one technique differs from another. This helps the selection of the appropriate test design technique when determining the test strategy.

Finally, several different test design techniques are described in detail. A selection of techniques is presented that will provide a useful basis for a tester. However, the reader is encouraged to study other literature as well – learning about more and newer techniques is part of improving testing skills.

11.1.2 Generic description of the steps

For the majority of test design techniques the structured test design process follows a generic plan consisting of discrete steps. A description of the steps is given below.

Identifying test situations

Each test design technique aims to find certain types of defects and achieves a certain type of coverage. This has to be related to information in the test basis, which may consist of the functional requirements, state transition diagrams, the user guide, or any other documentation specifying the system behavior. The first step

in the test design process is to analyze the test basis and identify each situation to be tested. The test design technique may state, for example, that the situations to be tested are all those that are formed by having all conditions both `true` and `false`, or that each input parameter is tested with valid and invalid values.

Specifying logical test cases

A series of test situations is transformed into a logical test case which passes through the test object from start to finish. It is possible that the logical test cases are the same as the test situations, but in general they contain several test situations. The test cases are described as "logical" because they describe the "kind of situation" that will be covered without giving concrete values for the parameters involved. The logical test cases serve to prove that the intended test goals and coverage will be achieved.

Specifying physical test cases

The next step is the creation of the physical test cases. For each logical test case concrete input values are chosen that correctly represent that particular logical test case and the resulting output is defined in concrete terms. Choosing concrete input values is not always completely unrestricted but can be limited by agreements on, for example, the use of boundary values or the use of certain initial data sets. In general, a physical test case provides all the information needed to execute that test case including input values, test actions to be executed, and expected results.

Establishing the initial situation

In order to execute physical test cases, the required initial situation must be present. This may mean that a certain data set must be installed or that the system must be put into a certain state. Often, several physical test cases can start from the same initial situation. Then it is useful to prepare initial data sets and procedures to put the system into a certain state, which can be reused for many test cases.

Assembling the test scripts

The final step is to define the test script, which places the test actions and result checks of the individual test cases in the optimal sequence for test execution. The test script serves as a step-by-step guide for the tester on how to execute the test. The physical test cases in conjunction with the prepared initial situations form the basis for the test scripts.

Defining test scenarios

This is an optional step which is advisable in complex situations where several test scripts have dependencies on other test scripts (for instance, script B can only be executed after succesful completion of script A). A test scenario can be seen as a kind of "micro test-planning" describing the order in which the test scripts should be executed, which preparatory actions are required, and possible alternatives in case things go wrong.

11.1.3 Advantages

Applying defined test design techniques and documenting the results has many advantages for the testing process. In short, it increases the quality and control of the testing process.

There are various arguments which illustrate this point.

- Using test design techniques provides insight into the quality and coverage of the tests, based on solid implementation of the test strategy, which gives the correct coverage in the correct place.
- As a test design technique aims to find certain types of defects (for example, in the interfaces, in the input validations, or in the processing), such defects will be detected more effectively than in the case of randomly specified test cases.
- Tests can be easily reproduced because the order and content of the test execution has been specified in detail.
- The standardized working procedure makes the test process independent of the person who specifies and carries out the test cases.
- The standardized working procedure makes the test designs transferable and maintainable.
- The test process can be planned and controlled more easily because the test specification and execution processes have been divided into well-defined blocks.

11.1.4 Characteristics

To assist in the selection of test design techniques and to be able to compare them, the following general characteristics of test design techniques are discussed:

- black-box or white-box;
- principles of test case derivation;
- formal or informal;
- application areas;
- quality characteristic to be tested;
- required type of test basis.

11.1.4.1 Black-box or white-box

Black-box test design techniques are based on the functional behavior of systems, without explicit knowledge of the implementation details. In black-box testing, the system is subjected to input and the resulting output is analyzed as to whether it conforms to the expected system behavior. White-box test design techniques are based on knowledge of a system's internal structure. Usually it is based on code, program descriptions, and technical design.

However, the boundary between black-box and white-box characteristics is fuzzy because what is white-box at the system level translates to black-box at the subsystem level.

11.1.4.2 Principles of test case derivation

There are various principles that test design techniques use to derive test cases. Some of these are described below.

Processing logic

An important principle is to base test cases on detailed knowledge of the logic of the processing of the program, function, or system to be tested. Here, processing is seen as a composite set of decision points and actions. A decision point consists of one or more conditions and basically states that "`IF` this condition holds true `THEN` continue with action 1 `ELSE` continue with action 2." The various combinations of actions and decisions that may be taken consecutively are referred to as paths.

The following shows an example of the description of processing logic. It describes when an elevator must move up, move down, or stop. This depends on its current position, the floor selected by people inside the elevator, and the floor that people (outside the elevator) want the elevator to move to.

Table 11.1 Logic control of an elevator

Specification	Explanation
`IF Floor_Selection = ON`	Decision B1, consisting of conditions C1
`THEN IF Floor_Intended > Floor_Current`	Decision B2, consisting of conditions C2
`THEN elevator moves up`	Action A1
`ELSE elevator moves down`	Action A2
`IF Floor_Intended = Floor_Current OR Floor_LiftRequested = Floor_Current`	Decision B3, consisting of condition C3 and condition C4
`THEN elevator stops`	Action A3

In the situation described in Table 11.1, when a person in the elevator on the ground floor pushes the button for third floor, and another person on the second floor pushes the elevator request button, the ensuing actions will follow the path B1/B2/A1/B3/A3. This results in the elevator moving up to the second floor and then stopping.

Test design techniques can focus on covering paths or on the possible variations at each decision point. They constitute different kinds of coverage. Well-known types of coverage related to processing logic are listed in Table 11.2.

Table 11.2
Types of coverage related to processing logic

Statement coverage	Each action (=statement) is executed at least once
Branch coverage or decision coverage	Each action is executed at least once and every possible result (`true` or `false`) of a decision is completed at least once – this implies statement coverage
(Branch) condition coverage	Each action is executed at least once and every possible result of a condition is completed at least once – this implies statement coverage
Decision/condition coverage	Each action is executed at least once and every possible result of a condition and of a decision is completed at least once – this implies both branch condition coverage and decision coverage
Modified condition decision coverage	Each action is executed at least once and every possible result of a condition independently affects the outcome of the decision at least once (for a more detailed explanation see section 11.4) – the modified condition decision coverage implies decision/condition coverage
Branch condition combination coverage	All possible combinations of results of conditions in a decision are completed at least once – this implies modified condition decision coverage
Path^n coverage, with n=1, 2,… also referred to as test depth level n	All the aforementioned types of coverage refer to discrete actions and decisions. In the case of path coverage, the focus is on the number of possible paths. The concept of "test depth level" is used to determine to what extent the dependencies between consecutive decisions are tested. For test depth level n, all combinations of n consecutive decisions are included in test paths The selection of a particular test depth level has a direct effect on the number of test cases (and therefore the test effort) on the one hand, and the degree of coverage of the tests on the other. Using the test depth level parameter, we can choose the number of test cases that corresponds to the defined test strategy For an example of this complex definition of the concept of test depth level, see section 11.3

Other terms used for this way of deriving test cases are logic testing (Myers, 1979), control flow testing, path testing, and transaction flow testing (Beizer, 1995).

The processing logic may be considered at various levels. In low-level tests, the focus is on the internal structure of programs. The statements in the program then constitute the decisions and actions. For high-level tests, the functional requirements can be viewed as the processing logic.

Equivalence partitioning

Using this principle of deriving test cases, the input domain (all possible input values) is partitioned into "equivalence classes." This means that for all input values in a particular equivalence class, the system shows the same kind of behavior (performs the same processing). Another term used for deriving test cases by means of equivalence partitioning is domain testing (Beizer, 1990, 1995).

A distinction can be made between valid and invalid equivalence classes. Input values from an invalid equivalence class result in some kind of exception handling, such as generating an error message. Input values from a valid equivalence class should be processed correctly.

The idea behind this principle is that all inputs from the same equivalence class have an equal chance of finding a defect, and that testing with more inputs from the same class hardly increases the chance of finding defects. Instead of testing every possible input value, it is sufficient to choose one input from each equivalence class. This greatly reduces the number of test cases, while still achieving a good coverage.

This is illustrated in the following example, where the system behavior is subjected to the following condition regarding the input temperature:

$$15 =< \text{temperature} =< 40$$

The number of possible values for the temperature is huge (in fact it is infinite). However, this input domain can be partitioned into three equivalence classes:

- temperature is lower than 15;
- temperature has a value in the range 15 through 40;
- temperature is higher than 40.

Three test cases are sufficient to cover the equivalence classes. For example, the following values of temperature could be chosen: 10 (invalid), 35 (valid), and 70 (invalid).

Equivalence partitioning can also be applied to the output domain of the system. Test cases are then derived that cover all equivalent output classes.

Boundary value analysis

An important specialization of the above principle is the boundary value analysis. The values that separate the equivalence classes are referred to as boundary values.

The idea behind this principle is, that defects can be caused by "simple" programming errors related to erroneous use of boundaries. Typically the programmer has coded "less than" when "less than or equal" should have been coded. When determining the test cases, values around these boundaries are chosen so that each boundary is tested with a minimum of two test cases – one in which the input value is equal to the boundary, and one that is just beyond it.

Using the example above (15 =< temperature =< 40) and assuming a tolerance of 0.1 in the temperature values, the boundary values to be selected are 14.9 (invalid), 15 (valid), 40 (valid) and 40.1 (invalid).

A more thorough use of boundary value analysis is that three values are chosen instead of two at each boundary. The additional value is chosen just within the equivalence partition defined by the boundary value. In the above example, the two extra values 15.1 (valid) and 39.9 (valid) must then be tested. If the programmer incorrectly coded "15 == temperature" (instead of "15 =< temperature"), this would not be detected with just two boundary values, but would be detected with the additional value 15.1.

Boundary value analysis, and equivalence partitioning, can be applied not only to the input domain but also to the output domain. Suppose a message on an LCD screen is allowed to contain up to five lines. Then this is tested by sending one message with five lines (all lines on one screen) and one with six lines (the sixth line on the second screen).

Applying boundary value analysis generates more test cases, but it increases the chances of finding defects compared to a random selection from within the equivalence classes.

Operational usage

Test cases can also be based on the expected usage of the system in the field. The test cases are then designed to simulate the real-life usage. This means, for example, that functions frequently used in practice will have a proportionate number of test cases specified, regardless of the complexity or the importance of the function. Testing on the basis of operational usage often results in a large number of test cases, all belonging to the same equivalence class. Therefore it offers little chance of finding new functional defects, but is more suited to analyzing performance and reliability issues.

CRUD

Sometimes an important part of system behavior is centered on the lifecycle of data (create, read, update, and delete). Data emerge, are retrieved and changed, and eventually removed again. Test cases based on this principle investigate whether the functions interact correctly through the data entities, and whether

the referential relation checks (consistency checks of the data model) are complied with. This provides insight into the (completeness of the) lifecycle of the data or entities.

Cause–effect graphing

Cause–effect graphing is a technique for transforming natural language specifications into a more structured and formal specification. It is particularly suitable for describing the effect of combinations of input circumstances. A *cause* is an input condition, such as a certain event or an equivalence class of an input domain. An *effect* is an output condition that describes the system's reaction. Now, an output condition can itself be the cause of another effect. It is not unusual that a cause–effect graph shows a chain of cause – effect/cause – effect/cause – effect – … .

Using standard Boolean operators (`OR`, `AND`, `NOT`), a graph can be produced that describes a system's behavior in terms of combinations of input conditions and the resulting output conditions. A more detailed explanation can be found in Myers (1979) and BS7925-2.

11.1.4.3 Formal or informal

A formal test design technique has strict rules on how the test cases must be derived. The advantage is that it minimizes the risk that the tester will forget something important. Different testers using the same formal test design technique should produce the same logical test cases. The disadvantage is that the test cases can only be as good as the specification they are based on. (A poor system specification will lead to a poor test.)

An informal test design technique gives general rules and leaves more freedom to testers. This places more emphasis on the tester's creativity and "gut feeling" about possible weaknesses of the system. It usually requires specific domain expertise. Informal test design techniques are less dependent on the quality of the test basis but have the disadvantage that they provide little insight into the degree of coverage in relation to the test basis.

11.1.4.4 Application areas

Some test design techniques are particularly suited to testing the detailed processing within one component, while others are more suited to testing the integration between functions and/or data. Yet another group is meant to test the interaction between systems and the outside world (users or other systems). The suitability of the various techniques is related to the type of defects that can be found with their aid, such as incorrect input validations, incorrect processing or integration defects.

11.1.4.5 Quality characteristic to be tested

A set of test cases that sufficiently cover the functionality to be tested may be inadequate to test the performance of the system or its reliability. In general the choice of a certain test design technique depends a great deal on the quality characteristic to be tested.

11.1.4.6 Required type of test basis

Because a test design technique is, by definition, a standard way of deriving test cases from a test basis, it requires that specific type of test basis. For example, the use of the "state transition testing" technique requires that a state-based model of the system is available. If a required test basis is not available, the organization may decide that the test team itself should produce such documentation – and of course this has obvious risks.

In general, formal test design techniques are more dependent on the availability of a specific test basis than informal test design techniques.

11.2 State transition testing

Many embedded systems, or parts of embedded systems, show state-based behavior. In designing these systems, state-based modeling is used. Models composed during this process serve as a basis for test design. This section describes a technique for deriving test cases from state-based models.

The purpose of the state-based test design technique is to verify the relationships between events, actions, activities, states, and state transitions. By using this technique, one can conclude if a system's state-based behavior meets the specifications set for this system. The behavior of a system can be classified into the following three types.

- *Simple behavior*. The system always responds in the exact same way to a certain input, independent of the system's history.
- *Continuous behavior*. The current state of the system depends on its history in such a way that it is not possible to identify a separate state.
- *State-based behavior*. The current state of the system is dependent on the system's history and can clearly be distinguished from other system states.

State-based behavior can be represented by using tables, activity charts, or statecharts of these, statecharts are the most common method of modeling state-based behavior and are often described using UML (the Unified Modeling Language).

Appendix B explains, by means of the UML-standard (OMG, 1997, 1999) what elements can be seen in a statechart and what purpose each element serves. Section 11.2.1 describes a list of fault categories related to statecharts and which fault categories can be covered by the presented test design technique. Section 11.2.2 will deal with how statecharts are used as basis for test design.

Section 11.2.3 will describe what is meant by coverage. Furthermore, this section will examine the defect detection capability of this test design technique and will present a number of possibilities to simplify the practicality of the test cases.

11.2.1 Fault categories

Incorrect state-based behavior can have three causes. The first is that the statechart(s) do(es) not represent a correct translation of the system's functional specifications. The state-based test design technique is not capable of revealing these types of faults because the statecharts themselves are used as the basis of the tests.

A second cause is that the statecharts are syntactically incorrect or inconsistent. These faults can be revealed by static testing (for example, by using a checklist or tools). If the faults found this way are corrected, then the statechart constitutes the basis for dynamic testing using the state-based test design technique.

The third cause is the translation from statecharts to code. It is becoming increasingly common that this translation is performed automatically. The code generated this way is (hopefully) an exact representation of the statecharts' behavior. The use of the statecharts as the basis for test design in this case is, therefore, not useful and the application of the state-based test design technique is superfluous. However, if coding based on statecharts is applied without the use of a generator, then the state-based test design technique should be used.

The following faults can occur in statecharts and software.

1 **States**
   - 1.1 States without incoming transitions (specification and/or implementation fault).
   - 1.2 Missing initial states – all paths in a statechart must be defined. In the case of a transition to a superstate in which the resulting substate is not indicated, the superstate must contain an initial state (see Appendix B.6 – these are the states indicated by a dot-arrow as startpoint). If the initial state is not indicated, the state in which the transition terminates can not be predicted (specification and/or implementation fault).
   - 1.3 An additional state – the system turns out to have more states than is represented in the statechart (implementation fault).
   - 1.4 A missing state – a state that is shown on the statechart but is not present in the system (implementation fault).
   - 1.5 A corrupt state – a transition takes place to a non-valid state, leading to a system crash (implementation fault).

2 **Guards**
   - 2.1 A guard must point to a transition and not to a state (specification fault).
   - 2.2 Guards on completion-event transitions – if the guard is evaluated as `false` then the system can get into a deadlock (specification and/or implementation fault).

2.3 Guards on initial transitions – an initial transition cannot have a guard. What happens if this guard should be evaluated as `false` (specification and/or implementation fault).
2.4 Overlapping guards – in this situation it is undefined as to which state the system changes (specification and/or implementation fault).
2.5 Guard is `false` but a transition still takes place – the system reaches a non-intended resulting state (implementation fault).
2.6 Incorrect implementation of the guard – this leads, under certain conditions, to non-intended behavior (implementation fault).

3 **Transitions**
3.1 Transitions must have an accepting and a resultant state (specification and/or implementation fault).
3.2 Conflicting transitions – an event triggers a change from one substate to another, and at the same time triggers a transition out of the superstate. This results in the fact that a certain substate is no longer within reach (specification and/or implementation fault).
3.3 Missing or incorrect transitions – the resulting state is neither correct nor corrupt (specification and/or implementation fault).
3.4 Missing or incorrect actions – incorrect actions are performed as a result of executing a transition (specification and/or implementation fault).

4 **Events**
4.1 Missing event – an event is ignored (specification and/or implementation fault).
4.2 Hidden paths – a reaction to a defined event takes place, although this reaction is not defined in the statechart (also known as "sneak path") (implementation fault).
4.3 A reaction takes place to an undefined event (also known as a "trap door") (implementation fault).

5 **Miscellaneous**
5.1 The use of synchronization in an orthogonal region – a synchronization substate is situated in one orthogonal region without a connection to an event in another orthogonal region. A wrong implementation of the synchronization pseudostate can lead to rarely or not at all reachable states, no synchronization at all, or to a barely detectable deviation from the required system behavior.

Most modeling tools are capable of performing a syntax check so that syntactical errors can be avoided in a model.

The specification faults can be covered by a checklist (sometimes even automatically using tools) while the implementation faults should be covered by the state transition technique (although some fault categories are not covered by the state transition technique). This is shown in Table 11.3

Table 11.3 Coverage of the fault categories

Fault category	Checklist	State transition technique
1.1	✓	✓
1.2	✓	✓
1.3		✓
1.4		✓
1.5		✓
2.1	✓	
2.2	✓	✓
2.3	✓	✓
2.4[1]	✓	✓
2.5		✓
2.6[2]		
3.1	✓	✓
3.2		✓
3.3		✓
3.4		✓
4.1		✓
4.2		✓
4.3[2]		
5.1	✓	

[1] This fault category can be tested by both a checklist and the test design technique. The checklist suffices as long as code is generated from the model. However, if programming is based on the model then this fault category should also be included using the test design technique.

[2] Category 4.3 is not covered by the test design technique. Error guessing is, in this case, the only available option. Category 2.6 is very difficult to detect with a test design technique – statistical usage testing is probably the only technique which has a chance of detecting this type of fault.

11.2.2 State transition test technique

11.2.2.1 General

When testing state-based behavior, the system must be verified so that in response to input events the correct actions are taken and the system reaches the correct state.

Several state-based testing techniques have been developed (Beizer, 1990, 1995; IPL, 1996; Binder, 2000). The technique described here is called the state transition test technique (STT) and is a combination and elaboration of these techniques. The technique applies test depth level 1 and uses flat state diagrams. Section 11.2.3 describes how the coverage can be varied and how to cope with hierarchical statecharts.

STT consists of the following steps:

1 composing the state–event table;
2 composing the transition tree;
3 composing the test script legal test cases;
4 composing the test script illegal test cases;
5 composing the test script guards.

11.2.2.2 Composing the state–event table

The statechart is the starting point for the composition of the state–event table – this plots the states against the events. If a state–event combination is legal, then the resulting state of this combination will be incorporated in the table. In addition, the transition will be assigned a number. The first column of the table holds the initial state. The other columns consist of states that can be reached directly from the initial state (one transition away from the initial state). Next come the states that are two states away from the initial state, and so on until all states are represented in the table. State–event combinations for transition-to-self (resulting state is accepting state) must also be included in this the state–event table.

Using the statechart concerning a video cassette recorder (VCR) shown in Figure 11.1, the state–event table shown in Table 11.4 is composed.

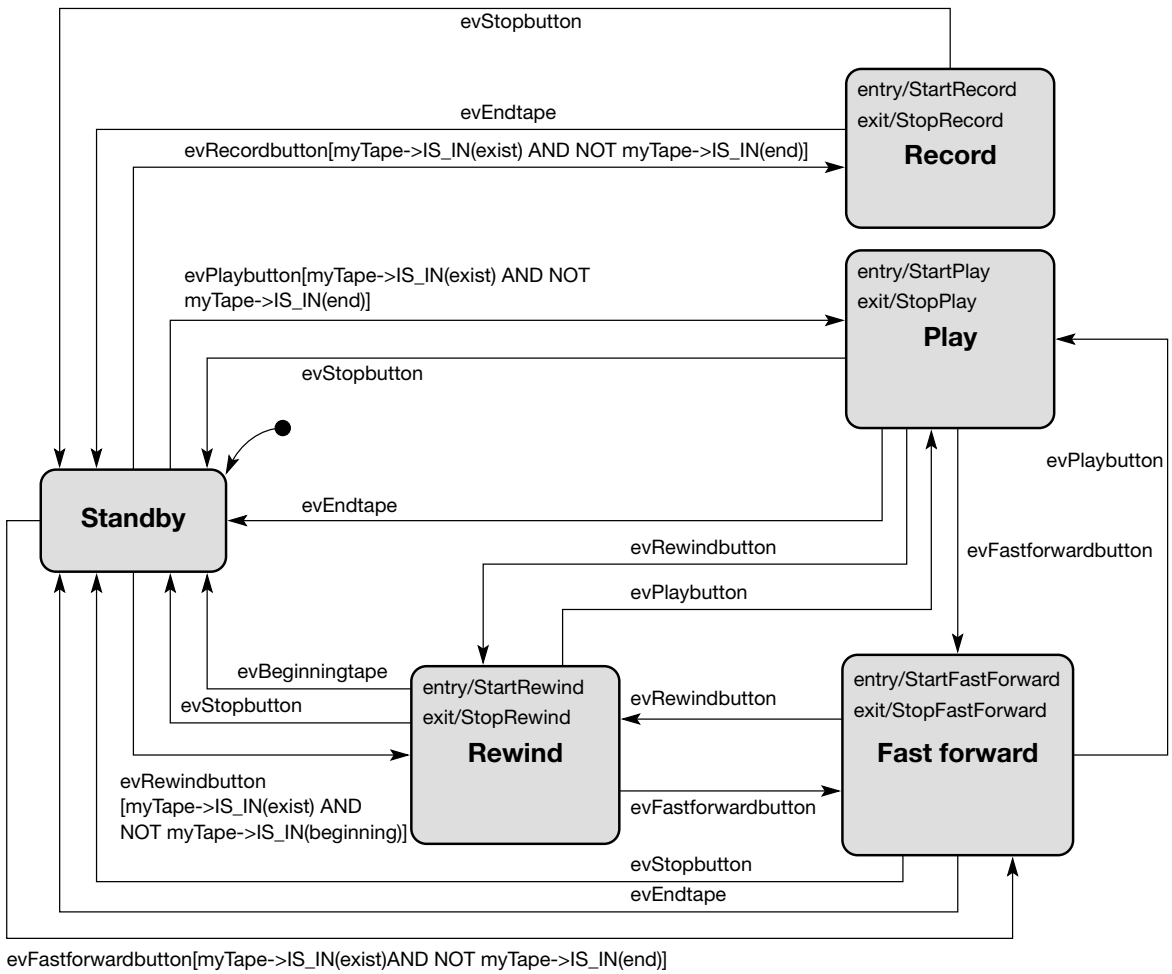


Figure 11.1 A simplified statechart of a VCR

Table 11.4
State–event table with illegal combinations indicated by a bullet

	Standby	Rewind	Play	Fast forward	Record
evRewindbutton	1-Rewind	●	9-Rewind	13-Rewind	●
evPlaybutton	2-Play	5-Play	●	14-Play	●
evFastforwardbutton	3-Fast forward	6-Fast forward	10-Fast forward	●	●
evRecord	4-Record	●	●	●	●
evStopbutton	●	7-Standby	11-Standby	15-Standby	17-Standby
evEndtape	●	●	12-Standby	16-Standby	18-Standby
evBeginningtape	●	8-Standby	●	●	●

● Illegal combinations (sneak path)

If, by means of a guard, a combination has several possible resulting states, all those resulting states are included in the table. These transitions are all given their own number.

The aim of the state-based test design technique is to cover all state–event combinations (both legal and illegal).

11.2.2.3 Composing the transition tree

The state–event table is used in composing the transition tree. The initial state forms the root of the transition tree. From this initial state, all outgoing transitions and related states are added to the tree. The numbering used in the state–event table is copied and the guarded transitions are marked. From these states, the next level transitions and states are placed in the transition tree. This is repeated until all paths reach the final state, or the initial state is reached again. If during the construction of the tree, a state is added that already appears elsewhere in the tree, then the path ends and is marked as a provisional terminal point.

If a transition-to-self is part of the statechart, the resulting state (= accepting state) is also placed in the transition tree. This is also the way a construction such as in Figure B.3 in Appendix B is treated. The test case then becomes a set of cascading event–transition pairs, finally resulting in an event–transition pair with a resulting state different from the accepting state.

In the test script that is derived from the transition tree, each path is completed along the shortest possible route. The guards are incorporated in the transition tree description. The result of this step for the VCR statechart is shown in Figure 11.2.

```
1   [myTape->IS_IN(exist)] AND NOT myTape->IS_IN(beginning)
2   [myTape->IS_IN(exist) AND NOT myTape->IS_IN(end)]
3   [myTape->IS_IN(exist) AND NOT myTape->IS_IN(end)]
4   [myTape->IS_IN(exist) AND NOT myTape->IS_IN(end)]
```

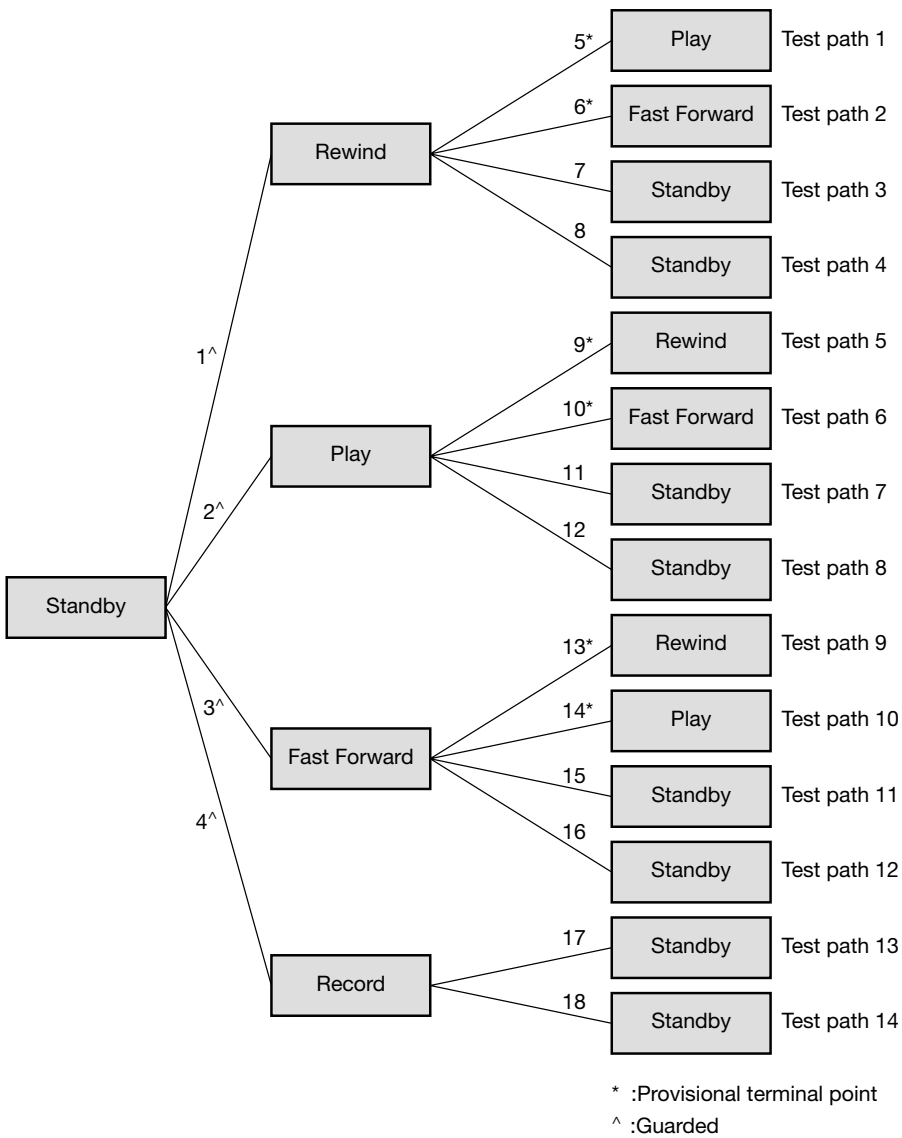


Figure 11.2
The transition tree of a VCR

Together with the state–event table, the transition tree now forms the complete description of the statechart.

11.2.2.4 Composing the test script legal test cases

With the aid of the transition tree and the state–event table, a test script can be created that covers only the legal test cases. Each path in the transition tree is a test case. This test case covers the entire path. Each line holds the event, the guard, the expected action(s), and the resulting state (see Table 11.5).

Table 11.5
The test script derived from the VCR statechart

Input			Expected result	
ID	**Event**	**Guard**	**Action**	**State**
L1.1	evRewindbutton	Tape exists, tape not at the beginning	StartRewind	Rewind
L1.2	evPlaybutton		StopRewind; StartPlay	Play
L1.3	evStopbutton		StopPlay	Standby
L2.1	EvRewindbutton	Tape exists, tape not at the beginning	StartRewind	Rewind
L2.2	evFastforwardbutton		StopRewind; StartFastForward	Fast forward
L2.3	evStopbutton		StopFastForward	Standby
L3.1	EvRewind	Tape exists, tape not at the beginning	StartRewind	Rewind
L3.2	EvStopbutton		StopRewind	Standby
L4.1	EvRewind	Tape exists, tape not at the beginning	StartRewind	Rewind
L4.2	EvBeginningtape		StopRewind	Standby
L5.1	EvPlay	Tape exists, tape not at the end	StartPlay	Play
L5.2	evRewind		StopPlay; StartRewind	Rewind
L5.3	evStopbutton		StopRewind	Standby
L6.1	evPlaybutton	Tape exists, tape not at the end	StartPlay	Play
L6.2	evFastforwardbutton		StopPlay; StartFastForward	Fast forward
L6.3	evStopbutton		StopFastForward	Standby
L7.1	evPlaybutton	Tape exists, tape not at the end	StartPlay	Play
L7.2	evStopbutton		StopPlay	Standby
L8.1	evPlaybutton	Tape exists, tape not at the end	StartPlay	Play
L8.2	evEndtape		StopPlay	Standby
L9.1	evFastforwardbutton	Tape exists, tape not at the end	StartFastForward	Fast forward
L9.2	evRewind		StopFastForward; StartRewind	Rewind
L9.3	evStopbutton		StopRewind	Standby
L10.1	evFastforwardbutton	Tape exists, tape not at the end	StartFastForward	Fast forward
L10.2	evPlaybutton		StopFastForward; StartPlay	Play
L10.3	evStopbutton		StopPlay	Standby
L11.1	evFastforwardbutton	Tape exists, tape not at the end	StartFastForward	Fast forward
L11.2	evStopbutton		StopFastForward	Standby
L12.1	evFastforwardbutton	Tape exists, tape not at the end	StartFastForward	Fast forward
L12.2	evEndtape		StopFastForward	Standby
L13.1	evRecord	Tape exists, tape not at the end	StartRecord	Record
L13.2	evStopbutton		StopRecord	Standby
L14.1	evRecordbutton	Tape exists, tape not at the end	StartRecord	Record
L14.2	evEndtape		StopRecord	Standby

11.2.2.5 Composing the test script illegal test cases

The illegal state–event combinations can be read from the state–event table. Illegal combinations refer to those where the system is not specified to respond to the event in the particular state. The expected result of all illegal test cases is that the system does not respond.

Note, an illegal test case should not be confused with a test case in which the system is explicitly specified not to process the event and instead to output an error message. The error message in this case is, in fact, the expected response of the system. If, for example, it is stated that for the state–event combination *Record* and *evPlayButton*, the system generates an error message but at the same time continues recording, then this is considered to be a legal test case. Is this error message not specified (illegal combination) and the error message does occur, then this is considered a defect.

In the test script, the start situation (the accepting state from the illegal state–event combination) is described. If this state does not correspond with the initial state, then a path to this state is created. This is done by using the test steps described in the legal test script. The result of this step is shown in Table 11.6. The test steps (IDs from Table 11.5) in the second column must be carried out to reach the state mentioned in the third column. A column for the expected results is not needed. Instead, an empty column is provided to record any unexpected system responses.

Table 11.6 Test script of illegal test cases

ID	Setup	State	Event	Result
I1		Standby	evStopbutton	
I2		Standby	evEndtape	
I3		Standby	evBeginningtape	
I4	L1.1	Rewind	evRewind	
I5	L1.1	Rewind	evRecord	
I6	L1.1	Rewind	evEndtape	
I7	L5.1	Play	evPlay	
I8	L5.1	Play	evPlay	
I9	L5.1	Play	evBeginningtape	
I10	L9.1	Fast forward	evFastforwardbutton	
I11	L9.1	Fast forward	evRecordbutton	
I12	L9.1	Fast forward	evBeginningtape	
I13	L13.1	Record	evRewindbutton	
I14	L13.1	Record	evFastforwardbutton	
I15	L13.1	Record	evRecordbutton	
I16	L13.1	Record	evBeginningtape	

11.2.2.6 Composing the test script guards

If the guard consists of a condition that holds a boundary value then the guard is subject to a boundary value analysis. For each guard, the boundary condition and a test case is used from both the right and left side of the limit.

If the guard consists of a complex condition then this is covered by the modified condition/decision coverage principle (see section 11.4).

Example: A transition only takes place for an event X if the timer > 100 s. This means three test cases, of which two are illegal – a test case with timer = 100 + ∂, where ∂ is the smallest number that can be represented in the type (e.g. 1 for integers), a test case with timer = 100 s and a test case with timer = 100 – ∂.

The results expected for the test cases is as follows:

- guard = `true`; the transition to which the guard is connected takes place;
- guard = `false`;
  - another guard is `true` and the transition related to the guard evaluated `true` takes place;
  - no reaction (➜ `IGNORE`).

In the guarded test case table, as in the table of illegal test cases, the path that leads to the start situation is given. This path is copied from the test script with legal test cases. In this example, there is no need for this because for all test cases the initial state is the start situation (therefore the *Setup* column is empty). If a test case is a test step in one of the test cases described in the script with legal test cases, then this is mentioned in the *Setup* column (see Table 11.7). The IDs of the guard's test cases refer to the IDs of the guards in the transition tree.

11.2.3 Practical issues

11.2.3.1 Testing hierarchical statecharts

Until now, it has been conveniently assumed that a statechart describes only one level of hierarchy (in fact, a flat statechart) and that there has been no relationship with other statecharts. In practice, however, statecharts can consist of different levels of hierarchy. A top-level statechart consequently breaks up into several underlying statecharts. These statecharts can be tested in either a bottom-up or a top-down manner.

If details of the underlying levels are not known, it is more practical to use a top-down approach. Another reason to select this approach could be that there is need for more insight into whether or not the several subsystems are connected properly. The underlying level is only reviewed at superstate level. Only the incoming and outgoing transitions of the superstate matter.

In the bottom-up approach, the lowest level of hierarchy is tested first. Successively higher levels of integration are then tested until finally the complete system has been tested. Levels of hierarchy functionality in the lower levels are tested in a black-box manner, that is only the incoming and outgoing transitions of superstates are included in the test.

Table 11.7
Test script with test cases for the guards

ID	Setup	State	Event	Condition	Expected result
G1.1	covered (L1.1)	Standby	evFastforwardbutton	Tape exists, tape not at the beginning	StartRewind; state Rewind
G1.2		Standby	evRewindbutton	Tape exists, tape at the beginning	IGNORE
G1.3		Standby	evRewindbutton	Tape does not exist	IGNORE
G2.1	covered (L5.1)	Standby	evPlaybutton	Tape exists, tape not at the end	StartPlay; state Play
G2.2		Standby	evPlaybutton	Tape exists, tape at the end	IGNORE
G2.3		Standby	evPlaybutton	Tape does not exist	IGNORE
G3.1	covered (L9.1)	Standby	evFastforwardbutton	Tape exists, tape not at the end	StartFastForward; state Fast forward
G3.2		Standby	evFastforwardbutton	Tape exists, tape at the end	IGNORE
G3.3		Standby	evFastforwardbutton	Tape does not exist	IGNORE
G4.1	covered (L13.1)	Standby	evRecord	Tape exists, tape not at the end	StartRecord; state Record
G4.2		Standby	evRecord	Tape exists, tape at the end	IGNORE
G4.3		Standby	evRecord	Tape does not exist	IGNORE

In both cases, the technique is applied in the same fashion. In top-down testing, underlying levels of hierarchy are not taken into account as they will be included in subsequent tests. Likewise, in bottom-up testing, underlying levels of hierarchy are not taken into account because they were tested in previous tests.

11.2.3.2 Extensiveness

The effectiveness of the state-based test method can be modified by deciding to what extent dependencies between successive transitions are tested. A transition can be tested individually, as in the description of the state-based test, but it is also possible to test a combination of transitions. The number of transitions tested in a combination is known as the *test depth level*.

The test depth level is used in the calculation of test coverage. In the literature, the terms *n-transition coverage* or *n–1 switch coverage* are used, where *n* stands for the number of transitions (and therefore also for the test depth level).

$$\text{1-transition coverage / 0-switch coverage} = \frac{\begin{array}{c}\textit{NumberOf}\\ \textit{TransitionsExcerised}\end{array}}{\begin{array}{c}\textit{TotalNumberOf}\\ \textit{TransitionsInTheStateModel}\end{array}}$$

$$\text{2-transition coverage/1-switch coverage} = \frac{\begin{array}{c}\textit{NumberOfSequencesOf}\\ \textit{TwoTransitionsExcerised}\end{array}}{\begin{array}{c}\textit{TotalNumberOfSequences}\\ \textit{OfTwoTransitions}\\ \textit{InTheStateModel}\end{array}}$$

Test depth level n (n-transition coverage or n–1 switch coverage) refers to the coverage of all n-successive transitions.

In the above coverage description, we only pass judgement with regard to coverage from the standpoint of "positive testing" (the testing of correct paths). Coverage measurements can be extended to include negative testing by considering the number of state–event combinations covered.

$$\text{State–event coverage} = \frac{\textit{NumberOfState} - \textit{EventPairsExercised}}{\textit{NumberOfState} \times \textit{NumberOfEvents}}$$

In this case, the coverage level cannot be used to infer any judgement with regards to test quality. The coverage level only indicates whether or not something has been overlooked during the specification.

11.2.3.3 Fault detection

The state-based test design technique described here is based on test depth level 1. This test depth level is in fact the minimum test effort required to be conducted for state-based behavior. Binder (2000) considers this insufficient and favours the application of the round trip test. That is, the application of test depth level n (n is the number of transitions in the model). This requires quite some effort and is only applied when it concerns critical parts of the system.

In section 11.2.1, a number of fault categories were mentioned that are covered with this method. Table 11.8 shows which parts of the state-based test design technique contribute to the fault detection. In addition, this table reveals what an increase in test depth level can add to the fault detection.

Increasing the test depth level may not be sufficient to detect corrupt states (fault category 1.5) or hidden paths (fault category 4.2). Often, such defects only manifest themselves after lots of test actions are repeated many times without resetting the system to its initial state. Of course, the trouble is that it cannot be predicted how many repeated test sequences are required to assure that no such defects are present.

Based on this table, choices can be made with regard to the use of various parts of the state-based test design technique and the depth of testing.

11.2.3.4 Practicality and feasibility

The testing of a system's state-based behavior doesn't always turn out to be simple in practice. The behavior of a system is hard to determine because the

behavior is concealed, to a large extent, from outside observation. In object orientation, for example, data encapsulation is used to hide the system's data state. An event can lead to a cascade of transitions before the system reaches a state that can be observed. Because of this, it is hard, or even impossible, to distinguish between individual transitions and states.

Table 11.8 Test effort versus fault detection

Fault category	Legal test cases	Illegal test cases	Guard test cases	Higher depth of testing
1.3 Extra state	+	+	+	+
1.4 Missing state	+	–	–	–
1.5 Corrupt state[1]	–	+/–	+/–	+
2.4 Overlapping guards	–	–	+/–	–
2.6 Transition with guard `false`	–	–	+	–
3.2 Conflicting transition	+	–	–	–
3.3 Missing or incorrect transitions	+	+	+	+/–
3.4 Missing or incorrect action	+	–	–	+/–
4.1 Missing or incorrect event	+	–	–	+/–
4.2 Hidden path[1]	–	+	+	+/–

"–" means that there is a very slim chance that a fault is detected

"+/–" means that there is a small chance that the fault will be detected

"+" means that there is a good chance the fault will be detected

[1] Increasing the test depth level may not be sufficient to detect corrupt states (fault category 1.5) or hidden paths (fault category 4.2). Often, such defects only manifest themselves after lots of test actions are repeated many times without resetting the system to its initial state. Of course, the trouble is that it cannot be predicted how many repeated test sequences are required to assure that no such defects are present.

There are a number of factors that positively influence the practicability and the determination of the system's reaction.

- *Step mode*. This provides the option to execute the system one transition at a time.
- *Reset option*. This provides the option to return to the initial state.
- *State set*. This makes it possible to set the system in a certain state.
- *Unique coding* for states, transitions, inputs, outputs, and events combined with the option to request this information at any time, provides insight into the system's state at any time.
- *Transition trace*. Provides the option to view the transition sequence.

In order to test the software, either a simulator, UML-compiler (see Douglass (1999) for a comparison of the relative advantages/disadvantages of simulators and compilers) or the end product is used. Simulators often include the step mode, reset, and state set functions. This may not be the case for compilers – "tailor-made" compilers are often used. For the tests' practicality, it is useful to incorporate the first three options as functions within the compiler. The end product merely provides the functionality required from the user's point of view. Because of this, the options mentioned will be mainly absent.

Unique coding must be realized in the coding or the modeling. Not only must every state, transition, input, output, and event have unique names, but also the system's status in relation to these items has to be "requestable." This can be done, for example, by linking a Boolean to a state. When the state is reached then the Boolean is set to `true`, and once the state is left the Boolean is set to `false`. If code is executed which is related to inputs, outputs, or events, this is logged in a trace file. In this way, a transition trace is set up, in which the order, in time, of actions, transitions, states, inputs, outputs, and events can be read. This is particularly useful when the test environment does not provide a step mode for walking through the model or code.

11.3 Control flow test

11.3.1 Introduction

The objective of the control flow test is to test the program structure. The test cases are derived from the structure of an algorithm and/or program. Every test case consists of a group of actions covering a certain path through the algorithm. The control flow test is a formal test design technique mostly used in unit tests and integration tests.

11.3.2 Procedure

The control flow test consists of the following steps:

1 making an inventory of decision points;
2 determining the test paths;
3 specifying the test cases;
4 establishing the initial data set;
5 assembling the test script;
6 executing the test.

11.3.2.1 Decision points

The program design or the technical design is used as the test basis. This should contain a description of the structure of the algorithm under test – a flow chart, decision table, activity diagrams, etc. If the test basis doesn't provide this information then it may be necessary to prepare documentation about the program structure based on the information available. If this is not possible, control flow testing can't be used in this case.

All decision points in the program structure must be identified and uniquely labeled. The actions between two consecutive decision points are taken, for this inventory, to be one big action and are also labeled uniquely.

11.3.2.2 Test paths

The combination of actions into test paths depends on the desired test depth level. The test depth level is used to decide to what extent dependencies between consecutive decision points are tested. Test depth level *n* means all dependencies of actions before a decision point and after *n*–1 decision points are verified. All combinations of *n* consecutive actions are used.

The test depth level has a direct effect on the number of test cases and on the degree of coverage of the tests. The direct effect on the number of test cases also means there is a direct effect on the test effort.

To show how the test depth level has an effect on the test cases, an example for test depth level 2 is given. For every decision point identified, the action combinations are determined. For test depth level 2 an inventory is made of all the possible action combinations of two consecutive actions. An action combination in test depth level 2 is a combination of an action before and one after a decision point. All possible combinations are mentioned. As an example the flowchart in Figure 11.3 is used.

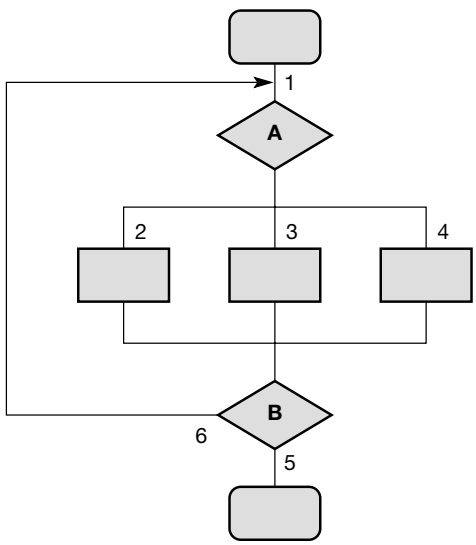


Figure 11.3
Flow chart of a sample program structure

The action combinations for the decision points are:

A : (1,2); (1,3); (1,4); (6,2); (6,3); (6,4)
B : (2,5); (3,5); (4,5); (2,6); (3,6); (4,6)

1 Put the action combinations in ascending order: (1,2); (1,3); (1,4); (2,5); (2,6); (3,5); (3,6); (4,5); (4,6); (6,2); (6,3); (6,4).
2 The action combinations have to be linked to create paths running from the start to the end of the algorithm. In this example, this means every path has to start with action 1 and end with action 5.
3 Start with the first action combination that has not yet been included in a path. In this case, this is (1,2). Subsequently, the first action combination that starts with 2 and has not yet been included in a path is linked after the first, in this case (2,5). This creates the path (1,2,5) and the next path can be started.
4 The remaining action combinations are: ~~(1,2)~~; (1,3); (1,4); ~~(2,5)~~; (2,6); (3,5); (3,6); (4,5); (4,6); (6,2); (6,3); (6,4).
5 Continue with the remaining action combinations. The first action combination that has not yet been included in a path is (1,3). The first action combination that starts with 3 and has not yet been included in a path is (3,5). This creates the path (1,3,5) and the next path can be started.
6 The remaining action combinations are: ~~(1,2)~~; ~~(1,3)~~; (1,4); ~~(2,5)~~; (2,6); ~~(3,5)~~; (3,6); (4,5); (4,6); (6,2); (6,3); (6,4).
7 Continue with the remaining action combinations. The first action combination that has not yet been included in a path is (1,4). The first action combination that starts with 4 and has not yet been included in a path is (4,5). This creates the path (1,4,5) and the next path can be started.
8 The remaining action combinations are: ~~(1,2)~~; ~~(1,3)~~; ~~(1,4)~~; ~~(2,5)~~; (2,6); ~~(3,5)~~; (3,6); ~~(4,5)~~; (4,6); (6,2); (6,3); (6,4).
9 The first action combination that has not yet been included in a path is (2,6). Because this action combination does not start at the beginning of the algorithm, a preceding action combination must be determined. (1,2) is selected for this. This has now been used twice, but that is clearly not a problem. Then continue with an action combination that starts with 6 and has not yet been included in a path. Action combination (6,2) is selected. The test path is completed with combination (2,5), which creates test path (1,2,6,2,5).
10 The remaining action combinations are: ~~(1,2)~~; ~~(1,3)~~; ~~(1,4)~~; ~~(2,5)~~; ~~(2,6)~~; ~~(3,5)~~; (3,6); ~~(4,5)~~; (4,6); ~~(6,2)~~; (6,3); (6,4).
11 The rest of the action combinations are included in test path (1,3,6,4,6,3,5) and all action combinations have now been included in the following paths:

Path 1: (1,2,5)
Path 2: (1,3,5)
Path 3: (1,4,5)
Path 4: (1,2,6,2,5)
Path 5: (1,3,6,4,6,3,5)

Test depth level two is based on the idea that execution of an action may have consequences for the action immediately after a decision point. Not only the decision, but also the previous action is important for an action. Contrarily, test depth level 1 assumes that an action is only affected by the decision.

For test depth level 1, the following action combinations arise:

– : (1)
A : (2); (3); (4)
B : (5); (6)

This will lead to the following path combinations:

Path 1: (1,2,5)
Path 2: (1,3,6,4,5)

A lower test depth level leads to a smaller test effort in exchange for a lesser degree of coverage.

Higher test depth levels imply that the execution of an action affects two or more subsequent actions. High test depth levels are only used in safety-related parts or very complex parts of the program.

11.3.2.3 Test cases

The derived logical test paths are translated into physical test cases. For each test path, the test input must be determined. The determination of the right test input to follow a certain path can be quite difficult. The test input must be chosen such that at every decision point the correct path is taken. If the input is changed during its path through the algorithm, its values will have to be calculated back to the start point. Sometimes the algorithm is so complicated that this is practically impossible – for example, when scientific-based algorithms are used. For every test input, an output prediction must be specified. Variables and parameters that have no impact on the path should receive a default value.

11.3.2.4 Initial data set

Sometimes the execution requires a specific initial data set. This has to be described very concisely. This description is added to the description of the test paths and the test cases.

11.3.2.5 Test script

All previous steps form the basis for the test script. The test script describes the test actions and checks to be executed in the right sequence. The test script also lists the preconditions for the test script to be executed. Preconditions often concern the presence of an initial data set, that the system should be in a certain state, or that stubs and drivers are available.

11.3.2.6 Test execution

The test is executed according to the test script. The result is compared with the output prediction. In the case of a unit test, the tester and the developer are often the same person. The next step is detecting the cause of the differences between output and predicted output.

11.4 Elementary comparison test

11.4.1 Introduction

In the elementary comparison test (ECT), the processing is tested in detail. The test verifies all the functional paths of a function. All functional conditions have to be identified and translated into pseudocode. The test cases are derived from the pseudocode and cover the identified functional paths.

The ECT establishes condition coverage and guarantees a reasonable degree of completeness. Because this formal technique is labor intensive, it is mainly used in very important functions and/or complex calculations.

11.4.2 Procedure

The ECT technique consists of the following steps:

1 analyzing the function description;
2 establishing test situations;
3 establishing logical test cases;
4 establishing physical test cases;
5 establishing test actions;
6 establishing checks;
7 establishing the start situation;
8 assembling the test script.

To illustrate these steps, the following functional description is used:

```
IF TempSensor_1 - TempSensor_2 ≤ 40
THEN Actor_heater_1 = OFF
ELSE
   IF TempSensor_2 ≥ 70 AND TempSensor_1 < 30 AND Actor_valve_1 = ON
         Actor_valve_2 = OFF
   ENDIF
   IF VolumeSensor < 1000 AND Actor_valve_2 = ON AND Actor_heater_1
            = ON AND (Actor_heater_2 = OFF OR TempSensor_2 ≥ 50)
         Actor_valve_1 = OFF
   ENDIF
ENDIF
```

11.4.2.1 Analyzing the function description

The function description should describe unambiguously the decision paths and related aspects. The function can already be described as pseudocode or by means of another technique, for example decision tables. Even if there is only a free format description of the function, the ECT can be applied if the decision paths are described unambiguously. It is very helpful to translate the function description into pseudocode if it is not already.

The first step is to identify the conditions. These can often be recognized by terms such as DO, IF, REPEAT, etc. These are singled out one by one, and given a unique identification (**C1**, etc. below). Only the conditions driven by data input are taken into account. Conditions such as `DO as long as counter < 10` are driven internally and not through explicit data input and will therefore be ignored.

```
C1  IF TempSensor1 - TempSensor2 ≤ 40
    THEN Actorheater1 = OFF
    ELSE
C2        IF TempSensor1 ≥ 70 AND TempSensor2 < 30 AND Actorvalve1 = ON
                Actorvalve2 = OFF
          ENDIF
C3        IF VolumeSensor < 1000 AND Actorvalve2 = ON AND Actorheater1 = ON
          AND (Actorheater2 = OFF OR TempSensor1 ≥ 50)
                Actorvalve1 = OFF
          ENDIF
    ENDIF
```

11.4.2.2 Establishing test situations

After the conditions have been identified, the situations to be tested for each condition must be determined. It is very important to differentiate between simple and complex conditions. Simple conditions consist of only one comparison, the elementary comparison. The C1 condition is an example of an elementary comparison. Complex conditions are those containing multiple comparisons, connected by AND or OR relationships. The C2 condition is an example of such a condition.

Simple conditions result in two situations, namely the condition is `false` or `true`. In the current example, this will lead to the two test situations shown in Table 11.9.

Table 11.9 Test situations for condition C1

Test situation	**1**	**2**
C1	true	false
$TempSensor_1 - TempSensor_2$	≤ 40	> 40

A complex condition is a combination of simple conditions. The complex condition is either `true` or `false`, but this depends on whether the constituent simple conditions are `true` or `false`. Whether the conditions are connected by the AND or OR is very important. The role of the conjunction on the value of the complex conditions is shown in Table 11.10.

Table 11.10 Possible situations of a complex condition for an AND and OR conjunction

Complex condition		Result	
A	**B**	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

A complex condition with an AND conjunction is only `true` if the two simple conditions are `true`. While the complex condition with an OR conjunction is `true` if only one of the two simple conditions are `true`.

The possible situations of complex conditions are exponentially related to the simple conditions. A complex condition of two simple conditions has $2^2 = 4$ possible situations, while a complex condition of three simple conditions has $2^3 = 8$ possible situations. For a combination of seven simple conditions this means 128 possibilities. ECT is based on the fact that it is not useful to identify all these possibilities as situations to be tested. Test situations are selected such that a change in the value of any one of the elementary comparisons changes the value of the complex condition. For example, the situations `true–true` in the case of an OR relationship, and `false–false` in the case of an AND relationship are neglected. The chances of finding a defect not already observed in the `true–false` or `false–true` situations are minimal.

In the case of an AND relationship, only those situations will be tested in which each elementary comparison has a truth value of 1, as well all those in which one and only one elementary comparison has a truth value of 0. The opposite applies to an OR relationship – the only situations tested are those in which each elementary comparison has a truth value of 0, and all those in which one and only one elementary comparison has a truth value of 1.

ECT reduces the number of test situations to the number of elementary comparisons plus 1.

The result of the application of ECT to the complex condition C2 is shown in Table 11.11.

Table 11.11 Test situations of the complex condition C2

Test situation	**C2.1**	**C2.2**	**C2.3**	**C2.4**
C2(=C2a & C2b & C2c)	1(111)	0(011)	0(101)	0(110)
C2a $\texttt{TempSensor}_1 \geq 70$ AND	≥ 70	< 70	≥ 70	≥ 70
C2b $\texttt{TempSensor}_2 < 30$ AND	< 30	< 30	≥ 30	< 30
C2c $\texttt{Actor}_{\texttt{valve1}} = \texttt{ON}$	ON	ON	ON	OFF

For complex conditions with only AND or OR relationships, the test situations are easy to find. It becomes more difficult when the complex condition consists of a combination of one or more AND and OR relationships, such as in the C3 example. In such cases, a matrix is used (see Table 11.12). In the first column, the elementary comparisons are enumerated. The second column shows how the truth value true for the complex condition can be arrived at such that a change in the value of the elementary comparison (its value is underlined) would change the value of the complex condition to false. The resulting situation is given in the last column. Obviously the only difference between the second and the third columns are the underlined values.

Table 11.12 Test situations for condition C3

C3 (=C3a & C3b & C3c & (C3dl C3e)	**1 (**$Actor_{valve2}$ = OFF**)**	**0**
VolumeSensor < 1000 (C3a)	1.1.1.01	0.1.1.01
$Actor_{valve2}$ = ON (C3b)	1.1.1.01	1.0.1.01
$Actor_{heater1}$ = ON (C3c)	1.1.1.01	1.1.0.01
$Actor_{heater2}$ = OFF (C3d)	1.1.1.10	1.1.1.00
TempSensor ≥ 50 (C3e)	1.1.1.01	1.1.1.00

After eliminating the repetitive test situations (see Table 11.13), the logical test situations are left.

Table 11.13 The remaining test situations for condition C3

C3 (=C3a & C3b & C3c & (C3dl C3e)	**1 (**$Actor_{valve2}$ = OFF**)**	**0**
VolumeSensor < 1000 (C3a)	1.1.1.01	0.1.1.01
$Actor_{valve2}$ = ON (C3b)	~~1.1.1.01~~	1.0.1.01
$Actor_{heater1}$ = ON (C3c)	~~1.1.1.01~~	1.1.0.01
$Actor_{heater2}$ = OFF (C3d)	1.1.1.10	1.1.1.00
$TempSensor_1$ ≥ 50 (C3e)	~~1.1.1.01~~	~~1.1.1.00~~

In the fifth row, the complex condition (C3d | C3e) is 10 rather than 01 because a change of the latter in the first position will lead to a situation (11) that can be ignored.

The next step is to translate the remaining test situations to logical test situations (see Table 11.14).

Table 11.14
Logical test situation of condition C3

Test situation C3	**C3.1** 1(11101)	**C3.2** 1(11110)	**C3.3** 0(01101)	**C3.4** 0(10101)	**C3.5** 0(11001)	**C3.6** 0(11100)
VolumeSensor	< 1000	< 1000	≥ 1000	< 1000	< 1000	< 1000
$Actor_{valve2}$	ON	ON	ON	OFF	ON	ON
$Actor_{heater1}$	ON	ON	ON	ON	OFF	ON
$Actor_{heater2}$	ON	OFF	ON	ON	ON	ON
$TempSensor_{1}$	≥ 50	< 50	≥ 50	≥ 50	≥ 50	< 50

11.4.2.3 Establishing logical test cases

Establishing the logical test cases means finding the functional paths in which each situation of each condition should be completed at least once. This approach can lead to the occurrence of the same comparison in more than one condition. When selecting path combinations, one should also ensure that the expected end result is as unique as possible for the chosen path combinations. Note that the subsequent conditions can influence each other. In this example, if condition 2 is `true` this will change $Actor_{valve2}$ to OFF and this will automatically mean that condition 3 will be `false`.

To help find test cases, and check whether all situations in the test cases have been recorded, a matrix can be used (see Table 11.15). All defined test situations are listed in the first column. The second column lists the value of the condition in this situation, and the third column contains the next condition to be processed. The test cases are then listed. If the test cases have been defined properly, all situations to be tested will be ticked off at least once. To cover all test situations in our example, seven test cases have been defined.

Some situations will be tested several times. This often unavoidable, and should be seen not as a waste of time but as an opportunity – by varying the input for that situation, more thorough testing is achieved.

Test situation	Value	To	1	2	3	4	5	6	7	Control
C1.1	1	End	×							1
C1.2	0	C2		×	×	×	×	×	×	6
C2.1	1	C3		×						1
C2.2	0	C3			×			×		2
C2.3	0	C3				×			×	2
C2.4	0	C3					×			1
C3.1	1	End						×		1
C3.2	1	End			×					1
C3.3	0	End					×			1
C3.4	0	End		×						1
C3.5	0	End				×				1
C3.6	0	End							×	1

Table 11.15
Matrix to help construct the logical test cases

11.4.2.4 Establishing physical test cases

The next step is to convert the logical test cases into physical test cases (see Table 11.16).

Test case	1	2	3	4	5	6	7
Path	C1.1	C1.2, C2.1, C3.4	C1.2, C2.2, C3.2	C1.2, C2.3, C3.5	C1.2, C2.4, C3.3	C1.2, C2.2, C3.1	C1.2, C2.3, C3.6
$TempSensor_1$	50	90	45	90	90	60	90
$TempSensor_2$	40	20	5	40	20	20	40
$Actor_{valve1}$	OFF	ON	ON	ON	OFF	ON	ON
$Actor_{valve2}$	OFF	ON	ON	OFF	ON	ON	ON
$Actor_{heater1}$	OFF	ON	OFF	OFF	ON	ON	ON
$Actor_{heater2}$	ON	ON	ON	ON	ON	ON	ON
VolumeSensor	500	500	500	500	1200	500	500

Table 11.16
Physical test cases

11.4.2.5 Establishing test actions

A test action is a predefined action that can be successful or can fail. For this purpose, all relevant subtasks are listed first. Our example contains only one test action and that is the start of the system.

11.4.2.6 Establishing checks

Checks are necessary to determine whether processing has been successful. A check is a description of the expected situation after a test case or test action has been executed. Table 11.17 shows the list of checks for the example used in this chapter.

Table 11.17 All checks

Check	Test case	Expected situation
C01	1	$Actor_{heater1}$ = OFF and other actors unchanged
C02	2	$Actor_{valve2}$ = OFF and other actors unchanged
C03	3	$Actor_{valve1}$ = OFF and other actors unchanged
C04	4	All actors unchanged
C05	5	All actors unchanged
C06	6	$Actor_{valve1}$ = OFF and other actors unchanged
C07	7	All actors unchanged

11.4.2.7 Establishing the start situation

Sometimes specific preconditions must be fulfilled before test case execution can start. These preconditions can be that a certain initial data set must be installed or that the system has to be set in a certain state. The state and the initial data set can differ from one test case to another. In our example, no special measures are necessary to establish the start situation.

11.4.2.8 Assembling the test script

The test script should provide all the information necessary to execute the test cases. Therefore a test script should contain the test cases and their related actions and checks in the correct order. In addition, all pre- and post-conditions are listed and the start situation is described.

11.5 Classification-tree method

11.5.1 Introduction

The classification-tree method (CTM) supports the systematic design of black-box test cases. It is an informal test design technique, although there is a well-described procedure to derive test cases. CTM identifies the relevant aspects

of the system under test – aspects that can influence, for instance, the functional behavior or safety of the system. The input domain of the test object is partitioned in disjoint classes using equivalence partitioning according to the aspects identified. The input domain partition is represented graphically in the form of a tree. The test cases are formed by a combination of the classes of different aspects. This is done by using the tree as the head of a combination table in which the test cases are marked.

Ideally, CTM is used with the functional requirements as test basis. Unfortunately these requirements are not always available or complete and up to date. A distinct advantage of CTM is that even in this situation it can be applied – the tester fills in the missing information. The tester should have at least a basic understanding of the system under test. If the functional requirements are missing, the tester must have in-depth knowledge about the desired functional behavior of the system.

The classification-tree method was developed by Grochtmann and Grimm (Grochtmann and Grimm, 1993). An extension for embedded systems specially developed for mixed signal description is given in section 16.2.2.4.

11.5.2 Procedure

The CTM technique consists of the following steps:

1 identifying aspects of test object;
2 partitioning input domain according to aspects;
3 specifying logical test cases;
4 assembling the test script.

This procedure can be supported by tools such as the classification-tree editor (Grochtmann, 1994).

To illustrate these steps an example of some cruise control software is used. The example is simplified and many aspects are not taken into account. The only aspects considered are the speed of the car and the difference between the actual speed and the desired speed. As a complicating factor, this cruise control system is also capable of reacting to cars in front of it. The difference in speed between a car in front and the car with the cruise control system is important for the system's behavior. In addition, the distance between the car in front and the car with the cruise control system influences the behavior of the system. In reality, most of the aspects mentioned here should be described as a function of time, place, and speed.

11.5.2.1 Identifying aspects of test object

The first step is the identification of the aspects that influence the processing of the system under test. For the example, these are the speed of the car, the speed difference between the actual speed and the desired speed, and whether there is a car in front or not. These aspects are graphically visualized in a tree (see Figure 11.4).

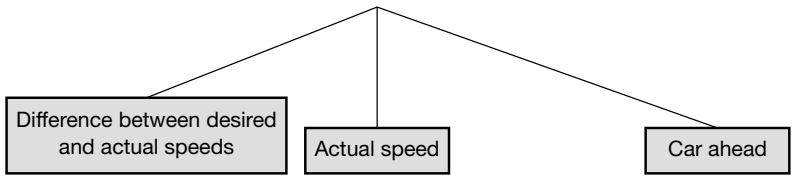


Figure 11.4 Three aspects of the intelligent cruise control

11.5.2.2 Partitioning input domain according to aspects

The input domain of the test object is partitioned into classes according to the aspects identified. For every member of a class, the system shows the same kind of behavior. This partitioning method is called equivalence partitioning (see section 11.1.4.2). The classes are disjoint, meaning that every input value can only be a member of one class and exactly a member of one class. Figure 11.5 shows the result of the partitioning into classes.

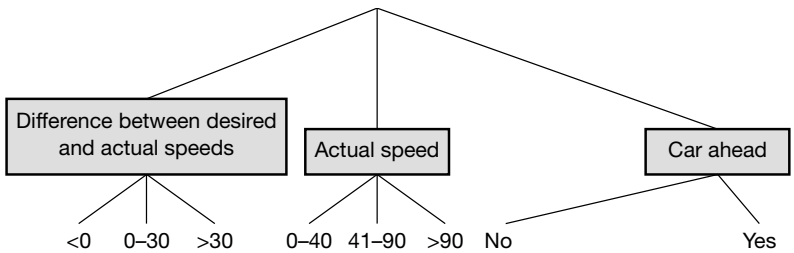


Figure 11.5 Partitioning of the input domains

Building up a classification-tree can be a recursive activity. In this example, if there is a car ahead the distance to it and the speed difference between the cars influence the functioning of the system. The Yes class of the aspect "Car ahead" forms the root for another level of the classification-tree (see Figure 11.6). Identification of aspects and partitioning of the input domain according to these aspects repeats itself until all relevant aspects are identified and the partitioning of the input domain is finished.

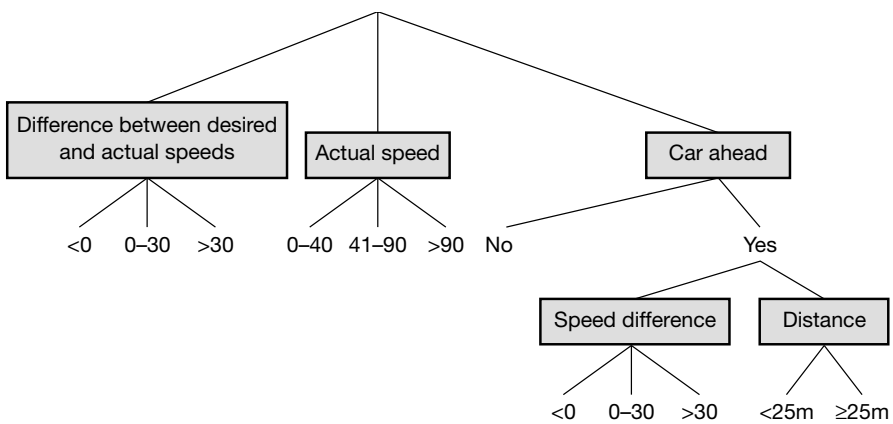


Figure 11.6 The complete classification-tree after recursive application of steps one and two

11.5.2.3 Specifying logical test cases

When the classification-tree is finished, all the information needed for preparing test cases is available. A test case is formed by:

- selecting a class for each aspect;
- combining these classes.

The minimum number of test cases is achieved when every class is covered by at least one test case. This minimal variant of this approach will lead in our example to four test cases (see Figure 11.7). When more thorough testing is necessary, combinations of classes must be covered by test cases. The theoretical maximal variant requires that every possible combination of the classes of all aspects must be covered by test cases. Due to impossible cases (for example, logically inconsistent test cases), the practically achievable number is, in most cases, smaller. In our example this means $3 \times 3 \times (1 + (3 \times 2)) = 63$ test cases. (Every combination of the classes of "Difference between desired and actual speed" and 'Actual speed' must be combined with all classes of "Car ahead." "Car ahead" has two classes, one of which is subdivided into two aspects with three and two classes.)

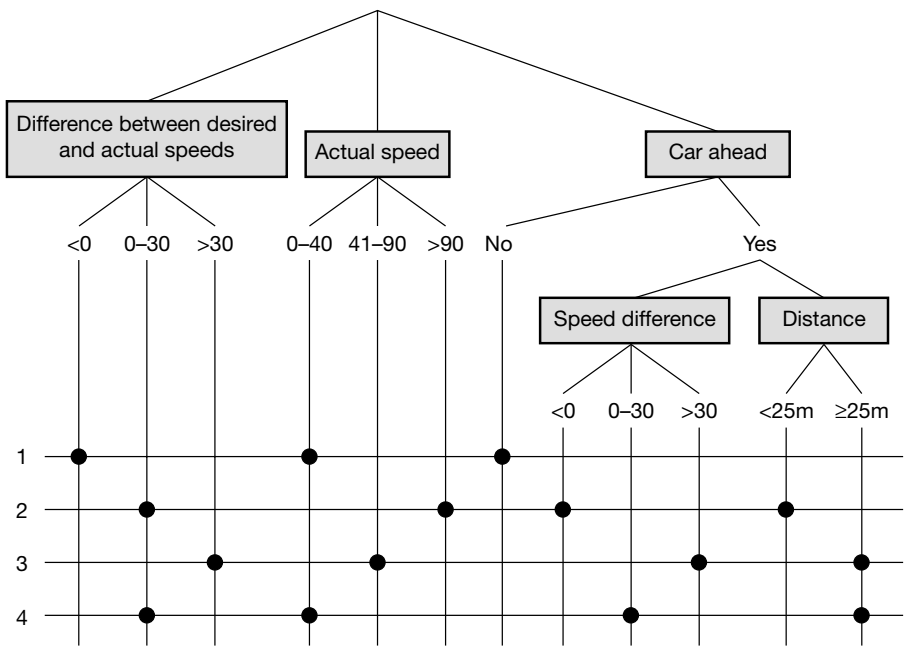


Figure 11.7
Every row is one logical test case – every class is covered at least once with a minimum of four logical test cases

11.5.2.4 Assembling the test script

The preparation of a test script consists of the following activities:

1 constructing physical test cases;
2 defining test actions;
3 defining check points;
4 determining the start situation;
5 establishing the test script.

Constructing physical test cases

The logical test cases are a combination of classes. In order to execute those test cases for every class, values have to be chosen representing the corresponding classes. For the example this could lead to the physical test cases shown in Table 11.18.

Table 11.18 Physical test cases

Test case	Aspect	Value	Comment
1	Speed of car	39 km/h	
	Desired speed	30 km/h	Difference between desired/actual speed: –9 km/h
	Car ahead	No	
2	Speed of car	100 km/h	
	Desired speed	120 km/h	Difference between desired/actual speed: 20 km/h
	Speed of car ahead	90 km/h	Difference between car ahead/car: –10 km/h
	Distance	10 m	
3	Speed of car	65 km/h	
	Desired speed	135 km/h	Difference between desired/actual speed: 70 km/h
	Speed of car ahead	105 km/h	Difference between car ahead/car: 40 km/h
	Distance	100 m	
4	Speed of car	25 km/h	
	Desired speed	50 km/h	Difference between desired/actual speed: 25 km/h
	Speed of car ahead	55 km/h	Difference between car ahead/car: 30 km/h
	Distance	30 m	

The comments are given to show the relation to the logical test cases.

Defining test actions

A test action is a predefined action that can either be successful or it can fail. For autonomous systems the only action is to switch on the system. For systems with a high user interaction many test actions can be needed. In the latter situation the description must be quite clear because the user often has a choice of different actions.

For our example, the only actions are to switch on the cruise control and set the desired speed.

Defining check points

A checkpoint is a description of when and how the behavior of the system is checked. The expected behavior for every test case is described.

For the example, the checkpoint is the moment when the speed of a car will stay constant. The speed of the car is measured and also the distance to the car ahead:

C01 speed of the car
C02 distance to car ahead

The expected behavior for the test cases in this example is shown in Table 11.19.

Check	Test case 1	Test case 2	Test case 3	Test case 4
C01	30 km/h	90 km/h	105 km/h	50 km/h
C02		40 m	45 m	> 30 m (increasing)

Table 11.19 Expected behavior of test cases

The distance to the car ahead is a function of the speed. The desired speed can only be reached if there is no car ahead, or the car ahead has the same or higher speed compared to the desired speed.

Determining the start situation

Sometimes it is necessary that some preconditions are fulfilled before test case execution can start. These preconditions can be the availability of a certain initial data set or that the system has to be set in a certain state. The state and the initial data set can differ from one test case to another. In our example, no special measures are necessary to establish the start situation.

Establishing the test script

The test script should provide all the information needed to execute the test cases. The deliverables of the previous activities together form the information needed to assemble the test script. The test script starts with a description of the start situation. Then the test actions and checkpoint are described in the correct sequence. Further information can be added if some additional test equipment is needed or a reference is given to a list of desired equipment.

11.5.3 Complex systems

In complex systems, many aspects influence system behavior. A class of an aspect may lead to another aspect. In such situations, the classification-tree tends to become unreadable. Refinement symbols can then be used to indicate that the next level of the classification-tree is continued elsewhere. An example of this is shown in Figure 11.8.

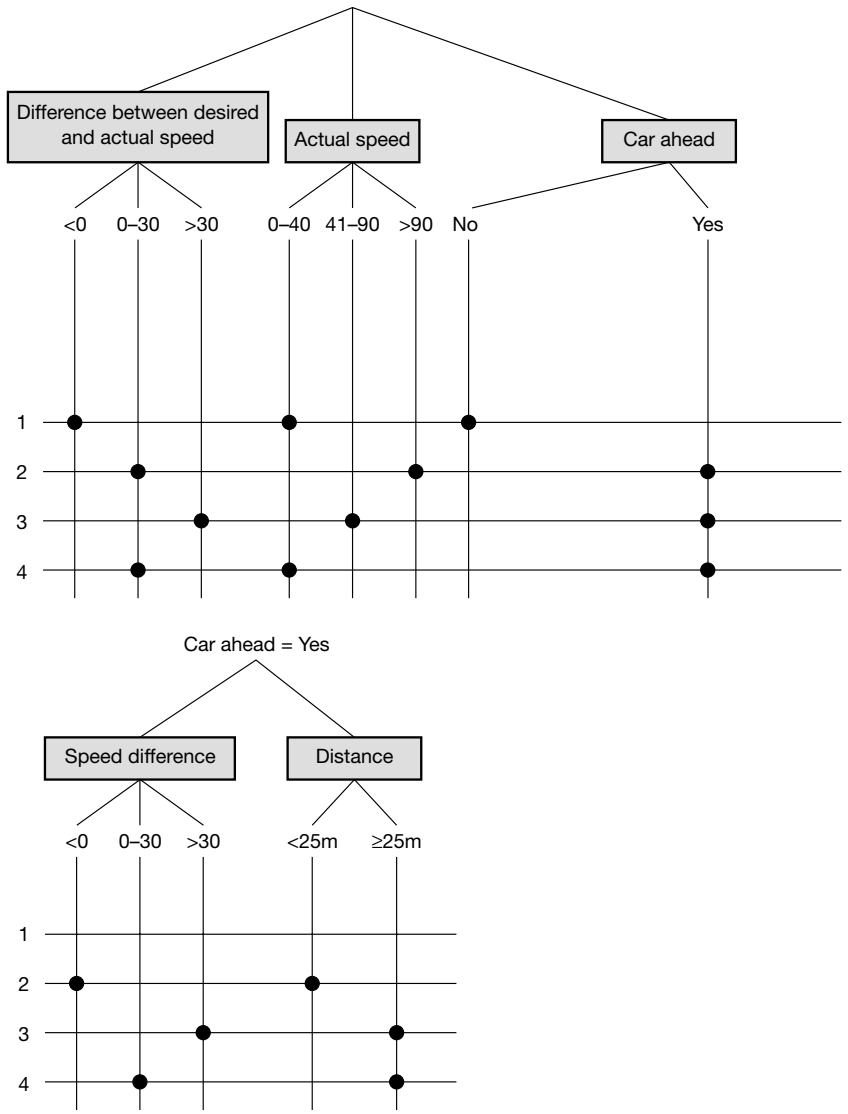


Figure 11.8 The next level of the classification is continued in another diagram

This refinement technique gives the opportunity to handle test objects with many aspects in different levels.

11.6 Evolutionary algorithms

11.6.1 Introduction

Evolutionary algorithms (also known as genetic algorithms) are search techniques and procedures based on the evolution theory of Darwin. Survival of the fittest is the driving force behind these algorithms. Evolutionary algorithms are used for optimization problems, or problems that can be translated into an optimization problem. Whereas test design techniques focus on individual test cases, evolutionary algorithms deal with "populations" and the "fitness" of individual test cases. Evolutionary algorithms can, for instance, be used to find test cases for which the system violates its timing constraints, or to create a set of test cases that cover a certain part of the code. Some application areas (Wegener and Groditmann (1998) and Wegener and Mueller (2001) for evolutionary algorithms are:

- safety testing;
- structural testing;
- mutation testing;
- robustness testing;
- temporal behavior testing.

In this section on evolutionary algorithms, only the basic principles are explained. For a more scientific background further reading of Pohlheim (2001) and Sthamer (1995) is suggested.

11.6.2 Evolutionary process

Figure 11.9 shows the steps in an evolutionary process for a single population. The process begins with the generation of the start population. Then the fitness of each test case is assessed – the system under test is used to determine this. The behavior of the system is compared with the defined optimal behavior. The closer the behavior is to the defined optimal, the higher the fitness of the test case is. This step is executed with usage of the fitness function. Defining the proper fitness function is not that easy but it is very important. Without a proper fitness function it is not possible to find the right test case.

If one of the test cases meets the exit criteria, the process stops and the result is known. Otherwise test cases are selected to form new test cases (offspring) by recombination and mutation. In the recombination step, parts of a test case are interchanged pair-wise with another test case to form two new test cases. After this, the test cases are subject to mutation whereby at a randomly chosen position in the test case, a change takes place.

The newly formed test cases have a certain (rather small) chance of becoming part of the group of test cases. In this step, called *insertion*, test cases of the original population are substituted by new test cases. The population formed in an insertion step is called the new generation. Then the process continues with fitness assessment and so on till the exit criteria are met.

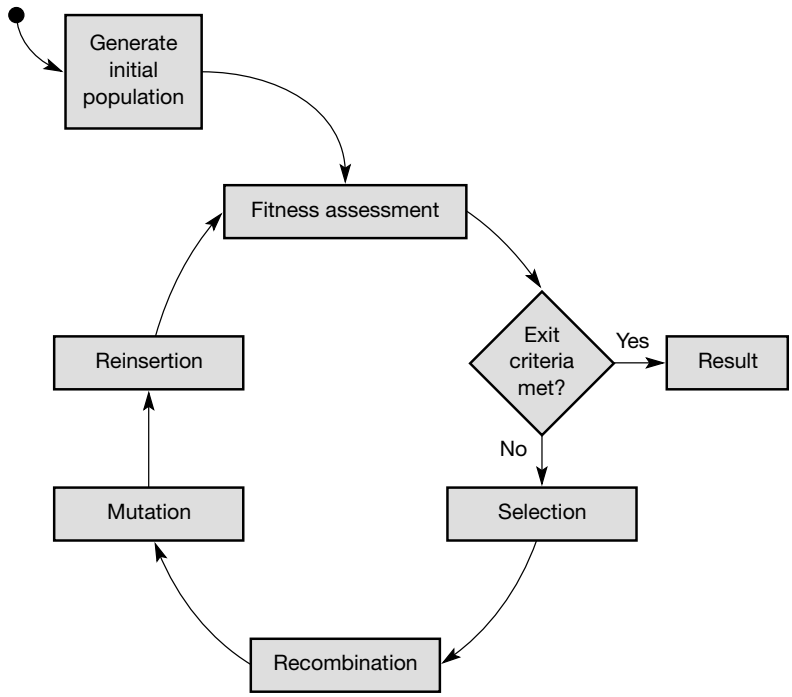


Figure 11.9
Evolutionary process

11.6.3 Process elements

The different elements of the evolution process are explained in detail in this section. To make the description of the different steps clearer, a particular example is used. A code consisting of four two-digit numbers (85 31 45 93) must be found. Every test case also consists of four two-digit numbers. The test case is added to the system and the system gives a reaction based on whether the test case matches the code or not. For every number in the correct place the system shows a black circle on a screen. For every number in the code but not in the right place, it shows a white circle. The system gives no information as to which number is in the code nor which number is in the right place. The fitness function interprets the reaction by the system. Every black circle is interpreted as ten points, every white circle as three points, and every number which is not in the code (no reaction by the system) is interpreted as one point. The maximum fitness function is 40 and that is the exit criterion.

11.6.3.1 Start population

A population is a collection of test cases. Each test case is an individual of the population. The start population is generated randomly. A start population has a size of 50–100 members (Sthamer, 1995). A small population has the advantage of requiring less computational effort and, in general, will reach the

optimum quite quickly. One disadvantage is the possibility of inbreeding, as in nature. Using a small population will sometimes lead to premature convergence and the effect is that the population gets stuck on a local optimum.

The following terms related to "population" are used in this chapter:

- a population is called *P*;
- the *x*th generation is P_x;
- the size of the population is P_{sz};
- the start population (first generation) is P_1.

The start population for the example is shown in Table 11.20.

Table 11.20 Start population

Individual	P_1
1	41 03 90 93
2	04 72 95 31
3	44 31 45 93
4	93 45 32 91

11.6.3.2 Fitness assessment

Fitness expresses how good the test case is in relation to the global optimum. The fitness is calculated for each test case and is used to compare the test cases. The closer a test case is to the optimum, the higher the fitness is. The fitness is calculated using a fitness function. Finding a suitable fitness function is not always the easiest thing to do as it should be a reliable representation of the problem to be solved. When the functon returns a high value, this can be interpreted safely as a "good solution" and vice versa. This is not always a straightforward one-to-one relationship.

- the fitness of an individual be *fi*;
- and the fitness of the population be

$$f_{\text{total}} = \sum_{i}^{P_{sz}} f_i$$

The fitness of the example population is given in Table 11.21.

Table 11.21
Fitness of the population

Individual	P_1	*fi*
1	41 03 90 93	13
2	04 72 95 31	6
3	44 31 45 93	31
4	93 45 32 91	8
	Total fitness	58

11.6.3.3 Selection

During the selection phase, test cases are selected to form new test cases. The selection can be done at random or the fitness of the test cases can be used. If the fitness is used, the test cases have to be ranked. The individual with the highest fitness has the highest chance of being selected. In our example, fitness is used for the selection of the test cases.

The first step is to normalize the fitness of the test cases. The fitness of a test case is divided by the total fitness of the population:

$$f_{i,\text{norm}} = \frac{f_i}{\sum_{i=1}^{P_{sz}} f_i}$$

where $f_{i,\ \text{norm}}$ is the normalized fitness for the i^{th} test case.

The next step is to aggergate the normalized fitness of the test cases:

$$f_{i,\ \text{accu}} = \sum_{k=1}^{i} f_{k,\ \text{norm}}$$

where $f_{i,\ \text{accu}}$ is the accumulative normalized fitness of the i^{th} test case.

The normalized fitness and the accumulative normalized fitness in our example are shown in Table 11.22.

To select the test cases, a random generator is used. The test case with the next highest $f_{i,\ \text{accu}}$ is selected. The test case with the highest fitness has the highest probability of being selected, because it has the biggest range of normalized accumulated fitness ($f_{i,\ \text{accu}} - f_{i-1,\ \text{accu}}$). For example, if the numbers 0.7301, 0.1536, 0.9005 and 0.5176 are generated, test cases 3, 1, 4 and 3 (again) are selected.

Individual	P_1	f_i	$f_{i,\ norm}$	$f_{i,\ accu}$
1	41 03 90 93	13	0.2241	0.2241
2	04 72 95 31	6	0.1034	0.3275
3	44 31 45 93	31	0.5345	0.8620
4	93 45 32 91	8	0.1379	1.0
	Total fitness	58	1.0	

Table 11.22
Normalized fitness cummulative normalized fitness

11.6.3.4 Recombination

Two test cases (parents) are recombined to produce two new test cases (offspring). During this process, crossover can take place – crossover means parts of the test cases being interchanged. Crossover occurs according to a crossover probability P_c: three kinds of crossover can be implemented.

- *Single crossover* – the change takes place at one position in the test case. The random generator determines at which place the parents are split. The second parts of the parents are changed during production of offspring. Thus, the offspring is now a combination of both parents. The probability is determined by the crossover probability.
- *Double crossover* – the change takes place at two positions, dividing the parents into three parts.
- *Uniform crossover* – the number of positions where change takes place and the positions themselves are both decided by a random generator. Thus, the maximum crossover rate is equal to the number of elements of a test case.

The results of the recombination for our example is shown in Table 11.23.

									f
44	31	45	93		44	31	90	93	22
41	03	90	93		41	03	45	93	22
31	85	32	91		31	85	32	91	17
44	31	45	93		44	31	45	93	22
									83

Table 11.23
Results of recombination

11.6.3.5 Mutation

Mutation is a random change of an element within the test case. The change can cause the next value to increase or decrease, or cause a substitution with a random value. Mutation may introduce values for an element of test case not present in the start population. The optimal mutation rate is the reciprocal of the number of elements of a test case (P_m) (Sthamer, 1995).

In this example a mutation rate of $P_m = 0.25$ is applied, which means that only one of the four elements mutates. A random generator is used to determine which element is mutated, and whether the mutation is an increase or a decrease. The result is shown in Table 11.24. The mutated elements are shown in bold.

Table 11.24
The results of mutations

44 31 90 93	0.637 –	44 31 **89** 93
41 03 45 93	0.352 +	41 **04** 45 93
31 85 32 93	0.712 +	31 85 **33** 93
44 31 45 91	0.998 –	44 31 45 **90**

11.6.3.6 Reinsertion

Not all the newly formed test cases will be part of the population for the next cycle. The survival of the new test cases depends on their fitness and the reinsertion rate. The reinsertion rate (P_s) is the probability of survival of the offspring.

The fitness of the newly formed test cases in our example is shown in Table 11.25.

Table 11.25
Fitness of newly formed test cases

Individual	O_1	f_i	$f_{i,\ norm}$	$f_{i,\ accu}$
1	44 31 89 93	22	0.2651	0.2651
2	41 04 45 93	22	0.2651	0.5302
3	31 85 33 93	17	0.2048	0.7350
4	44 31 45 90	22	0.2651	1.0
	Total fitness	83	1.0	

In this example the survival rate (P_s) is 0.25 and this means that only one test case is present in the new population. For example, if the random number 0.8193 is generated, test case 4 will be inserted in the population.

The survival of the test cases in the parent population also depends on their fitness. The test cases with the least fitness have the biggest chance of dropping out of the population. To evaluate survival, the reciprocal of fitness is calculated.

In our example, the reciprocal fitness function $f_{i,\ rec}$ is defined as $1/f_{i,\ norm}$. Table 11.26 shows the calculated values for the reciprocal fitness ($f_{i,\ rec}$) of the initial set of test cases (parent population P_1). They are calculated from the fitness values, $f_{i,\ norm}$, that resulted from the selection step (see Table 11.22). These values are then normalized ($f_{i,\ recnorm}$) and accumulated ($f_{i,\ recaccu}$). This last column is used to determine randomly which test case will be deleted fom the initial set.

Table 11.26 Reciprocal fitness values (normalized and accumulative) for the parent population P_1

Individual	P_1	$f_{i,\ rec}$	$f_{i,\ recnorm}$	$f_{i,\ recaccu}$
1	41 03 90 93	4.4623	0.1919	0.1919
2	04 72 95 31	9.6712	0.4159	0.7221
3	44 31 45 93	1.8709	0.0804	0.8025
4	93 45 32 91	7.2516	0.3118	1.0
	Total fitness	23.2560	1.0	

For example, if random number 0.4764 is generated, test case 2 drops out of the population and offspring test case 4 replaces test case 2 of the parent population. The fitness of the new population, P_2, is shown in Table 11.27.

Table 11.27 Fitness of new population

Individual	P_2	f_i
1	41 03 90 93	13
2	44 31 45 90	22
3	44 31 45 93	31
4	93 45 32 91	8
	Total fitness	74

The total fitness of P_1 was 58, so the evolution has increased the fitness of the population.

After reinsertion, the fitness of the test cases is calculated. If one of the test cases meets the exit criteria, the optimum test case is found and the process stops. Otherwise the process starts its next cycle.

11.6.4 Test infrastructure

Evolutionary algorithms are only usable in combination with automated testing. The number of test cases and cycles needed for a reliable result makes it impossible to execute the test cases by hand. This means that the evolutionary algorithms must be connected to an automated test suite. This executes all the test cases on the system under test and provides the evolutionary algorithms with the output results of the system under test. Figure 11.10 shows where the connection is made between the evolutionary algorithms and the test suite.

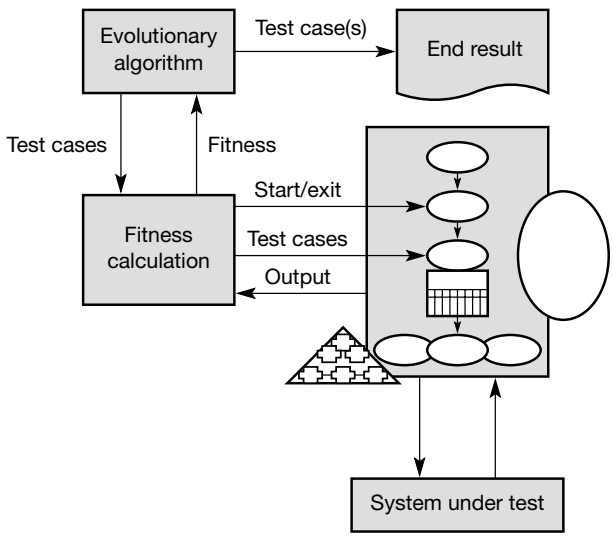


Figure 11.10 Evolutionary algorithm and test suite

Details of the automated test suite and the communication between this and the system under test are described in section 14.1. The evolutionary algorithm communicates with the test suite via the fitness calculation. This function block sends the test cases to the test suite in the correct format. It also returns the output from the system under test via the test suite. This output is used to calculate the fitness of the test case. This fitness is sent back to the evolutionary algorithm.

11.7 Statistical usage testing

11.7.1 Introduction

The objective of a real-life test is to detect defects concerning the operational usage of a system. The test should therefore simulate the real situation as much as possible. One method for describing the real usage of a system is called operational

profiles. An operational profile is a quantitative characterization of how a system will be used. Musa (1998) introduced the concept of operational profiles and, based on his work, Cretu (1997) and Woit (1994) developed some enhancements to the technique. Statistical usage testing uses operational profiles to produce a statistically relevant set of test cases.

Statistical usage testing pays most attention to the most frequently used operations focusing mainly on normal use of the system. Operational profiles are used for systems that can be described as state machines, or systems that have state-based behavior. An operational profile describes the probabilities of input in relation to the state the system is in. The preparation of test cases is based on the distribution of these probabilities.

Section 11.7.2 describes how operational profiles are prepared and how they are used for deriving test cases. Section 11.7.3 describes automated test case generation.

11.7.2 Operational profiles

Operational profiles deal with the use of a system. The user could be a human being or another system that triggers the system under test.

11.7.2.1 Operational profile analysis

To derive reliable operational profiles in a structured way it is necessary to use a top-down approach. This approach starts with identifying the different groups of customers and splitting these groups into smaller units till it is possible to describe the operational profiles. Hence, this leads to a tree structure with the system in the root with the functional profiles as the branches.

There are five steps in developing the operational profile.

1 differentiating the customers;
2 differentiating the users;
3 differentiating the system modes;
4 differentiating the functional profiles;
5 differentiating the operational profiles.

Customer profile

The customer is the one acquiring the system. The purpose of this step is to differentiate the customers into groups which have more or less the same kind of behavior and the same reasons for buying the system. The percentage this group represents out of the total group of customers is then calculated. Another option is to assess the relative importance of the different groups. This makes it possible to differentiate the test effort with respect to relative importance rather than the relative group size. The size of the group is not necessarily related to the importance of the group.

User profile

A user is a person or a (sub)system that uses the system. A user profile is the description of the usage of the system and the proportion of the customer group's usage it represents. Instead of the proportion, it is possible to make an estimation of the relative importance of this profile.

System mode profile

A system mode is a set of functions that belong to each other.

Functional profile

Each system mode is divided into functions. A list of functions is prepared and the probability of occurrence is determined for each. Functions should be defined so that each represents a substantially different task.

Operational profile

For every functional profile, an operational profile is created. The operational profile describes the usage of the functional profile.

11.7.2.2 Operational profile description

An operational profile describes quantitatively how the system is used by a certain type of user. It does this by determining the answer to the question: "When the system is in *this* state, what is the chance that *that* is the next event that will be triggered by the user?" In concrete terms, the operational profile specifies the probability for each combination of "state" and "next event." Woit (1994) suggested the use of "system history" instead of "state" since the user's behavior not only depends on the current state of the system, but also depend on what may have happened before. It allows more flexibility in describing the realistic usage of the system, but can still be applied to describe the "original operational profile." This chapter uses Woit's suggestion about system history.

The following steps must be executed to determine the operational profiles (for each functional profile).

1 *Determine the possible states of a system*. If, for some reason, the history of the system is important, the sequence of states has to be determined. This results in a list of states and histories. Differentiation into newly-defined histories is only useful if those histories have different probabilities defined for the events. States with no related events can be left out of the list.
2 *Determine the list of possible events*.
3 *Determine the probability of every combination*.

The combined information can be presented in a matrix (see Table 11.28).

	Event 1	Event 2	...	Event *m*
History class 1	$P_{1,1}$	$P_{1,2}$	...	P_{1},m
History class 2	$P_{2,1}$	$P_{2,2}$	...	P_{2},m
...	...	...	...	...
History class *n*	$Pn,_{1}$	$Pn,_{2}$	...	Pn,m

Table 11.28
Notation technique for operational profiles

11.7.2.3 Operational profile specification

To show how the information now available is translated into operational profile specification, a simplified example of an operational profile for a VCR is used. The VCR has six history classes. The last of these is an example of how the history of the system can influence the operational profile. If the tape is currently at more than 75 percent of its length, the probabilities of the events are dramatically changed. A VCR uses a counter to show the position of the tape. For this tape, the maximum position is 20 000. So, if the position of the tape is bigger than 15 000, the probabilities of events are changing (see Table 11.29). All the events except evStop have one variable. The variable is a member of *X*, where *X* is in the range 50, 100, ..., 20 000.

	evStop	evRewind <a>::*X*	evPlay <b>::*X*	evFastforward <c>::*X*	evRecord <d>::*X*
Standby	0	0.2	0.5	0.2	0.1
Rewind	0.3	0	0.6	0.1	0
Play (counter < 15 000)	0.5	0.3	0	0.2	0
Fast forward	0.5	0.1	0.4	0	0
Record	1	0	0	0	0
Play (counter=>15 000)	0.8	0.15	0	0.05	0

Table 11.29
Example of an operational profile for a VCR. All possible events are shown in the columns. The rows give the history classes. For every row, the total probability is one

The operational profile must be translated into a specification. For test case generation purposes, a higher programming language such as C, C++, or Java is used.

The history class class is described with transition functions. For every combination of a history class and an event with a probability greater than zero, a transition function has to be defined. So, the total number of transition functions is less than or equal to the number of history classes multiplied by the number of possible events. A transition function is a description of the event and the resulting state.

Every history class gets an identifier starting at the top, with 1, to the end of the list, *n*. Every event gets also an identifier starting at the left, with 1, to the

end of the row, *m*.

Every event can have one or more variables belonging to a certain set. This set is part of the operational profile specification. The value of the variable can be chosen randomly if the variable is not influenced by the history of the system. In this example, it is not possible to forward the tape by 15 000 if, for instance, the tape is set at position 10 000. This means that the value of the variable is subject to some conditions. These conditions are translated to a condition function (see Figure 11.11).

Figure 11.11 Pseudocode for the operational profile of the VCR

```
init:  Rewind till tape at start position
       counter = 0
       max = 20000

F1,2:  if counter = 0
              return 1
       else
              counter = RewindCondition(counter) + counter
              return 2


F1,3:  if counter = max
              return 1
       else
              counter = PlayCondition(counter) + counter
              if counter => 15000
                     return 6
              else
                     return 3


F1,4:  if counter = max
              return 1
       else
              counter = FastforwardCondition(counter) + counter
              return 4


F1,5:  if counter = max
              return 1
       else
              counter = RecordCondition(counter) + counter
              return 5
```

```
F2,1:   return 1

etc.


Set:   X={50, 100,…, 20000}

RewindCondition (in: counter)
      initialize random generator
      rand = random generator
      rank = (1 + rand div 0.0025)* 50
      while rank > counter
             rand = random generator
             rank = (1+ rand div 0.0025)* 50
      end while
      return -rank

PlayCondition (in: counter)
      return Cond(counter)

FastforwardCondition (in: counter)
      return Cond(counter)

RecordCondition (in: counter)
      return Cond(counter)

Cond (in: counter)
      initialize random generator
      rand = random generator
      rank = (1 + rand div 0.0025)* 50
      while rank + counter => max
             rand = random generator
             rank = (1+ rand div 0.0025)* 50
      end while
     return rank
```

Note that X in steps of 50 means 20 000/50 = 400 possible different values. Now, 1/400 = 0.0025. So, for the random generator every step of 0.0025 means the next member of X.

This specification contains enough information for automated test case generation.

The test case generator starts in history class 1 (Standby). A random generator selects a column and the transition function, which is correlated to this combination, is executed. Now, the test case generator knows the resulting state and again selects a column. This process goes on till the stop criteria are met.

One sequence is one test case. The test cases are stored in a file. During this process the selected combinations are recorded and the frequencies of the selections are counted.

11.7.3 Test case generation

The specification of the operational profile is written in a programming language in order to make it possible to use test case generation. The test case generator produces a sequence of states and events starting with the initial state. The sequence is produced based on the probabilities in the operational profile. Figure 11.12 shows an example of a test case generator written in pseudocode.

Figure 11.12 Pseudocode for a test case generator

```
P(1… n, 1… m) = probability matrix operational profile cumulated
per row
CaseLength = length of test case
AmountTestCase = number of test cases needed
F(i… n, 1… m) = frequency of combination in test cases
TestCase(1… AmountTestCase, 1… CaseLength)
rand = random number

Initialize random generator

amount = 0

while amount < AmountTestCase
      Init ()
      amount ++
      length = 0
      hisstat = 1
      while length < CaseLength
            rand = random generator
            Event = SelectEvent(hisstat, rand)
            F(hisstat, event) ++
            hisstat = ExecuteFunction(hisstat, Event)
            TestCase(amount, length) = Event & ", " & hisstat
            length ++
      end while
end while
TestCase to file
F to file
```

F is only necessary if a reliability estimation is made.

Now the test cases are available and can be executed. For large systems this means that many test cases are generated. It would be nice if these test cases could be executed automatically.

It is possible to do a reliability estimation (Woit, 1994). For this the frequency of the transitions in the test cases must be known. Woit described a test case generator in C including reliability estimation.

11.8 Rare event testing

11.8.1 Introduction

Statistical usage testing "ignores" events that happen very infrequently. It is often very important that these events are handled correctly by the system under test. These events, called *rare events*, are often related to safety aspects or to the main functionality of the system. Rare events do not deal with system failures.

Rare event testing is similar to statistical usage testing executed close to the release date. If the main functionality of the system is to deal with these rare events (e.g. an ABS braking system for braking under slippery conditions) this rare event handling has to be tested much earlier on in the process.

11.8.2 Specifying rare event test cases

The first steps of specifying rare event test cases are similar to those for statistical usage testing.

Customer profile

The customer in this case is the one purchasing the system. The purpose of this step is to identify groups of customers who purchase the system for different reasons or for different usage. It only makes sense to identify these different groups if this influences rare events or rare event frequencies.

User profile

A user is a person or a (sub)system that uses the system. The user can potentially trigger rare events.

System mode profile

Identifies the sets of functions that belong to each other.

Functional profile

Each system mode is divided into functions. It is not useful to break down a system mode profile further if it is already known that there are no rare events related to it.

Operational profile

For every functional profile, an operational profile is created. This is only done for a functional profile where rare events can occur. An operational profile is a matrix of histories of the systems and events. Every combination of a history and an event has a probability. Those combinations with a very small proba-

bility are of real importance here. The operational profile is the basis for the specification of the rare event test cases.

If the operational profiles are already specified for the statistical usage test, then it is possible to reuse these profiles. Be aware that in those operational profiles the rare events may have been omitted (given a probability of zero) because they happen too rarely for statistical usage testing. It is best to re-evaluate the events that received a probability of zero.

All the history–event combinations with a very low probability are selected. Every combination is a test case. A test script for such a test case contains the following issues.

- *Precondition*. How to get the system into the right state. The history state defines what state the system must be in. Information as to how to get the system in this state is also available in the operational profile. The precondition is sometimes a list of state–event combinations needed to reach the state necessary for this test case. Sometimes one has to take additional measures in the environment of the system in order to be able to trigger the rare event.
- *Description of the rare event*. There must be a description of how the rare event can be triggered.
- *Result state*. This is a description of the result state of the system. It is sometimes necessary to describe how this state can be checked.

11.8.3 Thoroughness

Every test case means a combination of a history state and a rare event. To increase the thoroughness of the test, a cascade of rare events can be tested. This is only possible if a rare event leads to a history state of the system where another rare event is possible.

11.9 Mutation analysis

11.9.1 Introduction

With mutation analysis, faults caused on purpose have to be detected. *Fault seeding* means introducing faults into a system under test. The testers of the system are not aware of just which faults these are. After execution of the test set, the fault detection rate is measured. This rate is equal to the number of faults found divided by the number faults introduced.

Mutation analysis at module or class level is done in a quite different manner. The faults, called mutations, are introduced in a line of code – each mutation is syntactically correct. Only one mutation is introduced per module or

class. Once a mutation is introduced, the program is compiled – this program is called a mutant. This mutant is tested with the test set, and after a set of mutants has been tested the fault detection rate is measured. The rate is equal to the number of mutants found divided by the total number of mutants introduced.

One of the objectives of testing is to gain confidence in the system under test. This is achieved by executing the system using test cases and checking the output result. Testing involves making choices, not only about what is important but also about which test design technique to use. If an informal test design technique is chosen, a decision has to be made as to which test cases to prepare. All these decisions are not taken in a formal way. Confidence based on a set of test cases derived in this way can be misleading or misplaced.

Confidence in the quality of the test set can be increased through mutation analysis. With this method the fault detection rate of the test set can be measured. This can be done at system level – it is then called fault seeding – or it can be done at module or class level.

The basic concept of this method is that the fault detection rate of the test set is an indicator of the quality of the test. A test set with a high fault detection rate can give a reliable measure of the quality of the system under test, or a good indication of the number of undetected faults in the system. Hence, it can be indicative of the confidence one can have in the system.

The benefits and usefulness of this method are open to debate. The method makes no direct contribution to the quality of the system under test and merely gives an indication of the quality of the *test set*. If formal test design techniques are used, it is questionable if it makes any sense to use mutation analysis.

11.9.2 Fault seeding

Fault seeding is used for testing at system level. The following steps are performed.

1 *Introducing faults*. Several faults are introduced into the system under test. The distribution of these faults is related to the risk analysis of the test strategy. The faults have to be introduced by someone with a good knowledge of the system under test. There is always the possibility that a fault can hide another fault from detection – this must be avoided.
2 *Testing*. The system released for test includes the introduced faults. The test team has to test the system according to their test designs. The team does not know which faults are introduced. The complete test set is executed.
3 *Analyzing*. The introduced faults are separated from the defect list and are counted. Dividing the number of detected faults by the number of introduced faults gives the fault detection rate.
4 *Removing faults*. Before release for production, the faults have to be deleted from the source code.

11.9.3 Mutation analysis at module level

Mutation analysis is carried out at module or class level. The determination of test set adequacy starts as soon as the module or class is functioning correctly according to the test set. The following steps are used in carrying out mutation analysis.

1 *Production of mutants*. The module or class is changed in one line of code and forms a mutant of the original module or class. The mutants are produced using rules called *mutation operators*. These are program language dependent because each language has its own fault categories. For instance, in the C program language there is arithmetic operator replacement, binary operator replacement, and statement deletion (Untch, 1995); and for Java there is the compatible reference type operator, constructor operator, and overriding method operator (Kim *et al*., 1999). All the mutants have to be syntactically correct. The introduction of a mutation can force, for instance, a division by zero, deadlocks or endless loops. The division by zero eliminates itself; having a limit for execution time can eliminate the other two. If the execution time of the mutant exceeds a certain limit, the mutant will be identified as an erroneous program.
2 *Testing mutants*. Mutants are tested with the original test set. The test stops if the mutation is detected or if the complete test set is used.
3 *Analyzing*. It is always possible that a mutant has the same behavior as the original module or class. These mutants are described as "equivalent" and are not detectable. The fault detection rate can be calculated by the number of detected mutants divided by the total number of mutants minus the number of equivalent mutants.

Rigorous application of this method leads to an enormous number of mutants that all have to be executed using the test cases. Untch identified, for just one equation, 71 possible mutations (Untch, 1995). The only way to deal with this quantity of mutants is to use automated test tools and tools which are able to introduce mutations automatically. The MOTHRA software testing environment has been developed (Hsu *et al*., 1992) for this purpose. Another approach for producing and testing mutants is described by Untch (1995). He suggested an approach in which every line of code is rewritten into a line in which the mutant operators are included. The tool he used can read this code and translate it into program code with the mutation in it.

12 Checklists

12.1 Introduction

Checklists are used as a technique to give status information in a formalized manner about all aspects of the test process. This chapter contains the following categories of checklists:

- checklists for quality characteristics;
- general checklist for high-level testing;
- general checklist for low-level testing;
- test design techniques checklist;
- checklists concerning the test process.

The checklists for quality characteristics are used to assess if the system fulfills the requirements concerning the quality characteristics. If no is the answer to one of the checklist questions, a defect is raised. For the test design techniques checklists, a detected omission will lead to an action to get the missing information or the decision to skip that design technique and use another.

The other checklists are used to support the test manager. If one of these questions is answered with no, action must be taken to correct the omission or measures taken to live with it. These types of checklists are also used as an aide-mémoire for test managers, and also as a communication instrument with stakeholders in mind.

12.2 Checklists for quality characteristics

This chapter is about checklists for quality characteristics, applicable to static testing in particular. In addition to these quality characteristics, user-friendliness is mentioned. These quality characteristics can be tested through a combination of dynamic testing and questionnaires. The checklists can be used as a basis for preparing custom-made checklists applicable to the specific needs of the organization, project, and/or product. The checklists concern the following quality characteristics:

- connectivity;
- reliability;
- flexibility;
- maintainability;
- portability;
- reusability;
- security;
- testability;
- usability.

12.2.1 Connectivity

Connectivity is the capability of the system to establish connection with different systems, or within this (distributed) system.

Checklist

- Has a general reference model been used?
- Is there a description of which measurements are made to prevent faulty input and actions?
- Is there a description of how fault recovery is implemented?
- Has the interaction between systems been described unambiguously?
- Has the connectivity been designed according to (inter)national or industry standards?
- Has a standard machine interface been used?
- Is it possible to secure additional critical or essential functions?
- Have the systems to connect to been described?
- Has a generalized interface been specified and used?
- Have standards been set for the interaction of data?
- Have standards been set for the connection between hardware and infrastructure components?

12.2.2 Reliability

Reliability is the capability of the system to maintain a specified level of performance when used under specified conditions.

The quality attribute "reliability" is divided into three subattributes – maturity, fault tolerance, and recoverability.

12.2.2.1 Maturity

Maturity is the capability of the system to avoid failure as a result of faults in the system.

Checklist

- Have (inter)national or industry standards been used?
- Have input, output, and processing been implemented separately?

- Have checkpoints and reboot utilities been used?
- Has dual processing been implemented?
- Has data processing been split into subtransactions?
- Have watchdogs been implemented?
- Is it a distributed system?
- Have program modules been reused?
- Is the software parameterized?
- Have algorithms been optimized?

12.2.2.2 Fault tolerance

Fault tolerance is the capability of the system to maintain a specified level of performance despite system faults or infringements of its specified interface.

Checklist

- Have watchdogs been implemented?
- Has dual processing been implemented?
- Have consistency checks been implemented?
- Have reboot utilities been implemented?
- Has a fail-safe condition been implemented?
- Have preconditional invariants been explicitly verified during system execution?
- Has fault identification been implemented?
- Is the core functionality stored in different modules?
- Is it a distributed system?

12.2.2.3 Recoverability

Recoverability is the capability of the system to re-establish a specified level of performance and recover the data directly affected in the case of a failure.

Checklist

- Have (inter)national or industry standards been used?
- Is a recovery system implemented?
- Has a built-in self-test been implemented?
- Have periodically fast-reboot facilities been implemented?
- Have watchdogs been implemented?
- Has dual processing been implemented?
- Have reboot utilities been implemented?
- Have consistency checks been implemented?
- Is it a distributed system?
- Has exception-handling been implemented?

12.2.3 Flexibility

Flexibility is the capability of the system to enable the user to introduce extensions or modifications without changing the system itself.

Checklist

- Is the software parameterized?
- Have logical instead of hard coded values been used?
- Has data processing been separated from data retrieval?
- Is it possible to modify input functions in order to change the function of the system?
- Is it possible to modify control functions in order to change the function of the system?
- Is it possible to modify processing functions in order to change the function of the system?
- Is it possible to modify output functions in order to change the function of the system?
- Is it possible to change the user dialog?
- Have expansion slots been implemented?

12.2.4 Maintainability

Maintainability is the capacity of the system to be modified. Modifications may include corrections, improvements, or adaptation of the system to changes in the environment, and also in requirements and functional requirements.

Checklist

- Have (inter)national or industry standards been used?
- Have development standards been used?
- Have watchdogs been used?
- Is data processing split up into subtransactions?
- Have input, output, and processing been implemented separately?
- Has essential functionality been stored in separate modules?
- Has the system breakdown structure been described?
- Is open, consistent, and up to date technical documentation available?
- Is the software parameterized?
- Is it a distributed system?
- Is it a real-time system?
- Have algorithms been optimized?

12.2.5 Portability

Portability is the capability of the software to be transferred from one environment to another.

Checklist

- Is the software parameterized?
- Have (inter)national or industry standards been used?
- Has a standard machine interface been used?
- Have the machine dependencies been implemented in separate modules?
- Have the algorithms been optimized?
- Have the subsystems been distributed?

12.2.6 Reusability

Reusability is the capability of the software to enable the reuse of parts of the system or design for the development of different systems.

Checklist

- Have (inter)national or industry standards been used?
- Is the software parameterized?
- Has the data processing been split up into subtransactions?
- Have the input, output, and processing been implemented separately?
- Have the machine dependencies been implemented in separate modules?
- Has a standard machine interface been used?
- Have program modules been reused?
- Have algorithms been optimized?

12.2.7 Security

Security is the capability of the system to ensure that only authorized users (or systems) have access to the functionality.

- Has functionality for identification, authentication, authorization, logging, and reporting been implemented?
- Is secured functionality physically separated from the remainder of the system?

12.2.8 Testability

Testability is the capability of the system to be validated.

Checklist

- Has the testware been completed and preserved for the benefit of future tests?
- Have tools for planning and defect management been used?
- Is open, consistent, and up to date functional documentation available?
- Have development standards been used?
- Have watchdogs been used?
- Have the input, output, and processing been implemented separately?
- Have machine dependencies been implemented in separate modules?
- Has the essential functionality been stored in separate modules?
- Have software modules been reused?

- Has the system breakdown structure been described?
- Have the subsystems been distributed?
- Has multiple processing been implemented?
- Have the algorithms been optimized?

12.2.9 Usability

Usability is the capability of the system to be attractive to the user under specified conditions, and to be understood, learned, and used.

Usability is a rather subjective quality characteristic – just what usability means depends on the individual taste of a user. However, it is usually possible to identify different groups of users with their own demands. Another factor often ignored is the culture that influences the acceptance of the user interface. The only way to get a clear picture of the usability of a system is to let a user implement use the system as if it were in daily use. If it is thought that culture could be an important factor, the forum used should have a diverse cultural background, or otherwise use more forums.

To give a clear picture of the usability of a system, a questionnaire can be filled in by the members of the user forum. Possible items in the questionnaire include the following.

- What is the general opinion of the user interface?
- Are there any negative remarks about it?
- Are all the terms and messages used clear and unambiguous?
- Is help available and doesn't make the task executable?
- Is the same standard used throughout the user interface?
- Are all the tasks easy to access?
- Is it easy to navigate through the interface with the user control instrument?

In addition to the questionnaire some measurements can be taken during the use of the system. If there measurements relate to some of the defined requirements it will be possible to give a more concrete idea of the usability. The following measurements could be made.

- How long does it take for a new user to perform a certain task?
- How many mistakes does a new user make during a certain task?
- How many times must a new user consult the documentation to complete the task?
- How many tasks does an experienced user complete during a certain time period?
- What is the ratio of failed and successful tasks?
- How many commands are used on average to complete tasks?
- Does the number of commands exceed a certain desired value?
- What is the change in the ratio of failures to successes over time for a new user (how long does it takes to become an experienced user)?
- How many times does the user lose control of the system?
- Does the execution time for a certain task exceed a certain desired value?

12.3 General checklist for high-level testing

12.3.1 General

- Have the relationships between the various documents been described sufficiently and clearly?
- Are all required documents available?
- Has a division of the system into subsystems been described?
- Have all interfaces been described?
- Have all functions and subfunctions been described?
- Has the dialog design been described?
- Has error handling been described?
- Have the input and output been described for each function?

12.3.2 Division into subsystems and interfaces

- Does the description of the selected subsystems include a summary description of the functions to be carried out, the required processes, and the data?
- Has the implementation sequence been indicated?
- Have the locations of the interfaces been indicated?
- Is there a description of the interface?
- Is there a description of the physical construction?

12.3.3 Function structure

- Have all functions been displayed, for example in the data flow diagrams?
- Has a short description of each function been given?
- Have the conditions that apply to execution been listed, including their frequency?
- Have the security requirements been described?
- Have the measures taken within the framework of correctness and completeness of data been described – for example:
  - validations of input;
  - redundant input;
  - duplicate input;
  - programmed checks on the results of data processing?
- Have the performance requirements been described?
- Has the relative importance of the function in relation to other functions been described?
- Is there a short description of data flows?
- Have the functions and the input and output flows been cross-referenced?
- Have all sources of input been listed and described?
- Have all destinations of output been listed and described?
- Have all basic functions been included in a data flow diagram?
- Has processing been described?

12.3.4 User dialog

- Is there an overview of the relationships between functions and user dialog?
- Has the structure of the user dialog been described?
- Does the user dialog comply with the applicable guidelines?
- Has the user dialog been described?
- Has the user dialog output been described?
- Has the use of the user input device been described?
- Is there a description of the user dialog layout?

12.3.5 Quality requirements

- Have the quality requirements been specified?
- Are the quality requirements quantifiable and measurable?
- Can the quality requirements be used as acceptance criteria?

12.4 General checklist for low-level testing

12.4.1 General

- Have the relationships between the various documents been described sufficiently and clearly?
- Are all required documents available?
- Has the system structure been described?
- Do program descriptions exist?

12.4.2 System structure

- Has the system structure of the subsystem been displayed, on the basis of which a program division can be made?

12.4.3 Program division

- Have all programs been displayed in the form of a system flow?
- Have the following been described for all programs within the system flow:
  - name of the program;
  - input and output;
  - processing.

12.4.4 Program description

Have the following aspects been described for the various program descriptions:

- the identification and the aim of the program;
- the description of the function;
- the description of the input and output flows;
- the description of the links to other programs and/or (sub)systems;
- the description of the buffers;
- the description of the key data;

- the description of the parameters (including checks);
- the description of logging;
- the schematic representation of the program;
- the description of the program that belongs to the diagram, containing the sections and paragraphs that are called, and the standard routines that have been used?

12.4.5 Performance requirements

- Have the performance requirements been specified?
- Are the performance requirements measurable?

12.5 Test design techniques checklist

12.5.1 Control flow test

The desired test basis is a flow chart, or a decision complemented by a description. The following checklist can be used to check the documentation.

- Have the algorithms been described?
- Is the triggering event indicated clearly?
- Is it clearly indicated which data are being used, and what the source is?
- Is it possible to make an output prediction?
- Have the various decision points been described, including the accompanying conditions?

12.5.2 Classification-tree method

The following checklist can be used to check the specifications.

- Has the method of processing the functions been described, including the input, output, and decision paths?
- Have the triggering events of the process been indicated clearly?
- Have the input domains been described?
- Is it possible to make an output prediction?

12.5.3 Elementary comparison test

The following checklist can be used to check the specifications.

- Has the processing method been described such that the various processing paths can be recognized?
- Have the conditions which determine the various processing paths been identified clearly?
- Has processing been described unambiguously, including input and output?
- Has processing been described for each input item?
- Is it possible to make an output prediction?

12.5.4 Statistical usage testing and rare event testing

The following checklist can be used to check the specifications.

- Has the expected frequency of use been described at the function level?
- Has it been described which functions may be carried out by each type of user?
- Have different customer groups been identified?
- Has it been described when each function will be used?

12.5.5 State-based testing technique

The following checklist can be used to check the specifications.

- Has a state transition diagram been given?
- Have all transactions/functions been distinguished?
- Are descriptions consistent with the state transition diagram?
- Are states uniquely identifiable?
- Is it described how the events can be triggered?
- Have pre- and post-conditions of the functions been described?

12.5.6 Evolutionary algorithms

The following checklist can be used to check the specifications.

- Have timing requirements been described?
- Is it possible to prepare a fitness function?

12.5.7 Safety analysis

Safety analysis is a process of iteration in which a lot of specific knowledge is used. When the first safety analysis is carried out, at least a global breakdown structure of the system should be available. In addition to design specifications, requirements from certification institutes and legal obligations provide the main input for safety analysis checklists.

12.6 Checklists concerning the test process

The checklists in this section provide guidelines for a diversity of activities concerning the test process. They are not exhaustive but can serve as a basis for constructing organization-dependent and project-dependent checklists. The following checklists are included.

- *Test project evaluation*. This checklist contains attributes for intermediate and final evaluations of a project.
- *Global investigation of the system*. This checklist can be used to support the global review and study activity from the planning and control phase.

- *Preconditions and assumptions*. This checklist contains examples of preconditions and assumptions to include in a test plan.
- *Test project risks*. The risks regarding the test project must be made explicit. This checklist contains a number of potential risks.
- *Structuring the test process*. This checklist is used to assess the current situation of testing.
- *Test facilities*. This checklist is useful for organizing and establishing the test infrastructure, the test organization, and test control during the planning and control phase.
- *Production release*. This is a checklist for the determination of completeness of the product, test activities, production parameters, and production operations.
- *Coding rules*. This checklist applies to the low-level testing activities concerning analysis of the code.

12.6.1 Test project evaluation

This checklist can be helpful in preventing problems during a test project.

- Has the production release checklist been completed?
- Is the test project running under time pressure? Has this led to adjustments in the test strategy?
- Have customers been involved in early stages of the test project?
- Have concessions been made during the test project with respect to the planned test volume and intensity?
- Have experienced (external) advisors and test specialists supported the test team?
- Has there been sufficient subject knowledge available in the test team?
- Has defect handling been organized properly and does the version control function adequately?
- Has the prescribed test method, as established in the test handbook, provided enough information and support in the case of any problems?
- Has there been a check as to whether guidelines are being followed?
- Was there a procedure for possible deviations from these guidelines?
- Has a test plan been created according to the regulations?
- Was there a critical end date by which testing must be finished?
- Has the test plan been specified in sufficient detail and/or maintained during the execution of the test?
- Have the requested supporting aspects (tools, appliances, etc.) been made available in sufficient quantities on time?
- Was it possible to complete the intended test plan within the limits set?
- Was the test plan clear and could it be executed by the members of the test team?
- Have scheduling techniques and tools been applied?
- Have the test activities and test progress been monitored adequately?
- Have the reasons for adjustment of the schedule been clearly documented?

- Has it been possible to indicate clearly if the activities are on schedule?
- Has there always been a good fit between the test plan and the overall project plan?
- Have the standard techniques and tools been sufficient?
- Has there been sufficient representation in the test team by:
  - system designers;
  - system administrators;
  - system management staff;
  - domain experts;
  - customers;
  - accounting department staff?
- Has management personnel been sufficiently involved in the testing process?
- Was there a proper and unequivocal division of tasks?
- Has version control of system components been implemented?
- Has there been a need for any special training with regard to hardware, software, and/or testing?
- Were the members of the test team universally deployable?
- Has the test lifecycle been used as prescribed?
- Were all system functions completed during testing?
- Were test cases other than those specified carried out?
- Have customers assessed the quality of the test method used by the development team?
- Has the development team provided complex test cases to the acceptance testers?
- Has error guessing been used?
- Have utilities been tested?
- Have provisions been made for a regression test at the end of the test?
- Have automated tools been used for the creation of test cases?
- Have all scheduled test scripts been carried out?
- Have all available checklists been used?
- Have the consultation structures been successful – both those within the test team and those with the developers?
- Has there been a sufficient amount of documentation discipline?
- Has the testware been systematically completed?
- Have the causes of defects in the test approach, scheduling, and techniques always been properly documented?
- Was the budget sufficient?
- Has there been enough time for test and consultation activities?
- Is the office environment as it should be?
- Have sufficient test tools, terminals, printers, etc. been made available?
- Has there been enough personnel?
- Have the test results been made available on time?
- Have checks been carried out on the execution of the tests (random checks)?

- Did the content of each test document accord with the agreements?
- Was the content of each test document complete?
- Has the test documentation been made accessible to users?

12.6.2 Global investigation of the system

During the global review and study in the planning phase, information is gathered about the test project and the system to be tested. This helps to clarify what is expected of the test team, and what the team expects of the organization.

Project information

- commissioner;
- contractor;
- objective (assignment, strategic importance);
- agreements (contracts);
- project organization, including personnel;
- reporting lines;
- activities and time schedules;
- financial planning;
- preconditions set for the test to be carried out;
- risks and measures.

System information

- standards and guidelines (development methods, etc.);
- requirements study (including description of existing situation);
- documentation from the basic design phase;
- development infrastructure (hardware, tools, and systems software);
- testware already present:
  - test design;
  - test scripts;
  - test scenarios;
  - test infrastructure;
  - test tools;
  - metrics;
  - evaluation report of previous tests.

12.6.3 Preconditions and assumptions

Preconditions are defined by the organization outside the test project, and also within the test project by the test team. Preconditions set outside the organization are, more or less, boundary conditions within which the test team has to work. In general, these concern limits and conditions with regard to resources, staff, budget, and time. The test team also defines preconditions that have to be met by the rest of the organization in order that they can succeed in their assignment.

- *Fixed deadline*. The test should be completed before the fixed deadline.
- *Project plan*. For all the test activities, the current project plan leads the way.
- *Delivery of test units*. The software must be delivered by the development team in functionally usable and testable units. When applicable, the user manuals must be available. Every functional unit must be subjected to a unit and system test.
- *Insight and changes of development plans*. The test team should be made aware of the delivery schedule of the development team. Changes in this must be communicated to the test team.
- *Participation in development schedule*. The test team must have the opportunity to influence decisions about the delivery sequence. The delivery sequence has an impact on the ease and speed of testing.
- *Quality of the system test*. The development team performs the system test on the designated quality characteristics and objects.
- *Insight into the system test*. The system test delivers the following to the acceptance test team:
  - scheduling;
  - test strategy;
  - test cases;
  - test results.
- *Scope of test basis*. The test team has access to all system documentation.
- *Changes in test basis*. The test team must be informed immediately about changes in the test basis.
- *Quality of test basis*. If the quality of the test basis is not sufficient, or essential information is missing, this must be reported immediately so that appropriate measures can be taken.
- *Availability of test team*. The test team must be available according to the test schedule. The members should have the appropriate knowledge and experience.
- *Support of development team*. Members of the development team must be available to correct any defects blocking the progress of test.
- *Definition and maintenance of the test environment*. The defined test environment must be available on time and maintenance support must be available during test execution.
- *Availability of test environment*. The test team is the owner of the test environment during test execution. Nothing is changed or used without the permission of the test team management.

12.6.4 Test project risks

The project risks are identified and described in the test plan. For each risk, the consequence and the solution, and/or measures to minimize the risk or consequences, are mentioned.

- The lack of a detailed schedule for the development team for the various subsystems.
- The previously determined deadline can influence the execution of the test activities as a whole, as mentioned in the test plan.
- The test basis not being available on time.
- Insufficient quality of the test basis.
- The availability of test personnel (capacity, suitability in terms of test experience and expertise).
- The availability and controllability of the desired test environment.

12.6.5 Structuring the test process

If structured testing is introduced in an organization, the present status of testing should be determined first. This requires a brief investigation called a *test inventory*.

Test lifecycle

- Which test levels are distinguished and what is the scope of each?
- Is there some co-ordination between these test levels?
- Does each test level have its own lifecycle?

Planning and control phase

- Have test plans been created for each test level?
- Have these detailed test plans been based on the master test plan?
- Does the test plan contain a description of the assignment, defining the responsibilities, scope, assumptions, and preconditions?
- Has the test basis been described in the test plan?
- Has a test strategy been determined?
- Has the test infrastructure been defined in the test plan and have agreements been made concerning control?
- Has a procedure for reporting defects been described in the test plan?
- Does the test plan describe what the test deliverables are?
- Does the test plan describe the parts of the testware and how control of the testware has been arranged?
- Does the test plan describe which (test) functions are distinguished, including the separation of tasks, responsibilities, and authorities?
- Does the test plan describe the different reporting lines for the test team?
- Does the test plan include a schedule of activities, and a financial and personnel plan?
- Has a well-founded budget been set up for testing?

Preparation phase

- Has a testability review of the test basis been carried out for each test level?
- Has a well-considered selection been made of the test techniques to be used?

Specification phase
- Have test designs been prepared?
- Have test design techniques been used for the preparation of test designs?

Execution phase
- Has defect management been formalized?
- Are there re-tests?

Completion phase
- Has the testware been stored for the benefit of maintenance tests?
- Does a formal hand-over/release take place?

Test techniques
- Have techniques been used to determine test strategies?
- Have test budgeting techniques been used?
- Have test design techniques been used?

Project organization
- How was the test project organized?
- How were authorities and responsibilities distributed?
- Who carries out the tests?
- Have agreements been made regarding reporting?
- Have any procedures been set up for defect control?
- How do reviews and handover occur?
- Who is authorized to release a tested system?

Line organization
- Have standards been used for testing?
- Has the use of the standards been checked?
- Has any historical data been collected for the purpose of metrics regarding testing?
- Has any domain expertise been available for testing?
- Has testing been co-ordinated across all test projects?
- Has there been training in the field of testing?
- What is the level of knowledge of the employees in the field of testing?

Infrastructure and test tools
- Has there been a separate test environment for testing? Who is the owner of this test environment?
- Has the test environment been comparable to the production environment?
- Have there been any procedures regarding version and configuration control for the test environment?
- Has the organization of and access to the test environment been separate from the development?

- Have there been any test tools available for:
  - test management (planning, time registration, progress monitoring);
  - test design;
  - defect management;
  - test budget;
  - test execution;
  - file management.

12.6.6 Test facilities

This section contains a list of attributes important in organizing the test infrastructure, the test organization, and test control during the planning and control phase.

Workspace

- rooms;
- meeting area;
- furniture (chairs, desks, tables, cabinets);
- copy facilities;
- office supplies;
- forms;
- diskettes.

Hardware

- PCs and laptops;
- input and output media such as printers, displays, PIN/card readers;
- communication lines;
- storage space (tapes, disk space);
- communication servers and network administration hardware;
- servers and collectors;
- interfaces, switches, modems, converters, adapters;
- connections to public networks;
- storage media such as tapes, cassettes, disks, and diskettes;
- power supply, cooling, cabling;
- simulators.

Software

- operating systems;
- communication software;
- word processing package;
- planning and progress monitoring package;
- test tools;
- logging;
- authorization and security;
- accounting and statistics;

- assemblers and compilers;
- workbenches;
- prototyping and simulation.

Training

- introductory course on testing;
- course on test techniques;
- elementary understanding of functional design;
- elementary understanding of technical design.

Logistics

- coffee and tea facilities;
- lunch facilities;
- rules for expenses claims;
- overtime regulations;
- access security outside office hours.

Staff

- test team;
- system control;
- users;
- production supervision;
- management.

Procedures

- production control;
- delivery from development team to test team;
- version control of the test object, the test environment, and the testware;
- defect procedure;
- test instructions;
- schedule monitoring;
- test consultation;
- configuration management;
- defect reporting and handling, restart;
- authorization and security.

Documentation

- regulations, or even a handbook, for testing;
- user documentation.

12.6.7 Production release

Before a recommendation for release is given, the following release checklist should be completed. For all checks in the list, a "pass" can only be awarded after all the tests have been completed.

Completeness of deliverables

- Is a regression test available for maintenance?
- Is the necessary help, emergency, and correction software available?
- Is the required system documentation, user documentation, and production documentation available, complete, accessible, and consistent?

Completeness of test activities

- Have the interfaces between the various software modules been tested and approved?
- Have the interfaces between the various subsystems been tested and approved?
- Have the interfaces with other systems been tested and approved?
- Have the interfaces with decentralized applications been tested and approved?
- Have the interfaces with decentralized devices been tested and approved?
- Have the interfaces with the operating systems been tested and approved?
- Have the interfaces with the manual procedures been tested and approved?
- Is the testware completed?
- Has the testware passed the required quality check?
- Are the required performance demands met?
- Is there a checkpoint recovery and does it work?
- Have all restart options been tested successfully?
- Does file access occur within the applicable (security) demands?
- Does the program also work with empty input files?
- Does the system respond adequately to invalid input?
- Have the various forms been tried out at least once?

Production preparation

- Are the defects that still exist at the time of going into production known?
- Have measures been taken to handle the remaining defects?

Production implementation

- Is it known to what extent the capacity of the system is used?
- Have the user and control functions been separated sufficiently?
- Is the troubleshooting procedure clear?

12.6.8 Checklist for development stage – coding rules

Maintainability, reusability, and testability are quality characteristics influenced by how a system is developed and documented. When standards are set for development, this will enhance the uniformity of the coding. These code standards can be forced or checked by static analyzers such as code analyzers. Most of these tools have their own built-in code testing which can be adjusted to the needs or standards of the organization. The coding standards also depend on the programming language used. The following checklist provides some examples.

- Does the cyclomatic complexity exceed a certain maximum?
- Does the number of statements exceed a certain maximum?
- Have the pre- and post-conditions been mentioned in the function body?
- Have comment lines been used to explain the coding?
- Does the number of nested levels exceed a certain maximum?
- Has a standard been used for the names of functions and variables?

PART IV

Infrastructure

Introduction

The term “infrastructure” refers to all the *facilities* that are required for structured testing. It includes those needed for executing tests (test environment) and those that support efficient and effective execution of test activities (tools and test automation).

Often, in the early stages of the development (and test) process, much of the hardware needed is not yet available to testers – they need simulators or prototypes of the hardware components. In the later stages of the test process these must be replaced by upgraded prototypes, or even the real components. Chapter 13 describes the different *test environments* required at different stages in the test process.

Tools can make life a lot easier for testers. The range of *test tools* is extensive, offering support for a broad range of test activities. Chapter 14 provides an overview of the various tools that can be applied in the different phases of the test process.

Successful automation of test execution requires more than just a powerful tool. The dream of “the tool that will automate your test for you” was shattered long ago. Chapter 15 explains how to achieve successful *test automation*, describing the technical as well as the organizational issues.

Chapter 16 discusses specific issues that arise when the tester works in an environment that deals with *mixed signals* (analog as well as digital). It describes the specific requirements for the test environment and various tools and techniques currently available.

13 Embedded software test environments

13.1 Introduction

The development of a system containing embedded software involves many test activities at different stages in the development process, each requiring specific test facilities. This chapter discusses what kind of test environments related to those test activities and goals are required. The differences in these environments will be illustrated using the generic layout of an embedded system (see Figure 1.2).

This chapter applies to all projects where embedded systems are developed in stages: starting with simulated parts (*simulation stage*) and then replacing them one by one by the real thing (*prototyping stage*) until finally the real system works in its real environment (*pre-production stage*). After development, a final stage can be distinguished in which the manufacturing process is tested and monitored (*post-development stage*). These stages can be related to the multiple-V lifecycle (see Chapter 3) as shown in Figure 13.1. The detailed descriptions of test environments in this chapter are structured according to the multiple V-model but they are definitely not restricted to projects that apply this model.

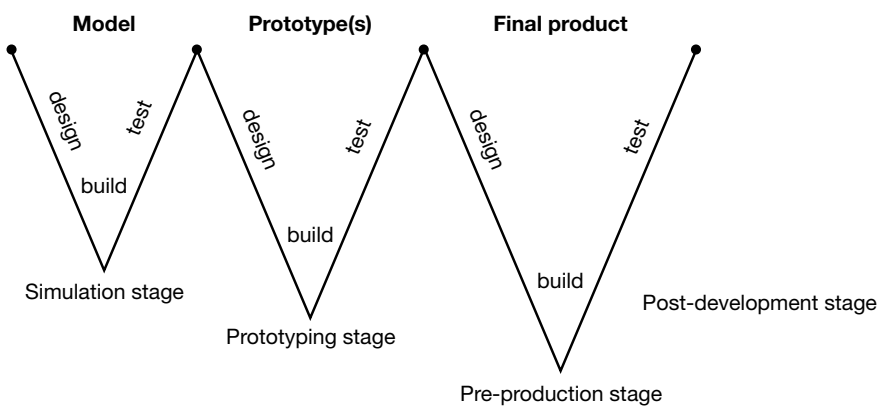


Figure 13.1
Relationship between the development stages and the multiple V-model

Some organizations have a testing process divided into tests as shown in Figure 13.2. This can be mapped onto the development stages mentioned above as follows. The early models developed to simulate the system are tested in "model tests" (MT) and "model-in-the-loop" tests (MiL) – this corresponds to the simulation stage. In "rapid prototyping" (RP) an experimental model with high performance capacity and plenty of resources (for example, 32-bit floating point processing) is tried out in a real-life environment to check if the system can fulfill its purpose – this is also part of the simulation stage. In "software-in-the-loop" testing (SiL) the real software including all resource restrictions (for example, 16-bit or 8-bit integer processing) is tested in a simulated environment or with experimental hardware. In "hardware-in-the-loop" testing (HiL) the real hardware is used and tested in a simulated environment – both "SiL" and "HiL" are part of the prototyping stage. In "system test" (ST) the real system is tested in its real environment – this corresponds to the pre-production stage.

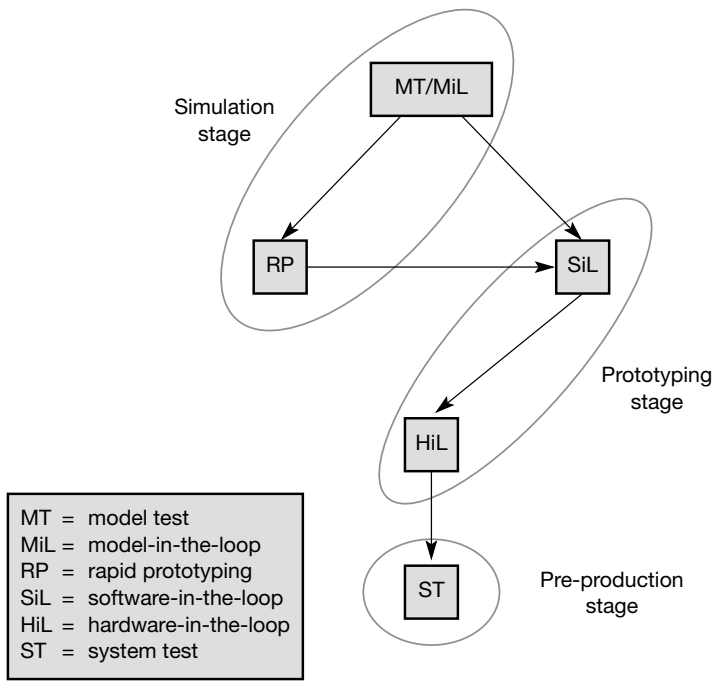


Figure 13.2
A test process showing the gradual transitions from simulated to real

This chapter focuses on the test facilities required to enable the *execution* of tests on the embedded system. In addition to this, test facilities may be used to support the *testing* process. Here it will be necessary to review and analyze earlier test results and compare these with current results. Sometimes the customer may demand that detailed information is submitted on how, when, where, and with which configuration the test results were obtained. Thus, it is essential to keep accurate test logs and to maintain strict hardware and software configuration

control over the prototype units. If the development (and test) process is long and complicated, it is advisable to maintain a database to store all test data. This should contain all relevant data such as date/time, location, serial number of the prototype under test, identification of hardware and software configuration, calibration data for data recording equipment, test run number, data recording numbers, identification of test plan, identification of test report, etc. Chapter 14 discusses more such test facilities.

Parallel execution of tests often requires more than one test environment. In addition, more than one specimen of the same prototype may be required because one prototype may be destroyed, for instance in environmental tests.

13.2 First stage: simulation

In model-based development, after verification of the requirements and conceptual design, a simulation model is built. The developer of an embedded system uses the executable simulation models to support the initial design, to prove the concept, and to develop and verify detailed requirements for the next development steps. This stage is also described as "model testing" and "model-in-the-loop." The goals of testing in this stage are:

- proof of concept;
- design optimization.

Executing a simulated model and testing its behavior requires a specific test environment or a test bed. Dedicated software tools are needed to build and run the simulation model, to generate signals, and analyze responses. The use of Petri nets (Ghezzi *et al*., 1991) should be mentioned here as a technique which is particularly useful for asynchronous systems. This section will explain different ways of testing the simulation models and the required test environments. For this purpose the generic illustration of an embedded system (Figure 1.2) can be simplified to show only the embedded system and its environment, the plant.

Testing in the simulation stage consists, in general, of the following steps.

1 *One-way simulation*. The model of the embedded system is tested in isolation. One at a time, input is fed into the simulated system and the resulting output analyzed. The dynamic interaction with the environment is ignored.
2 *Feedback simulation*. The interaction between the simulated embedded system and the plant is tested. Another term for this (often used with process control systems) is that the "control laws" are verified. The model of the plant generates input for the embedded system. The resulting output is fed back into the plant, resulting in new input for the embedded system, and so on.
3 *Rapid prototyping*. The simulated embedded system is tested while connected to the real environment (see Figure 13.3). This is the ultimate way of assessing the validity of the simulation model.

Figure 13.3
Rapid prototyping of an embedded system for a car (courtesy: ETAS)

Feedback simulation requires that a valid model of the plant, which is as close to the dynamic behavior of the plant as desired or as possible, is developed first. This model is verified by comparing it with the actual behavior of the environment. It often starts with the development of a detailed model followed by a simplification, both of which must be verified. The model of the plant can be verified by one-way simulation.

It is sound engineering practice, in model-based development, to develop the model of the plant *before* the model of the embedded system itself. The (simplified) model of the plant is then used to derive the control laws that must be implemented in the embedded system. Those control laws are then validated using one-way simulation, feedback simulation, and rapid prototyping.

13.2.1 One-way simulation

The embedded system is simulated in an executable model and the behavior of the environment is ignored. Input signals are generated and fed into the model of the embedded system and output signals monitored, recorded, and analyzed. The simulation is "one-way" because the dynamic behavior of the plant is not included in the model – Figure 13.4 depicts this situation schematically.

Tools are required to generate the input signals for the model of the embedded system and to record the output signals. Comparing the recorded signals against expected results can be performed manually, but alternatively by a tool. The signal-generating tool may even generate new input signals based on the actual results in order to narrow down the possible shortcomings of design.

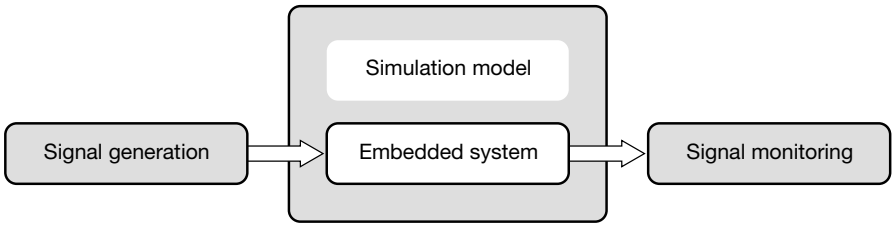


Figure 13.4
One-way simulation

Depending on the simulation environment, the generation of input signals and the recording of output signals can be done in various ways.

- Simulation of the embedded system on a computer platform. Delivery of input signals and recording of output signals by means of hardware on the peripheral bus of the simulation platform, and by manual control and readout of variables within the model via the operation terminal of the computer platform.
- Simulation of the embedded system in a CASE environment as well as generating the input stimuli and capturing the output responses in the same environment. The executable model is created in a modeling language such as UML. From the corresponding use cases and sequence diagrams, test cases can be generated automatically.

One-way simulation is also used for the simulation of a model of the plant. In this case, the dynamic behavior of the embedded system (incorporating the control laws) is not included in the model.

13.2.2 Feedback simulation

The embedded system and its surrounding environment (the plant) are both simulated in one executable dynamic model. This option is feasible if the complexity of the simulation model of the embedded system and its environment can be limited without compromising the accuracy of the model.

For instance, suppose that the embedded system to be designed is the cruise control for a car. The environment of the system is the car itself – and also the road, the wind, the outside temperature, the driver, etc. From the perspective of feasibility and usefulness it can be decided that the model is (initially) limited to the cruise control mechanism, the position of the throttle, and the speed of the car. Figure 13.5 shows a schematic representation of this type of simulation.

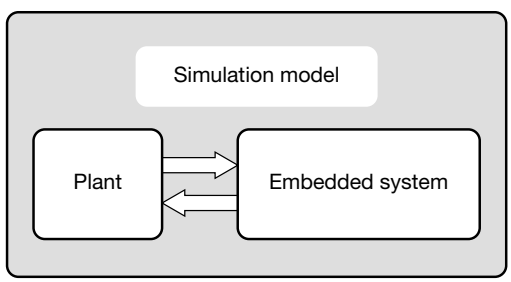


Figure 13.5
Feedback simulation

Feedback simulation may be the next step in a complex design process, after one-way simulation of the embedded system itself or the plant.

Verification of the design is obtained through the execution of a number of test cases that may include deviations in characteristics of the model and changes in the state in which the model is running. The response of the model is monitored, recorded, and subsequently analyzed.

The test bed for the model must afford the ability to change characteristics within the model of the embedded system or within the model of its environment. It must also offer readout and capture of other characteristics from which the behavior of the model can be derived. Again, as with one-way simulation, the test bed can be arranged on a dedicated computer platform as well as in a CASE environment.

The tools used in this stage are:

- signal generating and monitoring devices;
- simulation computers;
- CASE tools.

13.2.3 Rapid prototyping

An optional step is the additional verification of control laws by replacing the detailed simulation of the plant with the actual plant or a close equivalent (see Figure 13.6). For instance, the simulation of the control laws for a cruise control mechanism, running on a simulation computer, can be verified by placing the computer on the passenger seat of a car and connecting it to sensors and actuators near the mechanics and electronics of the car (see Figure 13.3). A high performance computer is typically used here and resource restrictions of the system are disregarded for a time. For instance, the rapid prototype software may be 32-bit floating point, while the end product is restricted to 8-bit integer processing.

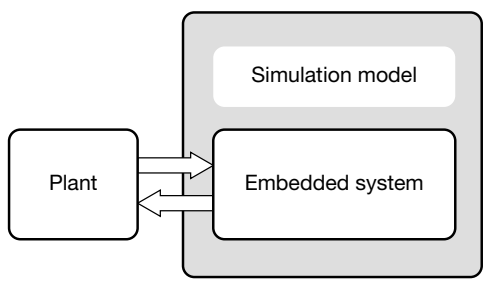


Figure 13.6
Rapid prototyping

13.3 Second stage: prototyping

The development of an embedded system without the use of a model starts at this point. As in the case of model-based development as described above, the prototyping stage starts after the goals for the simulation stage have been achieved.

The goals of testing in this stage are:

- proving the validity of the simulation model (from the previous stage);
- verifying that the system meets the requirements;
- releasing the pre-production unit.

In the prototyping stage, actual system hardware, computer hardware, and experimental and actual versions of the software gradually replace simulated components. The need for interfaces between the simulation model and the hardware arises. A processor emulator may be used. Signals that were readily available in the simulation model may not be as accessible with the real hardware. Therefore, special signal pickups and transducers, may have to be installed as well as signal recording and analysis equipment. It may be necessary to calibrate pickups, transducers, and recording equipment and to save the calibration logs for correction of the recorded data. One or more prototypes, are developed. Often this is a highly iterative process, where the software and hardware are further developed in parallel and frequently integrated to check that they still work together. With each successive prototype, the gap with the final product narrows. With each step, the uncertainty about the validity of the conclusions reached earlier is further reduced.

To clarify and illustrate this gradual replacement of simulated components, four major areas are identified in the generic scheme of an embedded system (see Figure 13.7). They are the parts of the whole that can be individually subjected to simulation, emulation, and experimental or preliminary versions in the prototyping stage:

1 The system behavior realized by the *embedded software*. It can be simulated by:
   - running the embedded software on a host computer compiled for this host;
   - running the embedded software on an emulator of the target processor running on the host computer. The executable of the embedded software can be considered "real" because it is compiled for the target processor.
2 The *processor*. In the early development stages, a high performance processor can be used in a development environment. An emulator of the final processor can be used to test the real software – compiled for the target processor – in the development environment.

3 The *rest of the embedded system*. It can be simulated by:
   - simulating the system in the test bed on the host computer;
   - constructing an experimental hardware configuration;
   - constructing a preliminary (prototype) printed circuit board with all of its components.

4 The *plant* or the environment of the embedded system. It can be simulated statically by means of signal generation, or dynamically by means of a simulation computer.

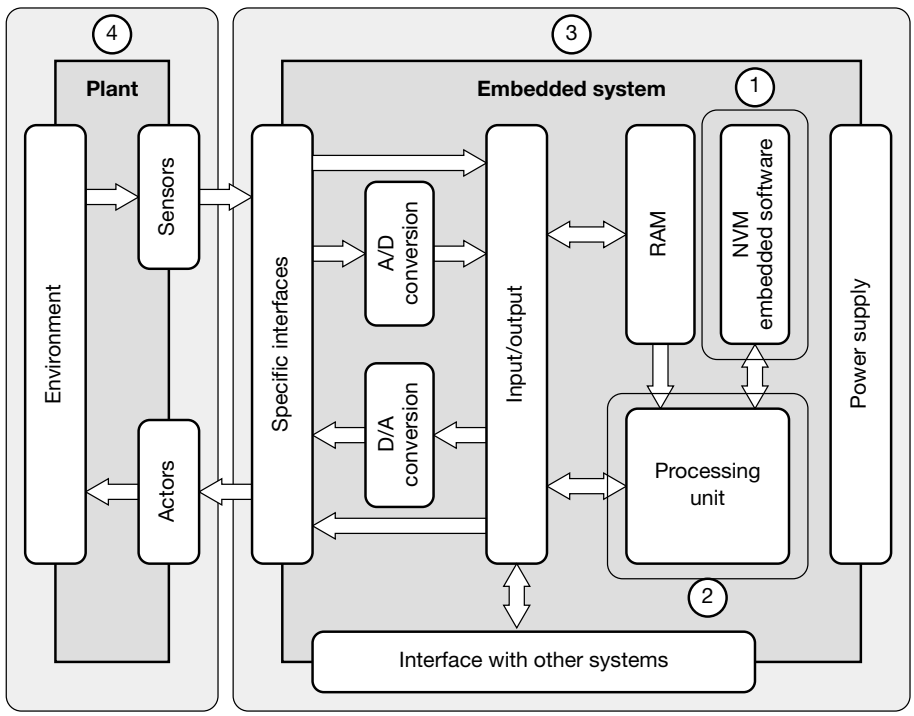


Figure 13.7 Simulation areas in an embedded system

In the prototyping stage, the following test levels (see section 4.1.2) are applicable:

- software unit test;
- software integration test;
- hardware/software integration test;
- system integration test;
- environmental test.

Each test level requires a specific test environment, which is described in the following sections. In these a table will recur which refers to the simulation areas of Figure 13.7 – these provide a simple overview of how each test level progresses towards the situation of a "final real product."

In the prototyping stage, the terms "software-in-the-loop" and "hardware-in-the-loop" are used sometimes. They correspond to the test types software integration test, hardware/software integration test, and system integration test as described here.

13.3.1 Software unit and software integration tests

For the software unit (SW/U) tests and the software integration (SW/I) tests, a test bed is created that is comparable to the test environment for the simulation model. The difference lies in the object that is executed. In the prototype stage the test object is an executable version of a software unit, or a set of integrated software units, that is developed on basis of the design or generated from the simulation model.

The first step is to compile the software for execution on the host computer. This environment (host) has no restrictions on resources or performance, and powerful tools are commercially available. This makes development and testing a lot easier than in the target environment. This kind of testing is also known as *host/target testing*. The goal of the tests on these "host-compiled" software units and integrated software units is to verify their behavior according to the technical design and the validation of the simulation model used in the previous stage.

The second step in SW/U and SW/I tests is to compile the software for execution on the target processor of the embedded system. Before actual execution of this software on the target hardware, the compiled version can be executed within an emulator of the target processor. This emulator may run on the development system or on another host computer. The goal of these tests is to verify that the software will execute correctly on the target processor.

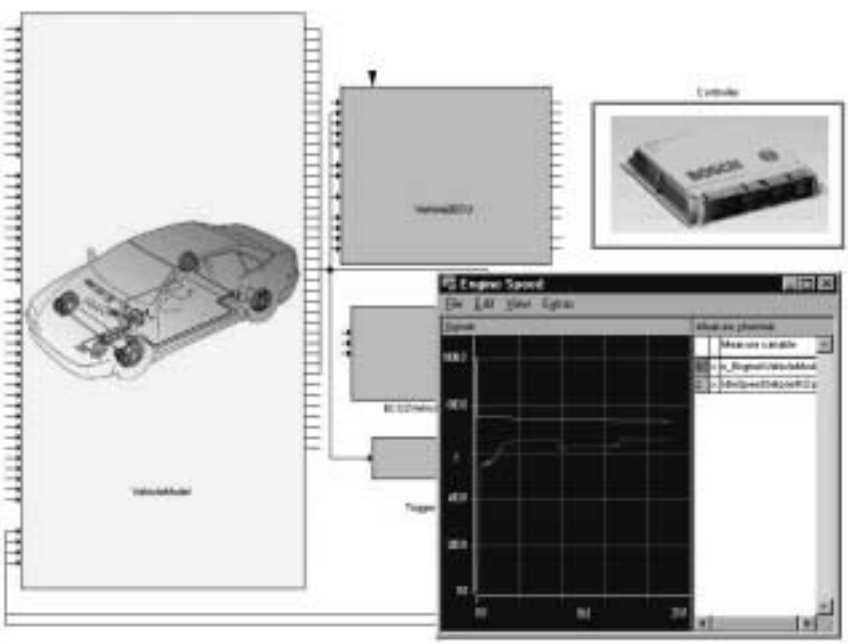


Figure 13.8
"Software-in-the-loop" testing (courtesy: ETAS)

In both of the above mentioned situations the test bed must offer stimuli on the input side of the test object, and provide additional features to monitor the behavior of the test object and to record signals (see Figure 13.8). This is often provided in the form of entering break points, and storing, reading, and manipulating variables. Tools that provide such features can be part of the CASE environment or can be specifically developed by the organization developing the embedded system.

Table 13.1 provides an overview of the level of simulation in both steps in SW/U and SW/I tests. The columns refer to the simulation areas in the generic scheme of an embedded system (see Figure 13.7).

Table 13.1 Simulation level for SW/U and SW/I tests

	Embedded software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated

13.3.2 Hardware/software integration tests

In the hardware/software integration (HW/SW/I) test the test object is a hardware part on which the integrated software is loaded. The software is incorporated in the hardware in memory, usually (E)EPROM. The piece of hardware can be an experimental configuration, for instance a hard-wired circuit board containing several components including the memory. The term "experimental" indicates that the hardware used will not be developed further (in contrast with a prototype which is usually subject to development. The goal of the HW/SW/I test is to verify the correct execution of the embedded software on the target processor in co-operation with surrounding hardware. Because the behavior of the hardware is an essential part of this test, it is often referred to as "hardware-in-the-loop" (see Figure 13.9).

Figure 13.9 Hardware-in-the-loop testing (courtesy: ETAS)

The test environment for the HW/SW/I test will have to interface with the hardware. Depending on the test object and its degree of completeness, the following possibilities exist:

- offering input stimuli with signal generators;
- output monitoring with oscilloscopes or a logic analyzer, combined with data storage devices;
- in-circuit test equipment to monitor system behavior on points other than the outputs;
- simulation of the environment of the test object (the "plant") in a real-time simulator.

Table 13.2 provides an overview of the level of simulation in the HW/SW/I test. The columns refer to the simulation areas in the generic scheme of an embedded system (see Figure 13.7).

Table 13.2 Simulation level for HW/SW/I tests

	Embedded software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated

13.3.3 System integration tests

All the hardware parts that the embedded system contains are brought together in the system integration test, generally on a prototype printed circuit board. Obviously, all the preceding tests SW/U, SW/I, and HW/SW integration level will have to be executed successfully. All the system software has to be loaded. The goal of the system integration test is to verify the correct operation of the complete embedded system.

The test environment for the system integration test is very similar to that for the HW/SW/I test – after all, the complete embedded system is also a piece of hardware containing software. One difference may be found in the fact that the prototype printed circuit board of the complete system is provided with its final I/O and power supply connectors. The offering of stimuli and monitoring of output signals, possibly combined with dynamic simulation, will take place via these connectors. After signal monitoring on connectors, in-circuit testing may take place at predefined locations on the printed circuit board.

Table 13.3 provides an overview of the level of simulation in the system integration test. The columns refer to the simulation areas in the generic scheme of an embedded system (see Figure 13.7).

Table 13.3
Simulation level for system integration tests

	Embedded software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated
System integration	Real (target)	Real (target)	Prototype	Simulated

13.3.4 Environmental tests

The goal of environmental tests in this stage is to detect and correct problems in this area in the earliest possible stage. This is in contrast with environmental tests during the pre-production stage, which serve to demonstrate conformity. Environmental tests preferably take place on prototypes that have reached a sufficient level of maturity. These prototypes are built on printed circuit boards and with hardware of the correct specifications. If EMI countermeasures are required in the embedded system, they will also have to be taken on the prototype that is subject to the tests.

The environmental tests require very specific test environments. In most cases the test environments can be purchased or assembled from purchased components to meet requirements. If specific types of environmental tests are performed only occasionally, it is recommended that the necessary equipment is rented, or that the tests are contracted out to a specialist company.

During the environmental tests, the embedded system may be tested under operation, in which case simulation of the plant will have to be created as in the two previous test levels.

Environmental tests consist of two types:

- *Emphasis on measuring*. This type of test is intended to determine the degree to which the embedded system influences its environment – for instance, the electromagnetic compatibility of the system. The tools required for this type of tests are therefore measurement equipment.
- *Emphasis on generating*. These tests serve to determine the susceptibility of the embedded system to surrounding conditions. Examples are temperature and humidity tests, shock and vibration tests, and tests on electromagnetic interference. The required test facilities are devices that inflict these conditions on the system, such as climate chambers, shock tables and for EMC/EMI tests equipped areas. Measurement equipment is required as well.

Table 13.4 provides an overview of the level of simulation in the environmental test.

	Embedded software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated
System integration	Real (target)	Real (target)	Prototype	Simulated
Environmental	Real (target)	Real (target)	Real	Simulated

Table 13.4
Simulation level for environmental tests

13.4 Third stage: pre-production

The pre-production stage involves the construction of a pre-production unit, which is used to provide final proof that all requirements, including environmental and governmental, have been met and to release the final design for production.

There are several goals of testing in this stage.

- To finally prove that all requirements are met.
- To demonstrate conformity with environmental, industry, ISO, military, and governmental standards. The pre-production unit is the first unit that is representative of the final product. Preliminary tests may have been performed on earlier prototypes to detect and correct shortcomings but the results of these tests may not be accepted as final proof.
- To demonstrate the system can be built in a production environment within the scheduled time with the scheduled effort.
- To demonstrate the system can be maintained in a real-life environment and that the mean-time-to-repair requirements are met.
- Demonstration of the product to (potential) customers.

The pre-production unit is a real system, tested in the real-life environment. It is the culmination of all the efforts of previous stages, as shown in Table 13.5. The pre-production unit is equal to, or at least representative of the final production units. The difference with the production units is that the pre-production units may still have test provisions such as signal pickoffs, etc. However, these provisions are of production quality instead of laboratory quality. It may be necessary to have more than one specimen because tests may run in parallel or units may be destroyed in tests. Sometimes the pre-production units are referred to as “red-label units.” whereas real production units are called “black-label units.”

Table 13.5 Pre-production as the conclusion of the gradual replacement of simulated components by real ones

	Embedded software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated
System integration	Real (target)	Real (target)	Prototype	Simulated
Environmental	Real (target)	Real (target)	Real	Simulated
Pre-production	Real (target)	Real (target)	Real	Real

The pre-production unit is subjected to real-life testing according to one or more predetermined test scenarios. In the example of a cruise control mechanism, these tests might consist of a number of test drives with a pre-production unit of the mechanism installed in a prototype car. Other pre-production units may be subjected to environmental qualification tests. In the previous stages, the test input signals may have been generated from the simulation model, and the output data may have been monitored online or stored on hard disk for analysis later. In the pre-production stage, instrumentation and data display and recording facilities are required. Maybe telemetry is needed – this might be the case if the tests involve a missile, an aircraft, a racing car or other vehicles with limited space for instrumentation equipment or human observers. Due attention should be given to the quality and calibration of the test instrumentation, and the telemetry, the data display and recording equipment – if the data gathered cannot be trusted, the tests are worthless.

The types of tests that are applicable to this stage are:

- system acceptance test;
- qualification tests;
- safety execution tests;
- tests of production and maintenance test facilities;
- inspection and/or test by government officials.

Applicable test techniques are:

- real-life testing;
- random testing;
- fault injection.

Typical tools used in this stage are:

- environmental test facilities, such as climate chambers, vibration tables, etc.;
- data gathering and recording equipment;
- telemetry;
- data analysis tools;
- fault injection tools.

Table 13.6 provides a final and complete overview of all the discussed test levels (using the terminology from both sections 4.1.2 and Figure 13.2) and the gradual replacement of simulated components by real ones.

Table 13.6
Overview of all test levels and the gradual replacement of simulated components by real ones

Test levels		Embedded software	Processor	Rest of embedded system	Plant
One-way simulation	MT	Simulated	–	–	–
Feed-back simulation	MiL	Simulated	–	–	Simulated
Rapid prototyping	RP	Experimental	Experimental	Experimental	Real
SW/U, SW/I (1)	SiL	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	SiL	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	HiL	Real (target)	Real (target)	Experimental	Simulated
System integration	HiL	Real (target)	Real (target)	Prototype	Simulated
Environmental	HiL/ST	Real (target)	Real (target)	Real	Simulated
Pre-production	ST	Real (target)	Real (target)	Real	Real

13.5 Post-development stage

The prototyping and pre-production stages finally result in a "release for production" of the system. This means that the tested system is of sufficient quality to be sold to the customer. But how can the organization be sure that all the products that are going to be produced will have the same level of quality? In other words, if the production process is of insufficient quality, then the products will be of insufficient quality, despite a successful release for production. Therefore, before the start of large-scale production, the organization may require additional measures in order to make this production process controllable and assure the quality of manufactured products.

The following measures should be considered:

- *Development and test of production facilities*. In the development of embedded systems, the development and test of the facilities for the production of the released system may be equally important. The quality of the production line has a major influence on the quality of an embedded system, therefore it is important to acknowledge the necessity of this measure. Development and subsequent testing of the production line is defined as a post-development activity, but can be done at any time during the development of the embedded system. However, it cannot take place during the modeling stage because of the absence of the final characteristics of the system. On the other hand, the production line has to be available before the actual start of production.
- *First article inspection*. Sometimes it is required to perform a first article inspection on the first production (black-label) units. The units are checked against the final specifications, change requests, production drawings, and quality standards to ensure that the production units conform with all specifications and standards. In the multiple V-model (see Chapter 3) this might be considered to be the very last V. The purpose is to develop and test the production process. After all, if the quality of the end product falls short, then all the development and test efforts will have been in vain.
- *Production and maintenance tests*. For quality control purposes it may be required to perform tests on production units. This may involve tests on each individual unit or more elaborate tests on production samples. Built-in test facilities may be required for troubleshooting and maintenance in the field. Facilities for production and maintenance tests must be designed in the unit (design for test) and tested in the development process. Moreover, development tests and their results may be used as basis for production and maintenance tests.

14 Tools

14.1 Introduction

Software now makes up the largest part of embedded systems and the importance of software is still increasing. This software becomes more complex and is implemented in critical systems more often. This has a major impact on testing – there is more to be tested and it is more complicated. Also changes in the embedded systems market have an impact on testing. Time-to-market and quality become major competing factors. This means that more and more complex software has to be developed at a higher quality level in less time. As a consequence, time pressure for testing and quality demands increase. A well-structured test process alone is now not sufficient to fulfill the quality demands within time. This, and the complexity of the systems, make today's testing job almost impossible without the use of test tools. With the proper use of test tools, faster testing with consistently good quality is possible.

A test tool is an automated resource that offers support to one or more test activities, such as planning and control, specification, constructing initial test files, execution of tests, and assessment (Pol *et al.*, 2002).

The using of test tools is not an objective on its own, but it has to support the test process. Using test tools makes only sense if it is profitable in terms of time, money, or quality. Introduction of test tools in a non-structured test process will seldom be successful. For high quality testing, structuring and automation are necessary. For successful introduction of test automation there must be a lifecycle and knowledge of all the test activities. The test process must be repeatable.

Up to date information about available tools can be found through:

- conferences and exhibitions;
- vendor exhibitions for free trial versions of tools;
- research reports such as the CAST report (Graham *et al.*, 1996);
- the internet
  - www.ovum.com (ovum reports)
  - www.testing.com (Brian Marick's Testing Foundations)
  - www.soft.com/Institute/HotList (Software Research Incorporated)
  - most tool vendors have a website with detailed information about their (commercial) products.

14.2 Categorization of test tools

Tools are available for every phase in the test lifecycle. Figure 14.1 shows how tools can be categorized according to where they are applied in the testing lifecycle (see Chapter 6). These tools are explained in more detail in the following sections – the list presented is not meant to be exhaustive.

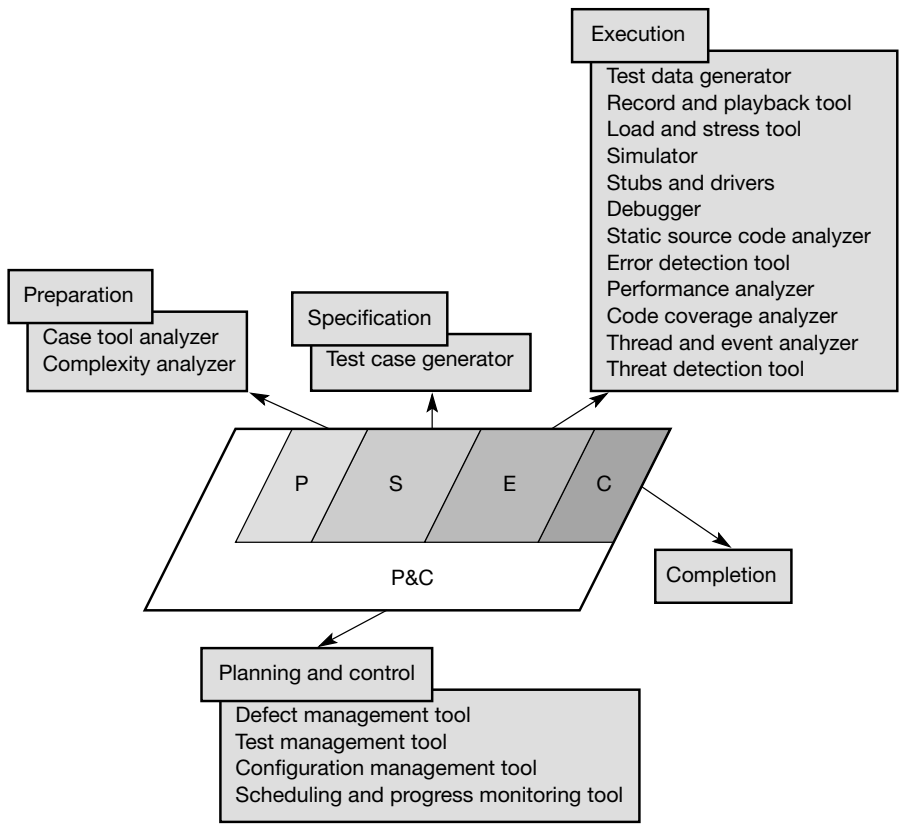


Figure 14.1
Test tools related to the testing lifecycle

Most tools are available commercially but it is also possible that project-specific tools are developed in-house.

14.2.1 Planning and control

Most tools for the planning and control phase are project management based. These tools are not developed especially for testing, except for test and defect management tools. The following tools support planning and control:

- defect management tool;
- test management tool;
- configuration management tool;
- scheduling and progress monitoring tool.

14.2.1.1 Defect management tool

Defects detected during the test process must be collated in an orderly way. For a small project, a simple file system with a few control procedures is sufficient. More complex projects need at least a database with the possibility of generating progress reports showing, for instance, the status of all the defects or the ratio of solved to raised defects. Several tool vendors have developed defect management systems. All these are designed around a database system and have tracing and reporting facilities. The tools have the ability to implement a workflow with security and access levels. Most have e-mail facilities and some of them are web-enabled.

A defect management system is used to store defects, trace them, and generate progress and status reports. The tracing and reporting facilities are almost indispensable as control instruments.

14.2.1.2 Test management tool

The test management tools provided by tool vendors offer only the functionality to store test cases, scripts, and scenarios and sometimes integrate defect management. They have no facilities to store test plans and project schedules. This type of test management tool is not very useful for planning and control activities. Tools with the ability to link system requirements to test cases are now available. These tools afford the ability to keep track of the coverage of test cases in relation to the system requirements. In addition, they become very useful if system requirements are changed or might change – the test manager is able to show the impact of these changes and this may give cause to block the changes.

14.2.1.3 Configuration management tool

All testware has to be stored in a proper way. After quality checks, the testware should be frozen. The next step is to make the testware configuration items and store them in a configuration management tool. In addition to the regular testware (test cases and test documentation) the description of the test environment, tools used, calibration reports, and specially developed stubs, drivers and simulators should be subject to configuration management. A configuration management tool provides the functionality to keep track of the configuration items and their changes.

14.2.1.4 Scheduling and progress monitoring tool

For scheduling and progress monitoring, many different tools are available which are specially made for project management. Scheduling and progress monitoring are combined in most tools. These kinds of tools are very useful for a test manager – combined with information from the defect and test management systems it is possible to keep track of progress in very detailed manner if necessary.

Some of these tools are quite complex and intended for experienced users. For many testers, such tools are too time consuming and will prove counterproductive. For them a simple spreadsheet will do the job. The test manager can import the required information from the spreadsheet into the advanced planning tool.

14.2.2 Preparation phase

14.2.2.1 CASE tool analyzer

CASE tools, such as tools for UML, are able to do consistency checks. These can be used to check whether or not the design has omissions and inconsistencies. In this way, a CASE tool can be used to support the "testability review of the test basis" activity – provided that the tool was used to design the system of course.

14.2.2.2 Complexity analyzer

This kind of tool is capable of giving an indication about the complexity of the software. The degree of complexity is an indicator of the chance of errors occurring and also of the number of test cases needed to test it thoroughly. An example of a measure of complexity is the McCabe cyclomatic complexity metric (Ghezzi *et al.*, 1991). The metric C is defined as

$$C = e - n + 2p,$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components (usually 1). C defines the number of linearly independent paths in a program. For structured programs with no `GOTO` statements, the cyclomatic number equals the number of conditions plus 1. Developers should try to keep complexity as low as feasible. McCabe is not the only person to develop a method to describe to complexity of software, another well-known method was developed by Halstead (Ghezzi *et al.*, 1991).

14.2.3 Specification phase

14.2.3.1 Test case generator

Test design is a labor intensive job and using tools would make it simpler – unfortunately, not many are available. To use a test design tool, a formal language (mainly mathematically-based) has to be used for the system requirement specification. An example of such a formal language is the Z language.

Model checking has the ability to generate test cases based on infinite state machines. Model checking is a technique in which no commercial tools are currently available.

With statistical usage testing there is also the possibility of using a test case generator (see section 11.7).

14.2.4 Execution phase

Many types of tools are available for the execution phase. Many tool vendors have generic and highly specialized tools in their portfolio. The following types of tools are described in this section:

- test data generator
- record and playback tool
- load and stress test tool
- simulator
- stubs and drivers
- debugger
- static source code analyzer
- error detection tool
- performance analyzer
- code coverage analyzer
- thread and event analyzer
- threat detection tool.

14.2.4.1 Test data generator

This generates a large amount of different input within a pre-specified input domain. The input is used to test if the system is able to handle the data within the input domain. This type of test is used when it is not certain beforehand if the system can handle all data within a certain input domain. To perform evolutionary algorithm testing (see section 11.6) an automated test data generator is essential.

14.2.4.2 Record and playback tool

This tool is very helpful in building up a regression test. Much of the test execution can then be done with the tool and the tester can mainly focus on new or changed functionality. Building up these tests is very labor intensive and there is always a maintenance problem. To make this type of tool profitable in terms of time, money, or quality, delicate organizational and technical decisions have to be made. Section 15.3 shows how to organize a test automation project and Appendix C describes how test automation can be implemented technically.

14.2.4.3 Load and stress test tool

Systems that have to function under different load conditions have to be tested for their performance. Load and stress tools have the ability to simulate different load conditions. Sometimes, one may be interested only in the performance under load or, sometimes more interesting, the functional behavior of the system under load or stress. The latter has to be tested in combination with a functional test execution tool.

Another reason to use this type of tool is to speed up the reliability test of a system. A straightforward reliability test takes at least as long as the mean time between failures, which often is in the order of months. The test duration can be reduced to a matter of days, by using load and stress tools that can simulate the same amount of use in a much shorter period.

14.2.4.4 Simulator

A simulator is used to test a system under controlled conditions. The simulator is used to simulate the environment of the system, or to simulate other systems which may be connected to the system under test. A simulator is also used to test systems under conditions that are too hazardous to test in real life, for example the control system of a nuclear power plant.

Simulators can sometimes be purchased on the open market, but usually a dedicated simulation environment has to be developed and built for the system under test. See section 13.2 for detailed information about simulation.

14.2.4.5 Stubs and drivers

Interfaces between two system parts can only be tested if both system parts are available. This can have serious consequences for the testing time. To avoid this and to start testing a system part as early as possible, stubs and drivers are used. A stub is called by the system under test and provides the information the missing system part should have given – a driver calls the system part.

Standardization and the use of a testbed architecture (see Figure 14.2) can greatly improve the effective use of stubs and drivers. The testbed provides a standard (programmable) interface for the tester to construct and execute test cases. Each separate unit to be tested must have a stub and driver built for it with a specific interface to that unit but with a standardized interface to the testbed. Techniques for test automation, such as data-driven testing (see section 15.2.1), can be applied effectively. Such a testbed architecture facilitates the testing of any unit, the reuse of such tests during integration testing, and large-scale automation of low-level testing.

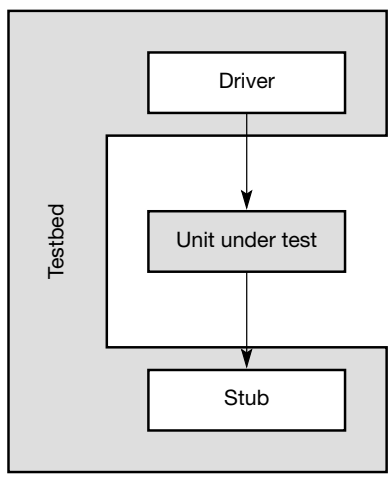


Figure 14.2 Testbed for testing a unit using stubs and drivers

14.2.4.6 Debugger

A debugger gives a developer the opportunity to run a program under controlled conditions prior to its compilation. A debugger can detect syntactic failures. Debuggers have the ability to run the application stepwise and to set, monitor, and manipulate variables. Debuggers are standard elements of developer tools.

14.2.4.7 Static source code analyzer

Coding standards are often used during projects. These standards define, for instance, the layout of the code, that pre- and post-conditions should be described, and that comments should be used to explain complicated structures. Additional standards which forbid certain error-prone, confusing or exotic structures, or forbid high complexity, can be used to enforce a certain stability on the code beforehand. These standards can be forced and checked by static analyzers. These already contain a set of coding standards which can be customized. They analyze the software and show the code which violates the standards. Be aware that coding standards are language dependent.

14.2.4.8 Error detection tool

This type of tool is used to detect run-time errors. The tool detects a failure and diagnoses it. Instead of vague and cryptic error messages, it produces an exact problem description based on the diagnosis. This helps to locate the fault in the source code and correct it.

14.2.4.9 Performance analyzer

This type of tool can show resource usage at algorithm level while load and stress tools are used to do performance checks at system level. The tools can show the execution time of algorithms, memory use, and CPU-capacity. All these figures give the opportunity to optimize algorithms in terms of these parameters.

14.2.4.10 Code coverage analyzer

This type of tool shows the coverage of test cases. It monitors the lines of code triggered by the execution of test cases. The lines of code are a metric for the coverage of the test cases. Be aware that code coverage is not a good metric for the quality of the test set. At unit test level, for instance, the minimum requirement should be 100 percent statement coverage (otherwise this would mean that parts of the code have never executed at all).

14.2.4.11 Thread and event analyzer

This type of tool supports the detection of run-time multi-threading problems that can degrade Java performance and reliability. It helps to detect the causes of thread starvation or thrashing, deadlocks, unsynchronized access to data, and thread leaks.

14.2.4.12 Threat detection tool

This type of tool can detect possible threats that can lead to malfunctioning. The type of threats are memory leaks, null pointer dereference, bad deallocation, out of bounds array access, uninitialized variable, or possible division by zero. The tools detect possible threats which mean that further analysis is necessary to decide whether the threat is real.

14.2.5 Completion phase

Basically, the same tools as for the planning and control phase are used for this phase – although the focus here is on archiving and extracting conclusions rather than planning and control. Configuration management tools are used for archiving testware and infrastructure, and tool descriptions, calibration reports, etc. Scheduling and monitoring tools are used to derive project metrics for evaluation reports. Defect management tools are used to derive product metrics and sometimes provide information for project metrics.

Test automation 15

15.1 Introduction

Test execution is situated on the critical path to product introduction. Test automation is used, for instance, to minimize the time needed for test execution. Automation can be profitable in terms of time, money, and/or quality. It can also be used to repeat boring tests procedures. Some tests, such as statistical usage testing and evolutionary algorithms, are impossible without test automation.

In principle, the execution of almost every test can be automated. In practice, only a small part of the tests are automated. Test automation is often used in the following situations:

- tests that have to be repeated many times;
- a basic test with a huge range of input variations – the same steps have to be executed every time but with different data (for instance, evolutionary algorithms);
- very complicated or error-prone tests;
- testing requires specialized equipment that generates the appropriate input signals, activates the system, and/or captures and analyzes the output.

Changes in the system, design, and requirements are common and are also a threat to the usability of the automated tests. The consequences of these changes can be:

- additional test cases;
- changed test cases;
- different result checks;
- changed system interface;
- changed functionality;
- different signals;
- different target platform;
- different internal technical implementation.

Automated testing saves money and time if a test set can be used several times (payback is considered to be achieved if the complete test set is used between three and six times). This means that the automated tests should be used for consecutive releases, or in different stages of development. Which also means that automated tests have to be designed to deal with these uncertainties due to changes in the system, design, and requirements.

This chapter describes how automated testing has to be implemented technically for commercially available tools in general as well as for proprietary tools. Then it describes how the process of introducing automated testing is similar in any development process with a requirement, design, deployment, and maintenance phase.

15.2 The technique of test automation

15.2.1 Design for maintainability

To maximize maintainability, flexibility and ease of use, some design decisions have to be made. To avoid a test suite from becoming shelfware it is designed to be as independent as possible of changes – changes in the test object, the technical connection between the test suite and the system under test, and the tests and data used. This is established by reducing the dependencies:

- *test data* is stored outside the test suite;
- the description of *test actions* is separated from the technical implementation of these actions;
- the process of how to handle an automated test is implemented independent of its environment and system under test – this explains why any test suite needs several *essential components*;
- the *communication* between the system under test and the test suite is implemented independent of the central process of the test suite.

These issues will be discussed in more detail in the following sections. The following terms will be used:

- *Automated testing*. Executing tests using an automated test suite.
- *Test suite*. Everything necessary to execute a test by pushing just one button.
- *Data driven*. An approach where physical test actions are separated from test data, and the test data is the impulse for the test.
- *Framework*. A library of reusable modules and design.

15.2.1.1 Test cases

The test cases are stored outside the test suite (see Figure 15.1). They can be stored in a database or spreadsheet. An additional test case means adding only this test case to the storage of the test cases. The test suite doesn't have to be changed.

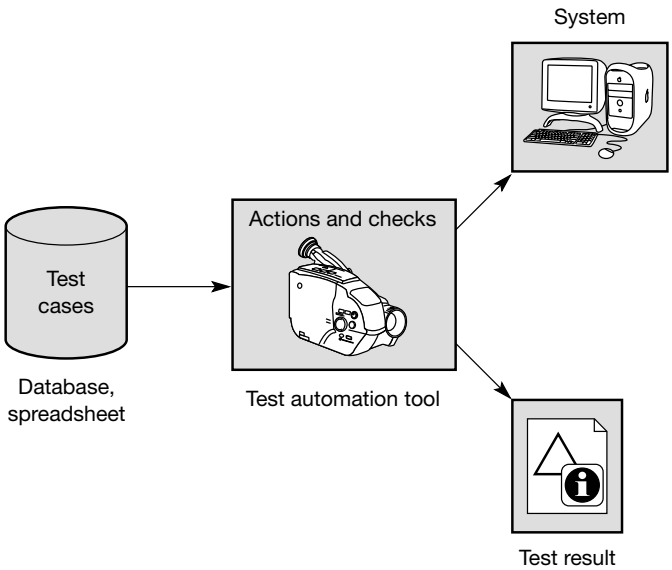


Figure 15.1
Test data stored outside the test automation environment

15.2.1.2 Test actions

The technical implementation of a certain test action can be rather complicated. This implementation should be hidden from the normal user of the test suite – the only thing to know is which test actions are available (see Figure 15.2). By choosing the right test actions the right test script can be composed. If the functionality of the system under test stays the same but the technical implementation is changed (for instance, another interface is used) only the technical implementation must be changed. The user can still use the same script as if nothing has happened. Table 15.1 shows an example where test data and actions are combined in one test script. This combination is quite common.

ID	**Test action**				
TC01	Open_address_book				
		Name	**First name**	**Street**	**Number**
TC02	Add_address	Stewart	Mark	Oak street	15
TC03	View_address	Stewart	Mark	Oak street	15
TC04	Delete_all_addresses				
		Check			
TC05	Count_addresses	0			
TC06	Close_address_book				

Table 15.1
Layout of test script. When the first column is empty the rest of the row is interpreted as comment

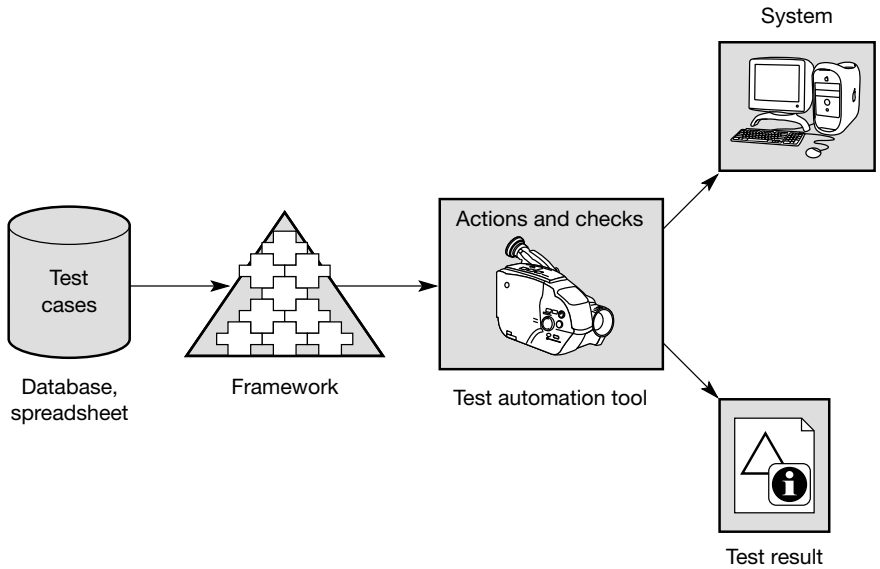


Figure 15.2 The technical implementation of the test actions is stored in a framework hidden from the user of the test suite

15.2.1.3 Essential components

Test suites must have several essential components to be able to support the test execution process properly. Those basic components are part of the generic design of a test suite (see Figure 15.3) which is called a blueprint – this has three major sections. The first is the input side with test scenarios and scripts. The last is the output side with status reports and progress reports. The processing unit, or test suite kernel, is situated in between. This kernel makes it possible to test with one push of a button. The test suite continues working without human intervention until the complete test scenario (see section 6.4.3) is executed. Appendix C provides a more detailed description of the blueprint.

15.2.1.4 Communication

The system under test can have the following forms:

- executable code on the same machine as that on which the test suite is installed;
- executable code on a test target (separated from the test suite environment);
- system under test in a simulation environment;
- system under test as pre-production system.

The forms mentioned here are related to different stages in the development process. So, the form of the system under test changes during the test process. Many test cases should be used during all stages of the development process. This means that the communication layer should be independent of the test

cases defined. The test case will stay the same while the communication, or technical implementation of the communication, is changed. There has to be a communication layer (see Figure 15.3) for all forms.

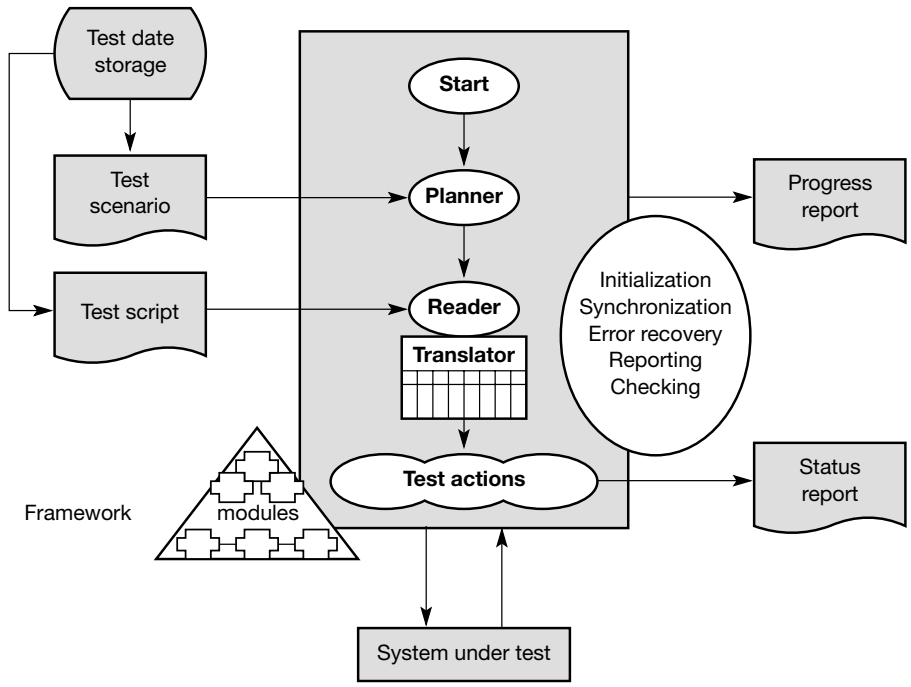


Figure 15.3 Blueprint of a test suite

15.2.2 Maintenance

Due to certain changes, the automated tests may not run correctly any more and maintenance becomes necessary. The blueprint is developed to keep maintenance to a minimum. The parts possibly affected by changes are:

- the test data;
- synchronization, error recovery, checks and initialization;
- application specific modules;
- the communication layer.

In the introduction to this chapter, a list of possible changes was presented. The impact of those changes on the test suite are listed below.

- *Adding test cases*. Only the data store has to be changed.
- *Changed test cases*. The data store has to be changed. Sometimes it is necessary to add new test actions to the framework.
- *Different result checks*. Only the data store has to be changed if only the output prediction is changed. Otherwise checks and or test actions have to be changed or added.
- *Changed system interface*. The communication layer has to be changed. If the signal definition is changed, the test data should also be changed if these test data contain information about the signals.
- *Changed functionality*. This change can have an effect on the test data because new test cases have to be added. The synchronization, error recovery, checks, and initialization can be changed because the system dialog and sequence of functionality can have changed. New functionality means new test actions. The new functionality can also affect the communication layer.
- *Different signals*. There have to be changes in the communication layer. Sometimes it is also necessary to change the test data.
- *Different target platform*. This can affect the communication layer.
- *Different internal technical implementation*. If the external interface and functionality are not changed then this can only affect the software implementation which bypasses the external interface.

15.2.3 Testability

To make test automation easier it is essential that testability is a quality attribute recognized during the design and implementation of the system. It is rather time consuming and expensive to built a dedicated communication layer because the behavior of the system is impossible to monitor. Some of the tests can be substituted by built-in self tests. Sometimes it is possible to have output on a certain pin that is only accessible during testing. In production, a parameter in the software or a slide change in the hardware switches this option off.

15.3 Implementing test automation

Test tools are used to enhance the quality, lower the costs of test execution, and/or save time. In practice, for functionality tests this is not always that easy to realize. The execution of automated tests once or twice for the same release is not a problem, but it is not obvious to use basically the same tests for many releases. When the system functionality has changed in the new release, the automated test will usually not run properly anymore and will often simply crash after encountering unexpected system responses. The automated test suite must be “repaired” and “upgraded” to make it run once again and cover the changed system behavior that must be tested. In bad cases, the repair costs more than executing the test manually. The objectives of quality, cost, and time are no longer realized and the overall perception is one of failure.

Investment in the implementation of these kinds of tools is high and so the complete test set should be executed three to six times at least to achieve pay back. This also means that at the start of the usage of test automation, one should already be aware that this is a long-term project. Most problems can be overcome by using a lifecycle model with proper analyzing activities and decision points (see Figure 15.4). This lifecycle has three consecutive phases:

- initiation;
- realization;
- exploitation.

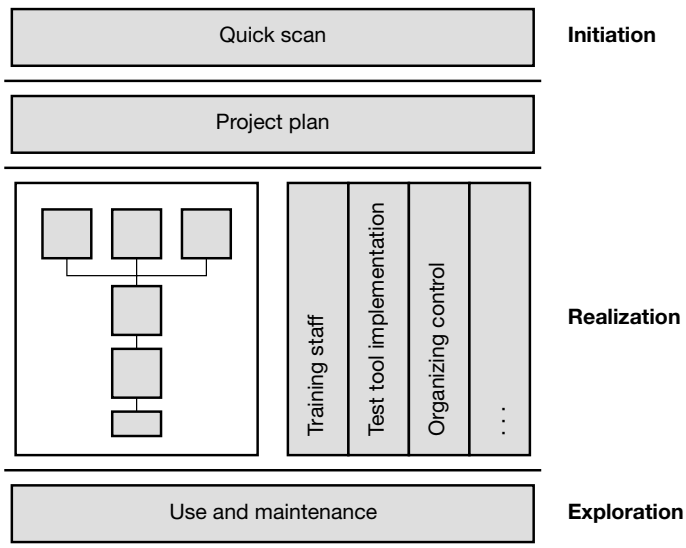


Figure 15.4
The lifecycle of a test tool implementation project

The initiation phase is merely a period of analysis and investigation. The objectives of test automation are determined along with their practicability. If there is no knowledge about the availability of proper test tools, a rough tool search can be started. Sometimes detailed test selection is part of this phase. The development of a dedicated tool can be the outcome of this search. This will have a major impact on the practicability of the objectives. This phase will help reach a formal decision about whether to start the realization of test automation or not.

At the start of the realization phase, a detailed test tool selection is conducted if this activity was not part of the initiation phase. Test automation objectives are quantified, the test objectives are described in detail, and the technical implications of the system under test are investigated. The output of this process biases the design process. According to the design, the automated test environment is developed. The end product of this phase will be an implemented test automation environment capable of realizing the test automation objectives.

In the exploitation phase, the test tool environment is used for test execution. This environment should be maintained due to changed and new functionality.

15.3.1 Initiation

The test automation project starts with a feasibility study. The main objective of this is to get enough information to decide if test automation is the proper thing to do. In order to get the right information, three items are analyzed:

- test automation objectives;
- system under test;
- test organization.

The test automation objectives are the expectations of the organization about test automation. The objectives are described in terms of quality, cost, and time – there should be a balance between these. High quality generally leads to lower cost and time saving. The objectives are also the criteria for the determination of the success of the test automation project. Other objectives include in which development stage should automated testing be implemented, and what should be tested with this automated test environment.

The system under test is investigated technically to find out how it should be tested and whether additional equipment, such as simulators and signal generators, is necessary. It is also necessary to know whether the interfaces, internal technical structure, and the functionality of the system are regularly changed during development or consecutive releases. A system that is changed quite often in respect of these points is difficult to test with test tools because the usage of the system has continuously changed. This will bring extra cost and more focus on the maintainability of the automated test environment.

The test organization should use the automated test environment and provide the test cases. The test cases can be already stored in a certain standard format. It would be rather nice if this format could be used in the test automation environment. The proper use of test tools demands a repeatable test process. This means that there are well-described test cases derived by using test design techniques and a test strategy which decides on the relative importance of the system parts and quality attributes. To use and develop the automated test environment, special knowledge is necessary. If this knowledge is not available then employees have to be trained. Another consequence can be that the usage of the automated test environment is made as simple as possible by developing an intuitive user interface. This will hide the complexity of the automated test environment.

In addition to all these primary conditions one should also know if there is a tool available to use for the system itself. This will depend on the system itself, the target, the instantiation of the system, and also the program language. Figure 15.5 shows the different stages in the test tool selection process.

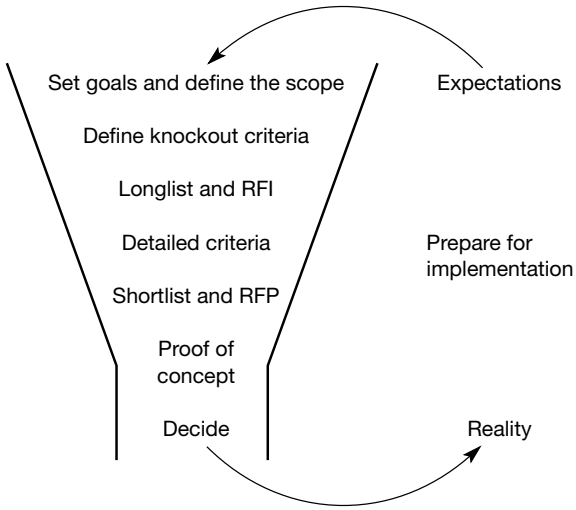


Figure 15.5
Test tool selection process

Test tool selection starts with defining the expectations. Based on these and the capabilities of the different test tools a decision is finally made. This decision can be that there is no proper tool available. The ultimate consequence is the development of a dedicated tool or no automated test execution.

The following activities have to be carried out.

1 *Set goals and define the scope*. Goals can include improving the test process, shortening test execution time, diminishing test execution capacity, or enhancing quality test execution. Be aware that goals such as solving organizational problems or attention-free testing can not be realized by tools. The scope shows whether the automated test environment should be used for only one particular system, a range of systems, or across the company.
2 *Define knockout criteria*. These criteria are mostly based on the technical implications of the system under test. A tool that cannot meet these criteria is useless and is immediately deleted from the list.
3 *Long list and RFI*. With the knockout criteria in mind, a long list of tools is prepared. All tool vendors get a Request For Information (RFI). This is a list of questions a vendor should answer. The accuracy of this answer is also an indicator of the vendor's service level – tool vendors that do not answer are discounted.
4 *Detailed criteria*. A detailed list of selection criteria is prepared. There are three kinds of criteria:
   - technical (compatibility, stability, script language, etc.);
   - functional (object recognition, synchronization, report functionality, operating system, target platforms, etc.);
   - demands to tool vendors (continuity, education, support, installed base, etc.).

   The criteria are grouped by knockout criteria, demands, and wishes.

5 *Short list and RFP*. Based on the detailed criteria, tools are selected from the long list. The selected tools are placed on a short list which is sent to the vendors with detailed criteria and a Request for Proposal (RFP). This gives an indication of the costs, and the possibility of support and training. Based on this information, two or three tool vendors are invited to show the capabilities of their tools on a demonstration test environment. A list of capabilities each vendor should show with their tool should be prepared.
6 *Proof of concept*. Before using the tool in practice, and investing much effort in its implementation, a pilot can be started. If this pilot is successful the tool can be implemented. Many tool vendors offer the chance to use the tool on trial for a certain period.
7 *Decision*. The results of the proof of concept give enough information to decide if an implementation of the tool will meet expectations. The outcome could be that the objectives can only be met with a dedicated developed test tool.

The next step is to analyze the costs and benefits. Both have objective and subjective properties. The costs can be measured in terms of money and manpower. The subjective costs are these incurred in dealing with the introduction of something new, and a change in working practice. The benefits can also be quantified in terms of saving money by spending less time and labor on testing. The enhancement of the quality of the test process is the subjective part of the benefits.

Be aware that the direct cost of tools is high but the cost of labor during implementation, usage, and maintenance are usually much higher.

This whole process should aid the enhancement of awareness and positive support for implemention as a major side effect.

15.3.2 Implementation

The major objective is to prepare an automated test environment capable of meeting the test automation objectives. In order to realize this some activities are defined. The first is the preparation of a plan of action – the purpose being to show the activities, people involved, and time estimation.

Then the design and development will start. In this process, six activities are identified (see Figure 15.6).

The definition of the test automation objectives, detailed analysis of test object and test approach are necessary for the design decisions. If, for instance, the main emphasis is on testing processes rather than details, this means that test actions can be defined on a very high level. If the test object is not very stable (many changes during the development process and consecutive releases) the test automation environment should be designed for easy maintenance. If the main focus is on time saving only the labor and time intensive tests are automated.

With this in mind, the architect designs the automated test environment based on the blueprint (see Appendix C). Based on this design, the engineer develops the automated test environment and is responsible for implementing

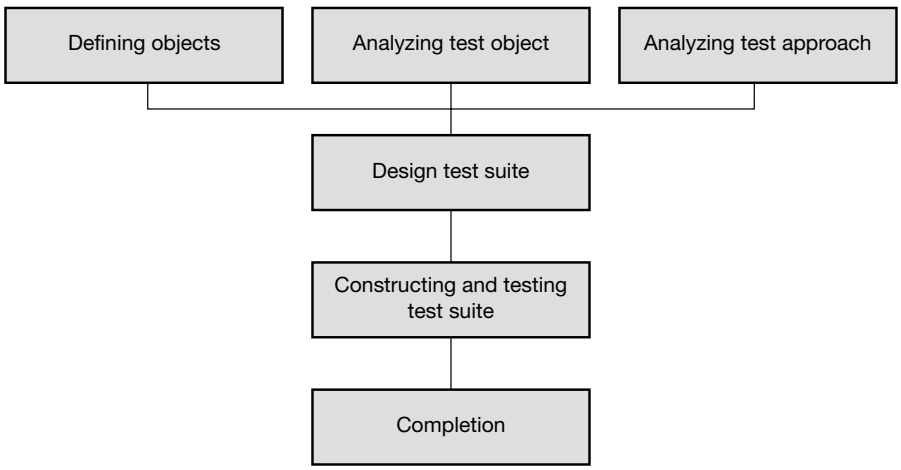


Figure 15.6
Activities of the design and development stage

the necessary functionality to execute the delivered test cases. The functional testers develop the test cases based on test design techniques. In the last step, a final test of the automated test environment is performed.

Now, the design and all other documentation about the automated test environment is brought up to date. Together with the automated test environment, this is deployed and used in the maintenance phase.

In additional to these major activities, parallel activities can be executed as necessary. These are activities such as educating people in the use of test tools, implementing these tools, maintaining them, and organizing control.

15.3.3 Exploitation

The automated test environment is now ready for use. The impact on the test process of the availability of an automated test environment is shown in Figure 15.7. The automated test environment affects all phases of the test process. During the testability review of the test basis, the impact on the automated test environment and the testability with the usage of this environment should be kept in mind. During specification, the test cases should be described in a readable format for the automated test environment, and the environment should be adjusted due to changes in the test object. The test execution is partly covered by the automated test environment and in the completion phase all changes are frozen and stored. It is essential that a complete copy of the automated test environment is subject to configuration management.

The major objective of the exploitation phase is to maintain and use the automated test environment. During this phase, the organization can profit from the benefits of the automated test environment. To meet the test automation objectives and to be able to benefit from this environment, in the next release the focus must be on maintenance of the environment. By neglecting this part of the job the environment will be useless after just one or two releases – and that is probably before the investment is paid back.

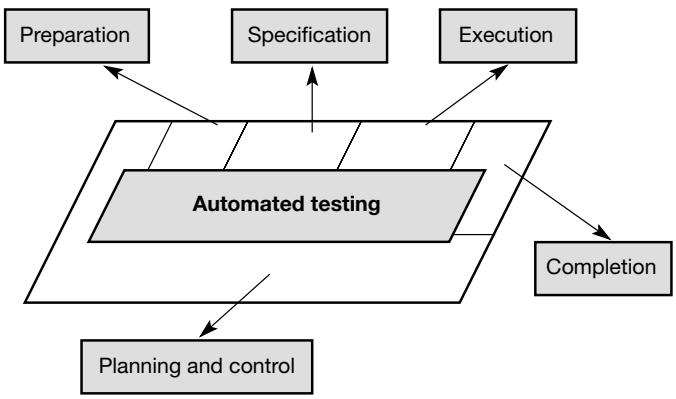


Figure 15.7
Test lifecycle with automated testing

16 Mixed signals

by Mirko Conrad and Eric Sax

A typical embedded system interacts with the real world receiving signals through sensors and sending output to actors that somehow manipulate the environment (see Figure 1.2). Because the physical environment is (by nature) not discrete, such an embedded system must not only deal with digital signals (as "pure computers" do), but also with continuous signals.

This chapter first introduces the typical aspects of the mix of digital and continuous signals ("mixed signals") and the impact on testing. Then it discusses how to deal with mixed signals on the input side (section 16.2) and on the output side (section 16.3)

16.1 Introduction

In a continuous environment there is nothing comparable to states in a discrete system. There are no state transitions and sudden changes due to events. The whole system is in a "state" which is not exactly definable. To clarify this fundamental difference between discrete and continuous systems, this section will classify signals through their dependency on time and value.

16.1.1 Signal categories

In general a signal has a value that changes over time. The change of both time and value can be either discrete or continuous. As a result of the time–value and discrete–continuous combinations, four categories of signals can be distinguished. These are illustrated in Figure 16.1:

As a consequence, the following signals can be differentiated for practical use:

- *Analog signals* are continuous in time and value. This is the physical type of a signal that can – in the electrical domain for example – be expressed by voltage depending on time. For the stimuli description, a functional expression can be used (e.g. voltage=f(*t*))
- *Time-quantized signals* are those for which the value is known at only certain points in time (sample points). A typical example is the capturing of a signal (e.g. system response) which is defined by time and value pairs.

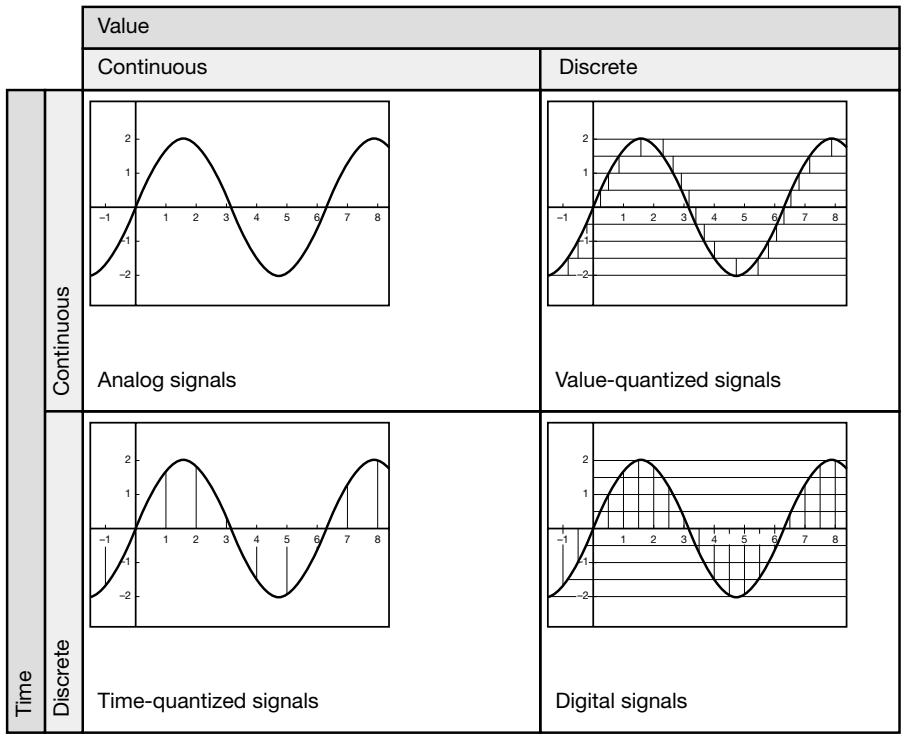


Figure 16.1
Signal categories / variants of signals

- *Value-quantized signals* are discrete in value and continuous in time. An analog signal passing an A/D-converter for further calculations in a processor unit is typical of this type.
- *Digital signals* are those whose values are elements of a limited set only. If the set contains only two elements this special case is called a binary signal. Due to the way data is handled by semi-conductors, binary signals are of significant importance for the internal signal processing and control in embedded systems.

The term "mixed signal systems" is used in this context for systems that handle both digital and analog components and have to deal, therefore, with both discrete signals and continuous signals.

To illustrate typical problems in dealing with continuous signals, the situations of purely analog signals and mixed signals will now be discussed further.

16.1.1.1 Purely analog

There is a fundamental difference between the *definition* of input signals (stimuli) and expected responses on the one hand, and *captured results* on the other. Signals can be *specified* by algebraic equations (e.g. f(x)=sin(x)) which represent continuous behavior perfectly (see section 16.2). However, a *captured* signal is

always sampled and is, therefore, time discrete. Furthermore, the separation of noise from a captured signal, which is also continuous, is far more complex in comparison to digital signals. Consequently, it is impossible to capture an analog signal exactly. There is always an uncertainty between the sample points. Due to value continuity – not just two, but an infinite number of elements belong to the range – the exact, expected, or defined value can seldom be realized in a physical environment. Even with a simulator that uses a variable step size for calculating complex algorithms, for example differential algebraic equations (DAE), an exact result is very often not predictable. Therefore, *tolerances* and the definition of ranges of a signal have to be used when deciding if two signals can be considered "equal." This is important, for instance, in establishing pass/fail criteria for test results.

A major difficulty that arises in testing analog components is that analog faults do not cause a simple state change in the logic value. Therefore, the decision as to whether a fault has occurred or not is not simply a matter of "hi" and "lo." Limits (tolerances in time) and bands (tolerances in value) are needed. As a consequence, the measurement and analysis of analog signals is no trivial task and will be illustrated in detail in section 16.3.

16.1.1.2 Mixed signals

The handling of binary signals is clear-cut – a signal is either hi or lo; consequently it is clearly definable and quantifiable. Faults can be detected and calculations executed without vagueness. Dealing with analog signals is different. But the distinguishing feature of mixed signal systems is the combination of digital controllers, computers, and subsystems modeled as finite automata coupled with controllers and plants modeled by partial or ordinary differential equations or difference equations. Therefore, mixed signal handling, and especially the synchronization of the two domains (digital and analog), is a very important issue (see Figure 16.2).

The combination of analog and digital signals is extremely challenging for internal signal processing. Therefore, in the design world, time is often handled differently from within the analog and digital world. For example, two different time-step algorithms are used for equation solving during simulation. The digital, clock-based part runs with one step size that defines the exact points in time for changes of values and states, whereas for solving DAEs that describe the analog part a continuous time base is needed (see Figure 16.1 and Figure 16.3). Now if an event occurs in the digital part which influences the analog part, harmonization must take place – the two different times need to be synchronized. Backtracking the leading time or waiting for the subsequent time can achieve this. In addition, they also need to be synchronized when threshold passing or trigger pulses occur which initiate influences from the analog to the digital world.

All these specific aspects of the mixed signal domain require adjusted and specialized test measures.

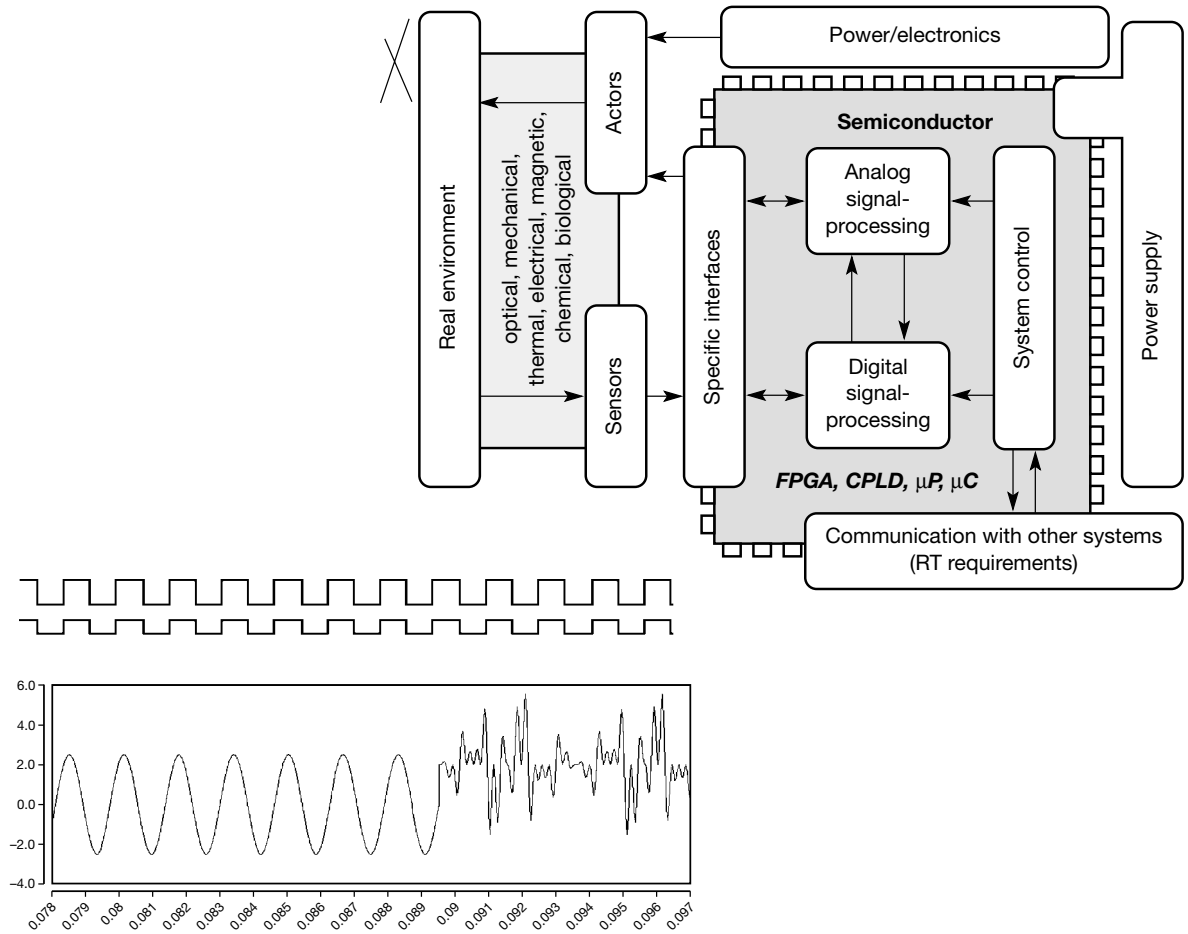


Figure 16.2 Mixed signals as input/output of embedded systems

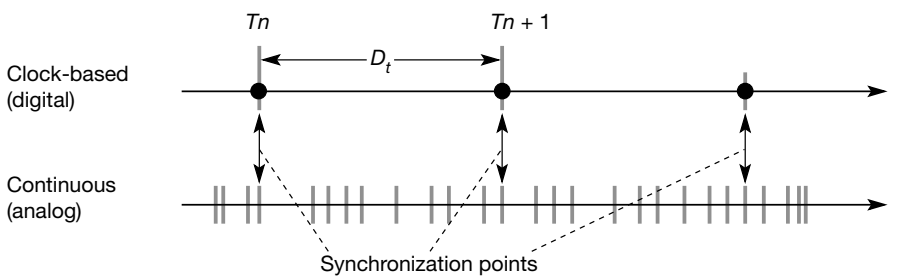


Figure 16.3 Synchronization between digital and analog time

16.1.2 Dealing with mixed signals in different test levels

Testing takes place in all development stages (see also Chapter 3 and Chapter 13). Even in the early design phases, a model-based approach for the development of mixed signal systems provides an executable system model (i.e. block diagram, and the hardware description language VHDL). This system model can be tested dynamically by the function developer (model testing, MiL) (see Figure 16.4). Regarding the model as the system specification, it can be used to generate software. As the first step of the software development for embedded systems, this software doesn't run on the production target yet. However, it is already the complete control software. Dynamic testing at that level is also based on execution (SiL). Finally, the software is integrated with the target controller for productive use. Execution in an prototypical environment (HiL) is the way to test this. See Chapter 13 for a discussion on how the different test environments are related to the different test levels.

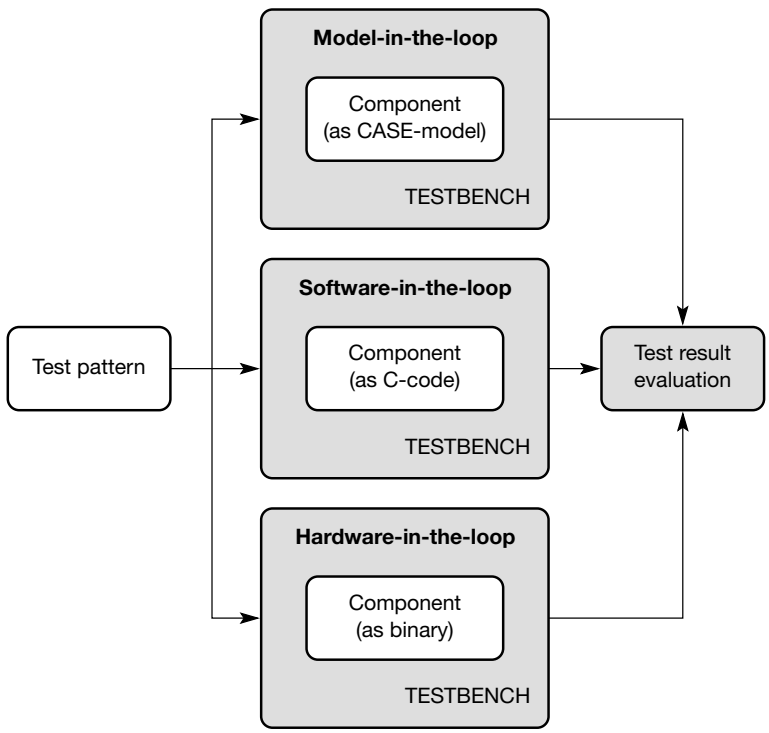


Figure 16.4
Reuse of test input and test output in different test levels

For all these test levels, test signals (often called patterns) are used. The pure description of the signal (for example changes in logical values or algebraic descriptions of continuous signals) is the same for all test levels. As a consequence, the test patterns can be reused during subsequent development and test stages (for example SiL, HiL). Also the test results from the different test levels can be compared and reused as "expected results" for the subsequent stage.

So the reuse of the logical descriptions of inputs and outputs is possible, but the physical representation of the signals is different for different test levels. Pure data flow is sufficient when testing the models or with SiL. But for HiL, the voltage and current must be specified in concrete terms.

16.2 Stimuli description techniques

This section discusses selected stimuli description techniques for mixed signal systems, and provides criteria which support the choice of an appropriate stimuli description technique and tool in a given situation of the project.

For an embedded system, stimuli descriptions are a set of analog or digital signals that are made up of several ranges (see Figure 16.5). A digital signal is clock-based. In relation to the clock, explicit timing delays for rise and fall times can be defined. Furthermore, a format for a concrete change in value (for example according to the previous clock) can be fixed. For a pure analog signal, the description can be based on a symbolic expression, offset, phase shift, and amplitude in addition to the function expression. Finally, realistic stimuli for the complex system under test require the sequential description of several independently defined signal ranges.

16.2.1 Assessment and selection criteria

Existing test methods and tools use rather different stimuli description techniques. As examples of the variety of description techniques there are (test) scripting languages, signal waveforms, tree and tabular combined notation (ISO/IEC 9646, 1996), message sequence charts (ITU, 1996; OMG, 1997), timing diagrams and classification-trees (Grochtmann and Grimm 1993).

In spite of the functional test being of great importance for the verification of systems and being widespread in industry, there are very few techniques and tools for the systematic, methodological-based generation of corresponding stimuli descriptions. In many cases therefore tools and notations of only limited applicability are used.

In order to support the choice of a problem-specific stimuli description technique or an adequate tool for a concrete project, suitable criteria are needed. In the description of tools and techniques in this chapter, the following criteria are used (it is not an exhaustive list). See also Conrad (2001).

- *Supported signal categories*. Which of the four signal categories (analog, time-quantized, value-quantized, digital) are supported by the technique?
- *Type of notation*. Does the description of the stimuli signals take place textually (e.g. scripting or programming language), graphically (e.g. message sequence charts, flowcharts), or through a combination of both? While textual descriptions usually offer a higher flexibility, visualization often enhances the comprehensibility of stimuli descriptions.

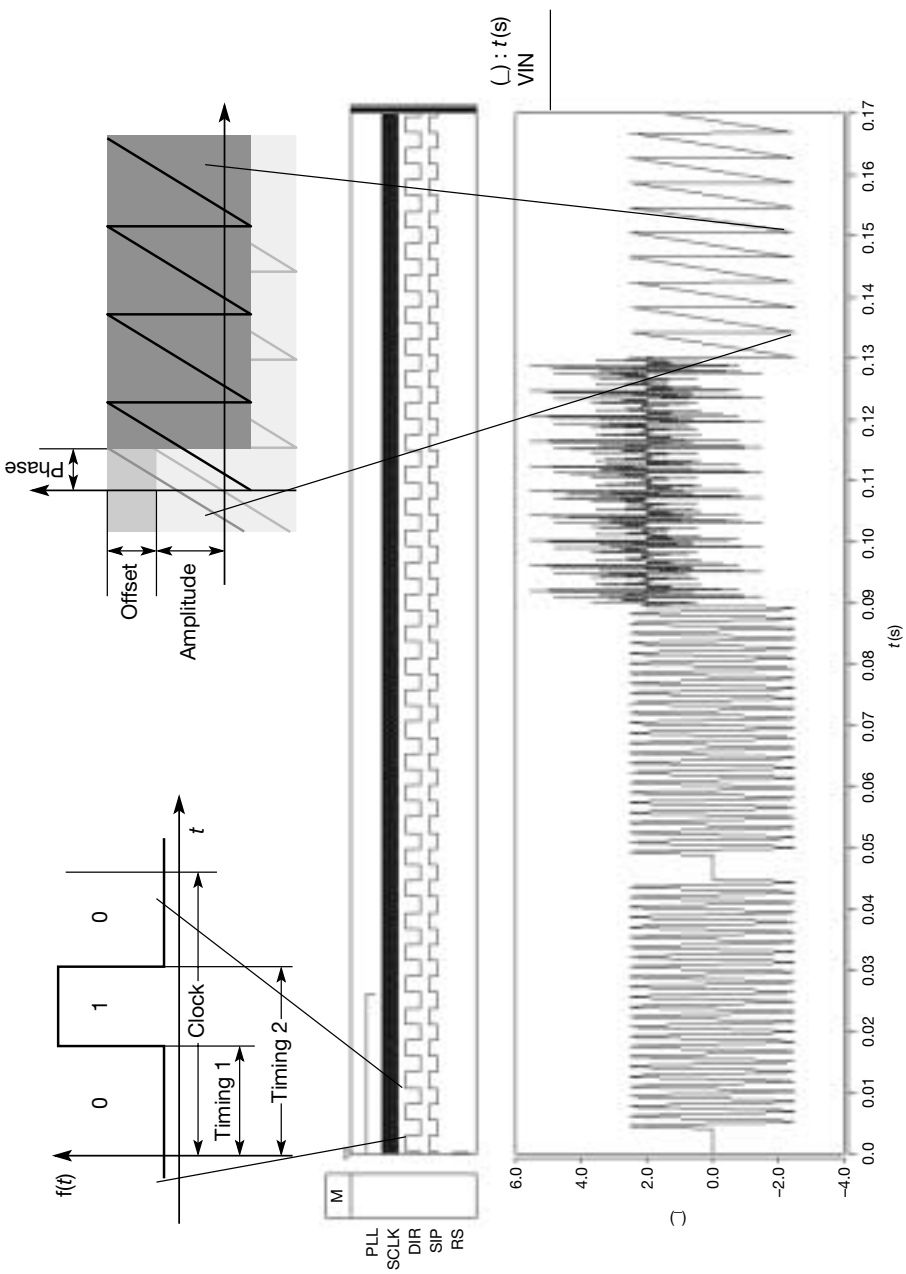


Figure 16.5
Example for a complex signal definition

- *Degree of abstraction*. Are the stimuli described with a low degree of abstraction (i.e. time–value pairs), or are descriptions at a higher level of abstraction (i.e. stepwise defined functions), or are multi-level descriptions at different abstraction levels supported? A multi-level description has the advantage that the tester can concentrate on the essentials.
- *Complexity*. Is it only possible to describe basic forms of signals, or can these be combined to make complex signal waveforms by sequencing and programming (e.g. if-then-else, loops)? See Figure 16.6.

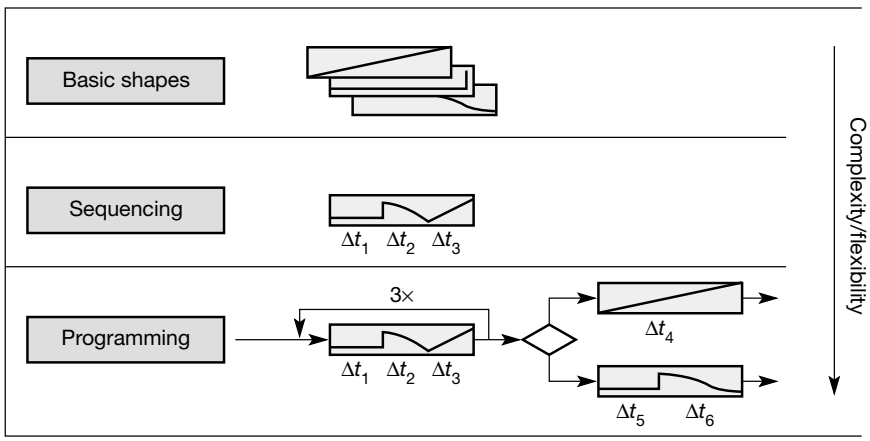


Figure 16.6
Levels of complexity of stimuli description

- *Consideration of outputs*. Is it possible to access the current values of the outputs of the system under test during the test run, and to vary the stimuli generation depending on it? In other words, does the specification of the stimuli take place in an open loop (i.e. a priori and without feedback), or in a closed loop, where the waveform of the stimuli signals depends on the behavior of the system under test and can only be specified during run-time?
- *Coverage criteria*. Are metrics/measures provided for the determination of the test coverage, especially for the more abstract description levels?
- *Methodological support*. Is there support for the determination of the stimuli signals? In other words, can a special test design technique (as in the classification-tree method) be used? This question is crucial because many of the general test design techniques don't deal with the timing aspects of stimuli description for mixed signal systems and are therefore often of only limited applicability.
- *Tool support*. What support is there for the description technique? Are there commercially available tools and prototypes or is it only a paper-based method?

- *Data storage format*. Are the stimuli descriptions stored in a standardized, or at least documented, format that is publicly accessible, or are they stored in a proprietary format? A standardized or publicly accessible format facilitates the separation of the test input specification from the test execution, tool-independent stimuli descriptions, as well as their reuse during different test levels (e.g. MiL➜SiL➜HiL).

16.2.2 Examples of currently available techniques and tools

In order to illustrate the range of existing stimuli description techniques and tools, and to demonstrate the application of the above mentioned set of criteria, a selection of tools and techniques is described and assessed in the following sections. Some of the stimuli description techniques described below are strongly linked to one special tool; others are supported by a variety of tools and could be described as being more tool independent.

16.2.2.1 Timing diagrams

Timing diagrams enable the abstract graphical representation of several discrete signals by means of symbolic waveforms over a global time axis, as well as the relationships between them (for example, delays, setups, and holds). Stimuli signals, as well as the expected outputs, can be specified and related to each other (see Figure 16.7). Different dialects of timing diagrams exist. Timing diagrams are applied, for instance, to the graphic description of test benches for VHDL models of circuits (Mitchel, 1996) or to the description of model properties during model checking (Bienmüller *et al*., 1999).

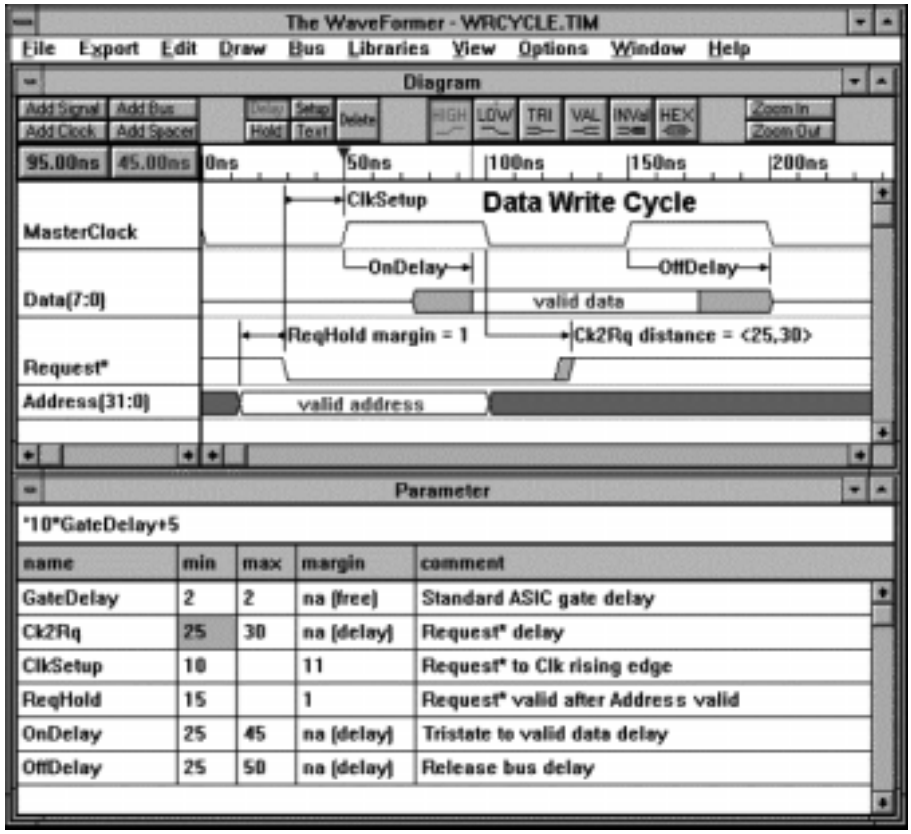
Timing diagrams are supported by different commercial tools, for example WaveFormer and Testbencher Pro developed by SynaptiCAD.

The timing diagram markup language (TDML) provides an exchange format for the superset of information from the different timing diagram and waveform formats. The XML-based TDML enables the storage of this information in a structured form so that it can be easily processed by different tools. An assessment of the suitability of the TDML stimuli description technique is shown in Table 16.1.

16.2.2.2 Graphical stimulus editor of ControlDesk Test Automation

The tool ControlDesk Test Automation by dSPACE (dSPACE, 1999) provides a graphical stimulus editor which makes it easy to create any signal sequences (see Figure 16.8).

The upper part of the stimulus editor facilitates an abstract description of stimuli signals by means of stepwise defined functions in a matrix (see Figure 16.8 upper part. Each row (signal slot) describes a basic signal waveform, such as sine, square, constant, pulse, ramp, triangular, or exponential. The constructed signal consists of a sequence of basic waveforms which are synchronized at certain time intervals (time tags). If necessary these basic waveforms can be



name	min	max	margin	comment
GateDelay	2	2	na (free)	Standard ASIC gate delay
Ck2Rq	25	30	na (delay)	Request* delay
ClkSetup	10		11	Request* to Clk rising edge
ReqHold	15		1	Request* valid after Address valid
OnDelay	25	45	na (delay)	Tristate to valid data delay
OffDelay	25	50	na (delay)	Release bus delay

Figure 16.7 Typical example of a timing diagram

Supported signal categories	–	Main focus on digital signals
Type of notation	+	Textually (TDML) and graphically
Degree of abstraction	+	Abstract
Complexity	0	Sequencing, no programming, dependencies between signals can be described
Consideration of outputs	+	
Coverage criteria	–	
Methodological support	–	
Tool support	+	Commercially available
Data storage format	0	TDML: published, not standardized

Table 16.1 Assessment of the stimuli description technique of TDML

parameterized. In addition to the signal slots it is possible to define control structures (loops, conditional branching, unconditional jumps) in the so-called control slots with which complex and cyclic signal waveforms can be described.

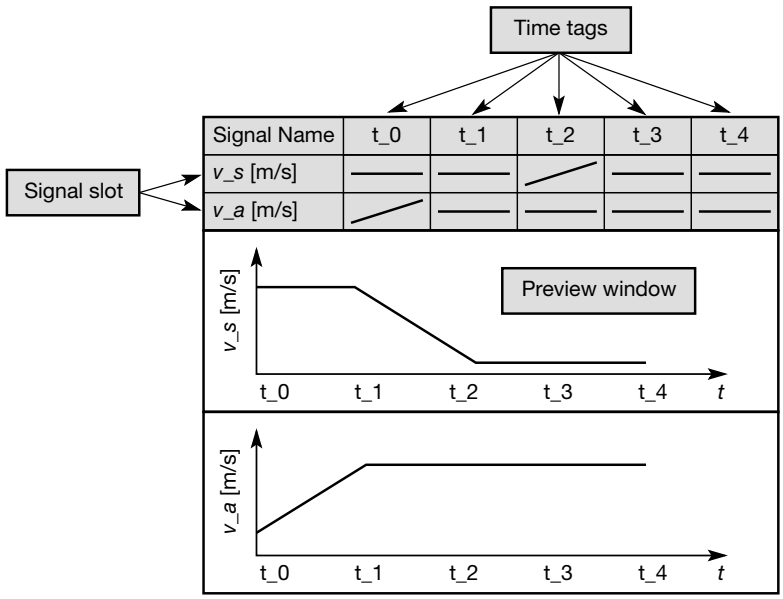


Figure 16.8
Stimulus editor of the tool ControlDesk

The lower part of the stimulus editor provides a low-level graphical description. The signal waveforms described in the matrix are plotted in a preview window.

The tool transforms the stimuli descriptions into the scripting language Python.

An assessment of the suitability of the ControlDesk Test Automation tool is shown in Table 16.2.

Supported signal categories	+	All signal categories
Type of notation	+	Graphical and textual (Python)
Degree of abstraction	+	Two different abstraction–levels
Complexity	+	Sequencing and programming
Consideration of outputs	+	
Coverage criteria	–	
Methodological support	–	
Tool support	+	Commercially available
Data storage format	0	Python: published, not standardized

Table 16.2
Assessment of the stimuli description technique of ControlDesk

16.2.2.3 TESSI-DAVES

TESSI-DAVES by FZI Karlsruhe (Sax, 2000) is a complete test bench generation environment. It supports a target system-independent description of stimuli signals in a general purpose signal description. Thereby, the description of both digital and mixed digital/analog signals is supported.

The basis of signal description in TESSI-DAVES is the strict division into subdescriptions, in which specific requirements for the desired test signals are formulated. This object-oriented approach enables the stepwise creation of signal descriptions. Finally the specification of the complete test description is created by the combination of subdescriptions. Thereby, the test signal description has to be specified only once for a model or a chip. It is then stored in a library and must merely be parameterized in later calls.

The intuitive stimuli definition is supported by a set of editors, for example:

- *Pattern editor*. Defines the digital test pattern for the model inputs and trigger commands for other signals or activities (see Figure 16.9, segment 3).
- *Timing editor*. Defines the exact timing and format for signal changes within a clock period (see Figure 16.9, segment 1).
- *Analog waveform editor*. Specifies an analog signal within different ranges of a standard interval (piecewise or functionally) and defines the parameters such as period, amplitude, offset, etc. (see Figure 16.9, segment 4).
- *Signal control editor*. Specifies a sequence of different signals, for example to shorten the digital pattern by using loops and iterations (see Figure 16.9, segment 2).

Additionally, a preview window is provided, so that all digital and analog signals can be viewed in one window (see Figure 16.9, segment 5).

Furthermore, there is a data manager which enables the use of templates, existing test descriptions, or objects. That way the user can build and vary new tests on the basis of existing ones and they do not always have to start from scratch.

The use of TESSI-DAVES as the integrating frame for the diverse design process demands a conversion from abstract, tool-independent specification of test signals to tool-specific signal descriptions in order to run the complete system. Physically this implies that generators are automatically included in simulator models if the target system is a simulator; test signal files are produced if the destination is an emulator or physical signal generator, or programmed if the conversion is done for a tester system.

An assessment of the stimuli description technique of TESSI-DAVES is shown in Table 16.3.

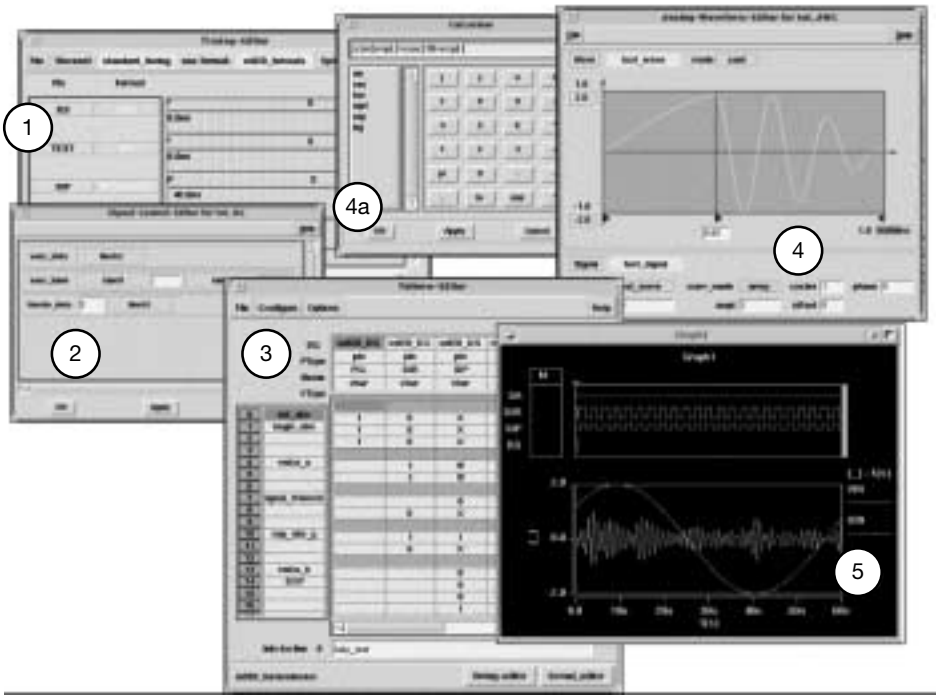


Figure 16.9
Some TESSI editors

1	Supported signal categories	+	all signal categories
2	Type of notation	+	graphical and underlying textual representation
3	Degree of abstraction	+	stepwise refinement supported by editors
4	Complexity	+	sequencing and programming
5	Consideration of outputs	o	if-then-else-constructs may depend on intermediate test execution results
6	Coverage criteria	–	
7	Methodological support	–	
8	Tool support	+	prototypical realization
9	Data storage format	o	proprietary data format: file-based (ASCII)

Table 16.3
Assessment of the TESSI-DAVES stimuli description technique

16.2.2.4 Classification-tree method for embedded systems

The classification-tree method (see section 11.5) is a powerful method for black-box partition testing. A notational extension of the classification-tree method (Conrad *et al.*, 1999; Conrad, 2001) referred to as the classification-tree method for embedded systems (CTM/ES) allows stimuli description for mixed signal systems.

The stimuli description takes place on two different levels of abstraction: in the first level an abstract description of the stimuli signals is carried out which, however, leaves certain degrees of freedom open. At this stage, the stimulus description is not yet restricted to a single stimulus signal. Instead, a set of possible signal courses is specified. The degrees of freedom are removed in the second level; in other words a single stimulus signal is selected from the number of possible input sequences.

For the abstract logical stimuli description, the input domain of the system under test is classified and visualized by means of a tree-like representation, called a classification-tree (see Figure 16.10, upper part). That classification-tree derived from the interface of the system under test defines the "language repertoire" for the description of test scenarios in the underlying matrix, the so-called combination table (see Figure 16.10, lower part). Here all temporal changes of the classified input signals are described in an abstract way.

The stimuli signals are represented in the classification-tree by rectangular nodes called classifications. For each signal the permissible values are determined and partitioned into equivalence classes (intervals or single values).

The building blocks for the abstract stimuli descriptions are test steps. These comprise, according to their temporal order, the rows of the combination table. Such a sequence of test steps is called a test sequence. Each test step defines the inputs of the system under test over a certain time span. The time spans are listed in a separate column on the right-hand side of the combination table. The starting and finishing points of these time intervals are called synchronization points, since they synchronize the stimuli signals at the beginning and end of every test step.

The description of the values of the single stimuli signals for each test step is done by marking the class defined for this signal in the classification-tree. This is indicated in the middle part of the combination table. The stimulus signal in the relevant test step is thus reduced to the part-interval or single value of the marked class. The combination of the marked input classes of a test step determines the input of the system under test at the respective support point.

The values of the single stimuli signals between the synchronization points are described by basic signal shapes. Different signal shapes (e.g. ramp, step function, sine) are represented by different arrow types which connect two successive markings in the combination table (see Figure 16.11, left). In this way analog and value-quantized stimuli signals can be described in an abstract manner with the help of parameterized, stepwise defined functions.

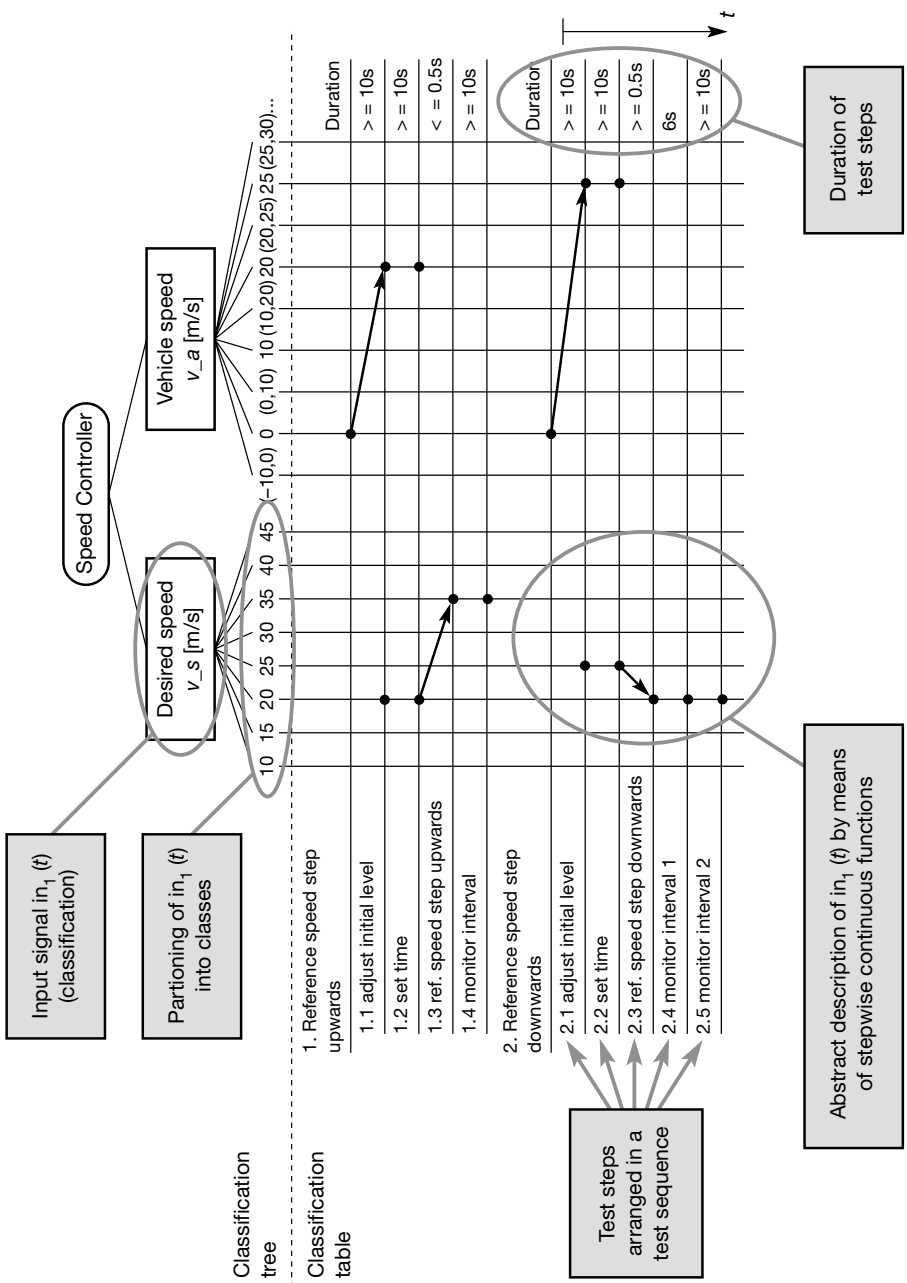


Figure 16.10
Abstract description of test scenarios with CTM/ES

The remaining degrees of freedom are removed by the selection of concrete test data. Discretization of the signal values follows so that time-quantized or digital signals are available (see Figure 16.11, right).

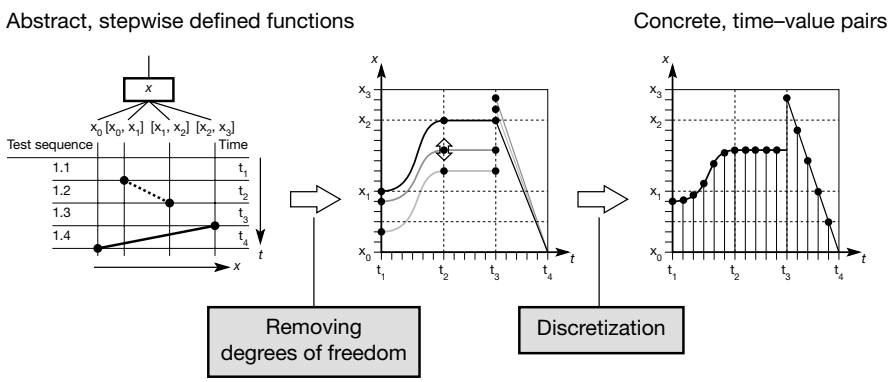


Figure 16.11 Stepwise refinement of abstract stimuli description (test sequences) into concrete signal courses

The abstract specification of the input signals in the classification-tree can also be understood as signal tubes or signal corridors in which the actual signal courses must fit (see Figure 16.11, middle). Since the stimulation of the test objects takes place time discretely, a sampling rate has to be specified with which the signal courses are discretized (see Figure 16.11, right). The result of the discretization is a concrete stimuli description by time–value pairs.

Worth mentioning is the embedding of the notation described here into a methodological framework (Grochtmann and Grimm, 1993; Conrad, 2001), and the definition of abstract coverage criteria on classification-trees (Grochtmann and Grimm, 1993; Simmes, 1997).

Commercial tool support for the classification-tree method for embedded systems is available in the form of the Classification-Tree Editor for Embedded Systems (CTE/ES) by Razorcat.

An assessment of CTM/ES is given in Table 16.4

It is also generally possible to apply the techniques mentioned for the stimuli description for specifying expected results.

1	Supported signal categories	+	All signal categories
2	Type of notation	+	Graphical, underlying textual representation
3	Degree of abstraction	+	Two different abstraction levels
4	Complexity	0	Sequencing
5	Stimuli specification under consideration of outputs	0	
6	Coverage criteria	+	Classification-tree coverage, data range coverage
7	Methodological support	+	
8.	Tool support	+	Commercially available
9	Data storage format	0	Published ASCII-based format

Table 16.4 Assessment of the CTM/ES stimuli description technique

16.3 Measurement and analysis techniques

This section discusses measurement and analysis techniques for mixed signal systems.

The comparison of signals captured at the system's output (here referred to as $f'(t)$) with expected results ($f(t)$) is one of the most frequently used techniques (see section 16.3.1). An alternative is to calculate the characteristic values (see section 16.3.2). These methods of analysis are suited to detecting certain fault categories – these can be classified as follows.

- *Functional faults*. The design is not fulfilling the requirements.
- *Statistical faults*. These include amplitude noise and noise in the time domain.
- *Systematic faults*.
  - Translations: $f'(t) = f(t+t_0)$.
    The captured signal is shifted in relation to the expected one by a defined value.
  - Stretching and compression $f'(t') = f(at)$.
    On the time axis, samples points have been added or left out.
  - Scaling $f'(t) = b \times f(t)$.
    The captured signal has been scaled by a factor of zero to infinite.
  - Modulation of an interference signal $f'(t) = f(t) + s(t)$.
    As an example, a ramp is added to the expected signal – consequently the captured signal is $f'(t) = f(t) + ct$.

A combination of several fault classes is possible. Furthermore, all faults can appear all the time (globally), or in only certain time slots (locally).

16.3.1 Comparison of expected and captured signals (tolerances)

Several techniques can be used to validate a captured signal with the help of a reference. Here, the following will be discussed:

- Static techniques
  - correlation;
  - parameter calculations.
- Non-static techniques: distance measurement
  - comparison without timing tolerance;
  - comparison including timing tolerance.

In general, when comparing a captured signal with an expected one, some defined type of difference is calculated. Due to the fact that no capturing method can guarantee an exact reproduction of a system response, tolerance definitions are needed. By doing this, a distinction between the absolute and relative error is made according to the tolerance.

$$|f(x)-f'(x)| \leq tol_y \quad \text{(absolute error)}$$

$$|f(x)-f'(x)| \leq \Delta_y \quad \text{(relative error)}$$

This simple tolerance definition, related to the value only, implies some severe problems (see Figure 16.12). In cases of dramatic changes in signal values over a small period of time, a tolerance tube defined only by the value at a certain point in time will lead to many observed faults if time tolerance is not considered as well. In other words, when small shifts in time occur, the tolerance levels are easily violated if the gradient of the signal is high.

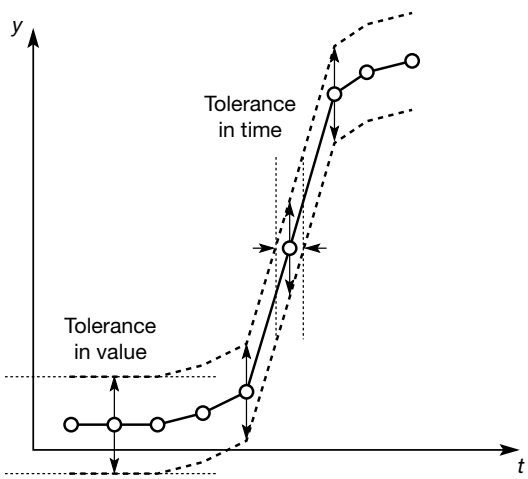


Figure 16.12 Tolerances

For this reason, signal comparison has to consider y and t tolerances in time as well as in value. Three approaches are suggested.

- Suggestion: Definitions of tolerance orthogonal to the signal.
  Problem: "Orthogonal" can only be defined in relation to the axis scale; standardization of the axis is needed; the tolerance band can produce spirals.
- Suggestion: Variable tolerance depending on the gradient.
  Problem: At the start and end of an extreme rise, the gradient is high but the tolerance tube does not need to be high as well.
- Suggestion: Definition of tolerance circles around a sample point.
  Problem: Circles might not be big enough to form a whole tube.

There are other suggestions to consider. In summary, there are no golden rules for tolerance definition but tolerance in time must be considered as well as tolerance in value.

16.3.2 Analysis of characteristic values

Instead of comparing all the sampled values of two signals, the comparison can be reduced to a few special significant points – this reduces evaluation effort significantly. The passing of a threshold, as one of such significant points, can be the indicator for a signal comparison at only one point. Some of these specific points, likely to be relevant for result evaluation, are shown in Figure 16.13:

- global and local minima and maxima;
- half amplitude values (50 percent);
- start and end points (10 percent /90 percent) of signal rise and fall.

16.3.3 Use of statistical methods

Functional evaluation based on a comparison of captured and expected results is often very expensive. An enormous number of points have to be captured and compared, and the tolerance definition is often a matter of intuition. Instead, or in addition, statistical methods are helpful. They can be applied especially if average values are relevant as opposed to the signal value at a certain point in time. Such calculations for the signal as a whole are, for example, helpful for the quality definition of audio or video signals. The following section gives a brief introduction to some of the methods used most often.

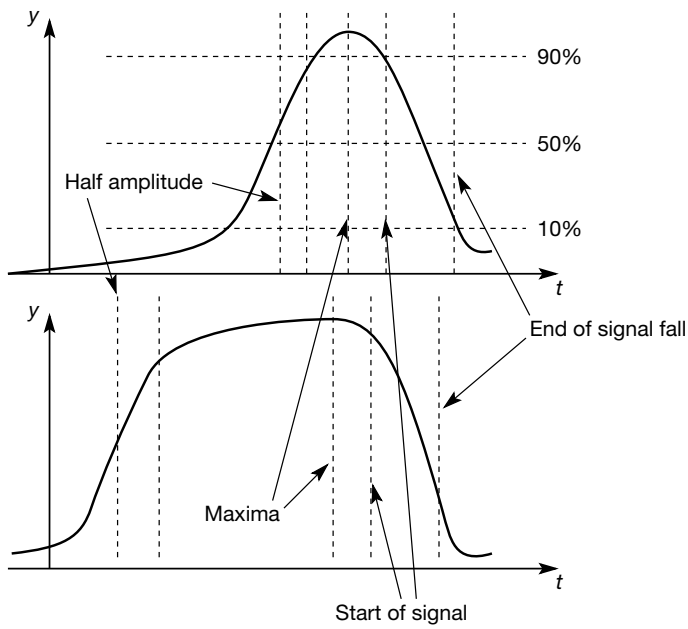


Figure 16.13
Characteristic points in signal comparison

16.3.3.1 Correlation

Correlation can help to compare the recorded signal with the expected signal by detecting similarities and shifts between those signals. In particular, cross correlation is the calculation that provides the most meaningful results:

$$z(k)=\sum_{i=0}^{N-1} x(i)\ h(k+i)$$

The reference signal is $x(i)$, the captured one $h(i)$, and $z(k)$ is the result – k is the shift between these two signals. The problem with this calculation is that it is done for the whole signal. This conceals local errors and provides validation only of the signal as a whole.

Therefore, it is useful is to separate the signals under evaluation into distinct areas. These areas should cover a period as short as is needed to get meaningful results.

Alternatively, the final summation is left out. Furthermore, no multiplication of the pairs of samples is done but a subtraction executed instead. The result is then a matrix spread out by k and i. In this matrix, the third dimension is given by the value of the difference. A digital indication (–1, 0, 1) of this difference derives from the following formula:

$$\text{temp} = \text{abs}\,[\mathit{refsignal}(i) - \mathit{simsignal}(i + \tau)]$$

$$z(\tau, i) = \begin{cases} -1 & \text{if sample is irrelevant} \\ 0 & \text{temp} > \text{tolerance} \\ 1 & \text{temp} <= \text{tolerance} \end{cases} \qquad 1 \le \tau \le 2 \times N,\ \ 1 \le i \le 3 \times N$$

16.3.3.2 Signal-to-noise ratio

The signal-to-noise ratio is given by

$$S = 20\text{dB} \times \log \frac{P_{\text{signal}}}{P_{\text{noise}}}.$$

This formula is used when the noisy signal can be captured and the wanted signal is also known. One of the application domains is A/D conversion. The signal-to-noise ratio there is an indicator of the loss of information by quantization.

16.3.3.3 Total harmonic distortion

The total harmonic distortion, *k*, is the ratio of the surface effective worth to the total effective worth, including *t*,

$$\text{k} = 100\% \times \sqrt{\frac{U_2^2 + U_3^2 + \ldots}{U_1^2 + U_2^2 + U_3^2 + \ldots}}.$$

THD is a characteristic number for the non-linear distortions of a signal. In order to capture such errors, a defined sine signal stimulates the system and the result is evaluated. The number of additional harmonic waves is an indicator of the distortion. THD is used mainly in high frequency and audio applications.

PART V

Organization

Introduction

This part of the book describes various organizational aspects of testing. Instead of technical issues, it is all about people issues. It is about those who perform all the testing activities, and their relationships and interactions with the others they have to deal with. The following topics are covered.

- *Test roles*. To perform the various test activities in a test project, specific knowledge and expertize must be available in the test organization. Chapter 17 defines the different test roles related to the test activities and the required expertise.
- *Human resource management*. Chapter 18 deals with the staffing of test projects. How to get the right people with the right qualities to fill the required test roles. It covers the acquisition of temporary personnel as well as the training of permanent test staff offering them a worthwhile career path.
- *Organization structure*. Testing supports a business by achieving certain quality-related goals. The place of a test organization in the hierarchy of larger business organizations must fit with what is expected from testing. Chapter 19 describes different ways of implementing the test organization, both in the line organization and in a project setting.
- *Test control*. Chapter 20 deals with the control of the test process and its products. The test team must be able to provide information about status and trends regarding progress and quality. This requires that staff follow certain agreed procedures.

These chapters describe the building blocks for organizing of the various test roles as well as the management and the organizational structure. Usually it is not necessary to apply all these building blocks to their full extent as described here. Depending on factors such as the test level, the size of the test object, and the culture within the organization, an optimal mixture of test roles, staff, and management will be chosen.

In the case of high-level tests (system or acceptance testing) many of the building blocks described will be needed in many instances. For low-level tests it often suffices to use a derivation of a part of these components. The wide range

of building blocks given offers the test manager the opportunity to carefully select an optimal implementation of the test organization for high-level as well as low-level tests. As always, there is the danger of selecting too much – leading to a counter-productive bureaucracy – or selecting too little – leading to lack of control and chaos. The primary goal should always be the best possible test within the limitations of time and budget.

17 Test roles

In order to have the appropriate testing personnel at the right moment, it is necessary to have a good understanding of the responsibilities and tasks involved in testing, as well as the required knowledge and skills. The various responsibilities are described in that context in this chapter. The general skills are described first. Then the various test roles and their specific skills are described.

17.1 General skills

Every member of the test team should have at least a basic knowledge of structured testing, information technology, and some general knowledge.

Testing

- General knowledge of a test lifecycle.
- The ability to review and use test design techniques.
- Knowledge of test tools.

Information technology skills

- General knowledge and experience in the field of IT.
- Broad knowledge of system development methods.
- The ability to analyze and interpret functional requirements.
- General computer skills.
- General acquaintance with hardware, software, and data communication facilities.
- Knowledge in the field of architecture and system development tools.

General

- Domain expertise and knowledge of line organization.
- Knowledge of project organization and experience of working of a project team.
- Creativity and accuracy.

17.2 Specific test roles

The following roles are described:
- test engineer;
- team leader;
- test manager;
- test policy manager;
- methodology support;
- technical support;
- domain expert;
- intermediary;
- test configuration manager;
- application integrator;
- test automation architect;
- test automation engineer;
- safety manager;
- safety engineer.

17.2.1 Test engineer

The test engineer's tasks are the most creative, but also the most labor intensive. The role of test engineer is necessary from the preparation phase through to completion.

Typical tasks

- Review of the test basis (specifications).
- Specification of logic and physical test cases, and the start situation.
- Execution of test cases (dynamic testing).
- Static testing.
- Registration of defects.
- Archiving testware.

17.2.2 Team leader

A team leader manages a small team (from two to four) of testers. The role is involving from the preparation phase through to completion.

Typical tasks

- Creating detailed plans.
- Spreading the workload across the team and monitoring progress.
- Securing facilities.
- Keeping a record of progress of the team and allocating the (time) budget.
- Reporting on progress and quality.
- Finding and implementing solutions for bottlenecks.
- Providing guidance and support for testers.
- Assessing the performance of team members.

Additional skills

- The ability to manage a project.
- The ability to motivate people.
- Good communication skills.
- Clear reporting style.

17.2.3 Test manager

The test manager is responsible for planning, management, and execution of the test process within time, budget, and quality requirements. Test management keeps a record of the progress of the test process (at project level) and the quality of the test object. The role of test management is active from the start until the end of the test process.

Typical tasks

- Creating, getting approval for, and maintaining the test plan.
- Executing the test plan according to the schedule and within the budget.
- Reporting on the progress of the test process and the quality of the test object.

Additional skills

- Experience as team leader, intermediary, and in support tasks.
- Extensive experience in managing a project.
- The ability to use planning and progress management tools.
- Excellent communication skills.
- Excellent reporting style.
- The ability to motivate people.
- A critical but open attitude.
- Tact, resistance to panic, and the ability to take criticism.
- The ability to settle conflicts and proficiency in negotiating techniques.
- Pragmatic and prepared to tackle problems.

17.2.4 Test policy manager

The test policy manager is active when testing is completely or partially embedded in the line organizational structure. "Partially embedded in the line organization structure" means anything between the role of test policy manager implemented as a full-time function and the role of test regulation included in a function containing all roles except the testing role.

The role of test regulation may be active during the whole test lifecycle.

Typical tasks

- Setting up and maintaining test regulations.
- Distributing test regulations.
- Monitoring for compliance to the test regulations.
- Monitoring for suitability of the test regulations.
- Reporting results.
- Developing new test techniques and procedures.

Additional skills

- Creative, punctual, and strictly methodical approach to work.
- The ability to motivate people.
- Good communication skills.
- Clear reporting style.
- Critical but open attitude.
- Being able to alert the organization to the importance of quality management and testing; being able to ensure, as necessary, the improvement of the respective testing activity.
- Expert at settling conflicts and negotiating, as well as being proficient in presentation, reporting, and meeting techniques.
- Pragmatic and willing to tackle problems.

17.2.5 Methodology support

The role of methodology support involves rendering assistance concerning methodology in the broadest sense. It refers to all testing techniques and all members of the testing team in addition to, for example, strategy development by the test management as well as test case design by a tester.

The methodology support role can be active during the whole test lifecycle.

Typical tasks

- Establishing testing techniques.
- Developing new testing techniques.
- Making test regulations.
- Developing relevant training courses.
- Teaching and coaching.
- Giving advice and assistance in implementation of all possible testing techniques.
- Giving advice to management.

Additional skills

- Creative, punctual, and strictly methodical approach to work.
- The ability to motivate people.
- Good communication skills.
- Clear reporting style.
- Critical but open attitude.
- Being able to alert the organization to the importance of quality management and testing; being able to ensure, as necessary, the improvement of the respective testing activity.
- Expert at settling conflicts and negotiating, as well as being proficient in presentation, reporting, and meeting techniques.
- Pragmatic and willing to tackle problems.

17.2.6 Technical support

The role of technical support involves rendering assistance concerning technology in the broadest sense. This role is responsible for making the high-quality test infrastructure continuously available to the test process. This secures the operability of test environments, test tools, and office requirements.

The technical support role can be active during the whole test lifecycle.

Typical tasks

- Establishing the test infrastructure.
- Supervising the test infrastructure.
- Physical configuration management.
- Technical troubleshooting.
- Ensuring reproducibility of the tests.
- Giving assistance and advice.

Additional skills

- Extensive knowledge of test tools.
- Advanced knowledge of supervision tools.
- Acquaintance with methods of development and construction of test environments.
- Knowledge of simulation environments.
- Pragmatic and willing to tackle problems.

17.2.7 Domain expert

The domain expert renders assistance to the test process concerning functionality in the broadest sense. The role provides support in carrying out the activities where the functionality issues can be important. These activities may include test strategy development or risk reporting by test management, as well as the testability review, or design of test cases by a tester.

The domain expert role can be active during the whole test lifecycle.

Typical tasks

Providing assistance and advice on the functionality of the test object whenever necessary.

Additional skills

- The ability to motivate people.
- Good communication skills.
- Clear reporting style.
- Specialist knowledge of the system under test, and line organization.
- The ability to bring together functional solutions and their practical application.

17.2.8 Intermediary

The intermediary is the contact (concerning defects) between the test team and other parties – they play an important part in risk reporting. The intermediary is the person who has insight into defects and solutions – they analyze all defects before they are released to other parties and have the ability not only to reject defects, but also to reject fixes.

The intermediary role is needed from the preparation phase, when testability reviewing is started, through to completion.

Typical tasks

- Sets requirements for procedures and tools for defect management.
- Collects defects and the relevant background information.
- Checks the integrity and correctness of defects; analyzes, classifies, and evaluates the categories of urgency.
- Reports the defects to the appropriate parties, using the procedures available.
- Takes part (as necessary) in communications with respect to analysis and decision taking.
- Collects relevant information from other parties about defects and problems for testers.
- Checks solutions for integrity and takes care of their controlled implementation; reports availability of solutions for retesting.
- Keeps a record of the status of defects and solutions in respect of the actual test objects.
- Renders assistance to test management in creating (on demand) risk reports.

Additional skills

- Excellent communication skills.
- Excellent reporting style.
- Critical attitude.
- The ability to settle conflicts and proficiency in negotiating techniques.

17.2.9 Test configuration manager

The test configuration manager has an administrative, logistic role. They are responsible for registration, storage, and making available all the objects within the test process that require control. In some cases, controllers carry out the duties themselves, in others they set up and manage a controlling body. This role is the same as that of a regular configuration manager only now dedicated to testing.

The allocation of operational control duties depends strongly on the environment and the type of test process (scope, project or line organization, new development, or maintenance). How deeply control tasks are embedded in the organization is certainly as important.

Typical tasks

- Progress control.
- Test documentation control.
- Keeping a record of defects and collecting statistics.
- Logical control of the test basis, and test object exposed to the test process.
- Logical control of testware.
- Premises and logistics.
- Control of the delegated supervising duties.

Additional skills

- Extensive knowledge of control tools.
- Knowledge of line organization.
- Good communication skills.
- Excellent administrative skills.

17.2.10 Application integrator

The application integrator (AI) is responsible for the integration of separate parts (programs, objects, modules, components, etc.) into a correctly functioning system. The AI reports to the development project leader (see Figure 19.1) or team leader (Figure 19.2) on the quality of the test object and the progress of the integration process, according to the plan for the integration test.

To avoid a conflict of interests, the AI will not have the role of development project leader or team leader at the same time. In this way, an area of tension is deliberately created between the AI, who is responsible for the quality, and the project leader of development, whose work is largely assessed on the basis of the functionality supplied, time, and budget.

It is highly desirable to have an AI with a development background. Compared with a high-level tester, the AI knows much more about the internal operation of the system and is better equipped to find the root cause of a defect. In comparison with other developers, the AI has more knowledge and understanding of testing and a broad view of the system.

Good contacts with developers are essential for the proper functioning of an AI. In addition, the AI is the principal contact person during later test levels.

Typical tasks

- Writing, getting approval for, and maintaining the integration test plan.
- Writing the entry criteria, with which the separate programs, objects, modules, components, etc. will comply in order to be admitted to the integration process.
- Writing the exit criteria, with which an integrated part of the system will comply in order to be approved for the following stage.
- Facilitating and supporting programmers in execution of unit tests.
- Integrating stand-alone programs, objects, modules, components, etc. into user functions or subsystems.

- Administering the integration test plan according to the planning and the budget.
- (Enforcing) configuration and release management.
- (Enforcing) internal defect management.
- Mediating with the commissioner's organization during subsequent test levels.
- Reporting about the progress of the integration process and the quality of the test object.
- Issuing release authorization.
- Assessing the integration process.

Additional skills

- Experience as a team leader, mediator, and in support tasks.
- Acquaintance with the method of system design.
- Extended knowledge of architecture and system development tools.
- Clear reporting style.
- Good communication skills.
- The ability to motivate people.
- Critical attitude.
- Tact, resistance to panic, and the ability to take criticism.

17.2.11 Test automation architect

A test suite consists of the combination of test tools, test cases, test scripts, and observed results. The test automation architect is responsible for the quality of the automated test suite as a whole, as well as for achieving the automation goals set for the respective test suite. The architect translates the goals, the approach selected, and the features of the test object into a test suite architecture. Special attention is paid to the design and construction of the suite.

Typical tasks

- Administering a quick scan for a test automation assignment.
- Writing, getting approval for, and maintaining the action plan for the construction of the test suite.
- Designing the test suite.
- Being in charge of design and construction of the test suite.
- Reporting about the progress and quality of the construction of the test suite.
- Enforcing or administering test tool selection.
- Training users in applying the automated test suite.

Additional skills

- Experience as a team leader.
- Expert knowledge of (the capacity of) test tools.
- Experience in test process automation.
- Acquaintance with structured programming.
- Good conceptual and analytical skills.

- The ability to learn about new test tools quickly.
- Clear reporting style.
- Good communication skills.
- The ability to motivate people.
- Critical attitude.

17.2.12 Test automation engineer

The test automation engineer is responsible for the technical side of the automated test suite. The engineer translates the test suite design into the modules to be constructed within the test tool. They construct the test suite by means of new and existing test tools. Within the framework of the test automation architect's directions, the engineer administers the detailed design, construction, and implementation of test suites.

Typical tasks

- Constructing, testing, supervising, and maintaining a test suite.
- Providing support in designing a test suite.

Additional skills

- Acquaintance with test process automation.
- Experience in structured programming (preferably in the programming language of the tool).
- Good analytical skills.
- Creativity and punctuality.
- The ability to learn about new test tools quickly.
- Critical attitude.

17.2.13 Safety manager

The safety manager is responsible for establishing the safety project and recruiting the right people for it. They have responsibility for the awareness of safety and the initiation of the safety program. They have to convince project members of the importance of safety and the responsibilities of the project members for this.

Typical Tasks

- Preparation of the safety plan.
- Establishing project organization.
- Project management.
- Consulting the certification institution, government (if necessary), the commissioner, and contractors.

Knowledge and skills

A minimum of a degree in an engineering or science subject related to the scope of application of the safety program. Formal training in safety engineering would be highly desirable.

17.2.14 Safety engineer

Most of the practical safety activities are part of the role of safety engineer. In addition to this, the safety engineer organizes and leads regular project safety committee meetings.

Typical tasks

- Determining risk classification.
- Determining risks and classifying them.
- Supervising the analysis.
- Maintaining a safety log.
- Advising about the impact of change requests on safety.
- Reviewing the test plan.
- Reviewing defects.

Knowledge and skills

A minimum of a degree in an engineering or science subject related to the scope of application of the safety program. Formal training in safety engineering would be highly desirable.

18 Human resource management

Testing requires a great variety of expertise. Chapter 17 lists the distinct roles and tasks, as well as the required knowledge and skills. This chapter examines the human resource and training aspects that are important in this context. Test management is responsible for appointing the right person to the right position, preferably in close consultation with the HRM department. Careful recruitment and training of testing staff is required. Suitable and experienced testing personnel are scarce. Offering a career perspective to professional testers may be a solution.

18.1 Staff

It is important to have the proper combination of expertise in a test organization. Testing requires expertise in the fields of:

- the developed system;
- the infrastructure (test environment, developing platform, test tools);
- testing;
- test management.

The low-level testers are usually developers and the required test expertise is primarily to do with the developed system and the development infrastructure. High-level testing involves developers, users, managers, and/or testers – the required expertise is shifted more towards domain expertise and test knowledge.

In addition to the usual problems relating to personnel, the provision of staff for testing is characterized by a few peculiarities which must be considered during selection, introduction, and training.

- *Incorrect idea of testing, and hence also of the necessary testing staff.* Testing is usually carried out alongside regular activities, and designers or users temporarily take on the task of testing. In practice, there is insufficient test expertise in the test team. Few organizations value test functions equally highly as comparable development functions.

- *Testing requires a particular attitude and special social skills*. Testers set out to (dis)approve of their colleagues' work after they have pushed themselves to provide the best possible product. A more subtle way of expressing their "approval" is for testers to enable others to add quality to their product. Thus testing also requires tact.
- *Testers must be thick-skinned because they are criticized from all sides*.
  - they overlook defects (because 100 percent testing is impossible);
  - testers are nit-pickers;
  - testers stay in the critical stage too long – testing takes too long;
  - the test process is too expensive – it could be cheaper and (especially) involve fewer staff;
  - testers are too emphatically present – we do not construct in order to test;
  - when a system shows a defect, it is often the testers who are blamed; when a system functions well, it is the system developers who receive praise;
  - not all tests are completed and test advice is often ignored – a tester must be capable of dealing with this;
  - testers are often inexperienced because (structured) testing is a new service to them;
  - testers must be talented in the field of improvisation and innovation – test plans are usually thwarted by delays incurred in previous processes.
- *Difficulty of planning the recruitment and outflow of testers*. The start, progress, and completion of test processes are difficult to predict. A test process is characterized by underutilization, extra work, and unexpected peaks in the workload. The suppliers of human resources want to know long beforehand who should be available and at what moment in time. This causes conflict, disappointment, decline in quality, and a planning void.
- *Test training is regarded as superfluous*. Testing is often looked on as something that anyone can do.

The test manager would be wise to engage a professional in the field of personnel management. Experience has shown that personnel officials are happy to help, certainly when it involves role descriptions and training demands as described in this book. Ensure that there is good insight into the needs, both with regard to the jobs and the planning (of the inflow of staff). If external expertise is required, which is regularly the case in test processes, there must be insight into the available budget.

18.2 Training

Great demands are made of testers. They are expected to have wide-ranging competence (IT knowledge in general, and social skills) as well as thorough knowledge (test methods and techniques). Training can be helpful here. It should create a bridge between the demands and the knowledge and skills available among the selected testers. A training program for testers could include a large number of topics some of which are listed here.

Test specific

- Test methods and techniques:
  - importance of (structured) testing;
  - test lifecycle model, planning;
  - test design techniques;
  - techniques for reviews and inspections.
- Test environment and test tools:
  - automation of the test process;
  - test tools.
- Test management and control:
  - staff and training aspects;
  - test planning, test budget;
  - techniques for risk assessment and strategy development;
  - control procedures;
  - improving the test process.
- Reporting and documentation techniques.
- Quality management:
  - quality terminology (ISO9000);
  - standards;
  - auditing.
- Test attitude, i.e. social skills.

General

- General knowledge of IT:
  - policy, planning;
  - design strategy.
- Methods and tools for system development:
  - lifecycle models;
  - modeling techniques;
  - CASE, OO, 4GL, workbenches, and component-based development.
- Planning and progress monitoring techniques:
  - planning tools.
- Good communication, and personality:
  - reporting and presentation;
  - social skills;

- Technical infrastructure:
  - hardware – target platforms, PCs, etc.;
  - software – operating systems, middleware;
  - communication – servers, LANs, WANs;
  - GUIs, client/server, internet, etc.
- Domain expertise and organizational knowledge:
  - functionality;
  - organization structure;
  - business objectives.

It is advisable to have the test training course constructed and made available in a modular fashion. Depending on the role and the employee's expertise, training can then be carried out more or less individually per (sub)topic. The training structure is illustrated in Figure 18.1.

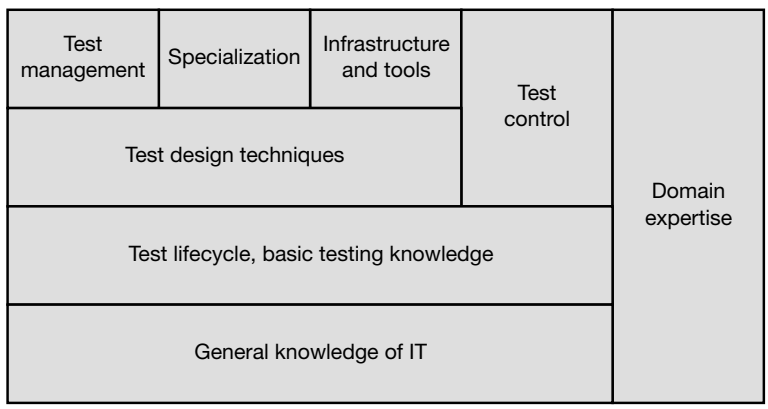


Figure 18.1 Structure of test training courses

One usually has to rely on third parties for training courses. Hence, the requirements with respect to modularity and training strategy indicated here are, in fact, requirements for the training institute.

Independent certification of a tester is performed by the ISEB (Information Systems Examinations Board) a section of the British Computer Society. ISEB offers three certificates (of increasing difficulty) for testers.

18.3 Career perspectives

18.3.1 Introduction

To raise the professional status of testing in an organization, a career perspective must be offered. This section describes a methodology for the specification of test career paths. Essential components include training, gaining work experience, and a supervision program. These are shown by function and by level in

the so-called career cube (see Figure 18.2). The cube is a tool for career guidance used by the human resource manager in order to match the available knowledge and skills, and the ambition of the professional testers, in accordance with the requirements from within the organization. The three dimensions of a career cube are defined as follows:

- height – functional growth;
- width – functional differentiation;
- depth – knowledge and skills.

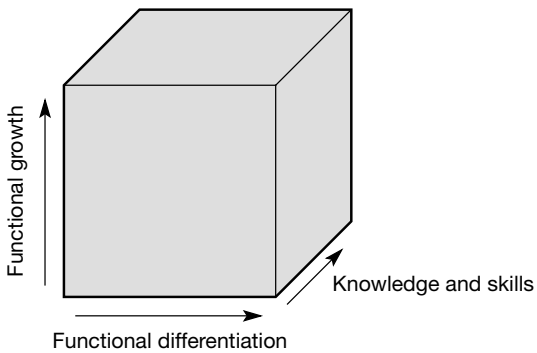


Figure 18.2
Dimensions of the career cube

18.3.2 Functional growth

Functional growth is the function progress that an employee makes, for example from tester to test manager (see Figure 18.3). A distinction is made here between vertical growth and horizontal growth. When further vertical growth is no longer an option, there is still the possibility of horizontal growth. Vertical growth means a higher function level; horizontal growth means reaching a higher performance level within the same function level. Improvement in conditions of employment can be achieved in this structure by reaching either a higher function level or a higher performance level.

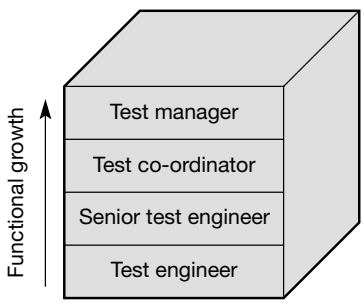


Figure 18.3
Functional growth

18.3.3 Functional differentiation

Three differentiations are distinguished (see Figure 18.4).

- *Team and project leadership*. Employees who are capable of, and interested in, leading a test project choose this direction.
- *Methodological advice*. This direction is meant for employees who want to give test advice (for example when a test strategy is being established) in the selection of test design techniques, or organizing control.
- *Technical advice*. Individuals who feel attracted to the technical side of testing opt for technical (test) advice. This includes the organization and utilization of test tools or the setup and management of a test infrastructure.

Figure 18.4
Functional differentiation

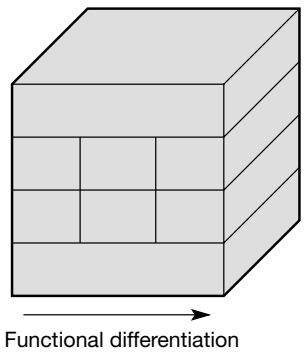


Figure 18.5 shows examples of function levels with the function name indicated for each direction. At level A, there is no clear demarcation into columns. At this level, staff gain a wide knowledge of test execution in all its aspects. At function levels B and C, differentiation applies. At function level D, differentiation again does not apply. Employees at this level are expected to be capable of implementing successfully, and/or managing, each of the three differentiations.

Figure 18.5
Examples of function levels

D	General test management		
C	Test project leadership	Methodological test advice	Technical test advice
B	Test team leadership	Methodological test specialization	Technical test specialization
A	Test execution		

18.3.4 Knowledge and skills

Building up knowledge and skills is represented by the third dimension of the career cube. The following components are distinguished:

- (test) training
- social skills
- experience demands
- coaching and support.

Coaching and support are essential for a sound career path, in particular in the case of (a start in) one of the lower function levels. Each employee receives support in various ways. This is implemented by the test manager, the personnel official, and/or experienced colleagues. Testers who have just started receive extra attention from a coach, who supervises all aspects and developments.

As the function level increases, the importance of practical experience and social skills training also increases considerably in relation to (test) training and coaching and support.

The coherence between the required knowledge and skills is specified in greater detail in Figure 18.6.

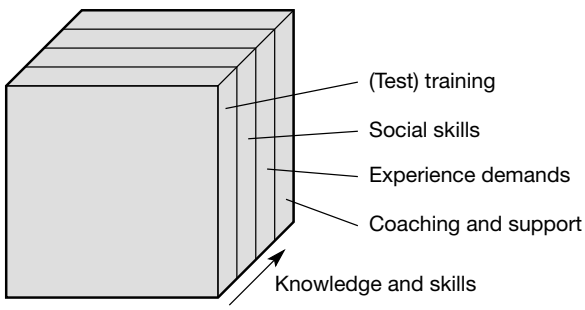


Figure 18.6
Knowledge and skills

Finally, the full career cube is completed on the basis of the above (see Figure 18.7). It shows the required knowledge and skills for each function (differentiation).

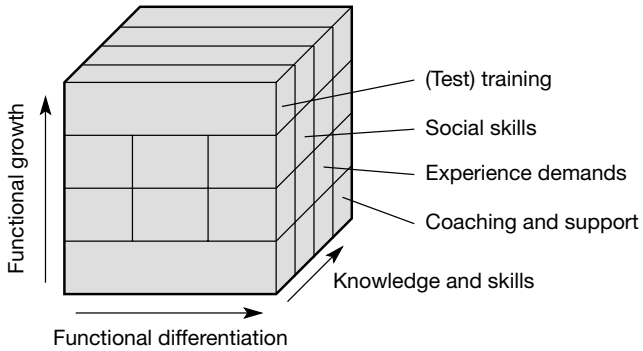


Figure 18.7
Complete career cube

Organization structure 19

In an organization the responsibilities, tasks, hierarchy, and communication structure should be clear. This should be realized by identifying the required test roles and selecting the right people for these. The communication structure should be made clear by establishing consultation and reporting structures inside the test organization and with stakeholders outside.

19.1 Test organization

Low-level tests are carried out by developers most of the time. Therefore, it is not very useful to organize low-level testing in a different manner to development.

An independent test organization carries out high-level tests most of the time, and can be organized either as a project or line activity. If products are released on a regular basis, it is useful to build up testing in the form of a line organization. It offers the opportunity to build up expertise and knowledge about testing, to gain more experience, and offers the test team members a career perspective. It also gives the opportunity to build up the necessary infrastructure and the expertise to maintain it. This is very important if the infrastructure is very specific and expensive. For unique products (i.e. one-shot systems) the project form is the best way to organize testing. The organization structure built up is then dedicated to the project.

19.1.1 Project organization

Most systems are developed as projects. Even if development and testing are line departments, projects are often started for the development of a certain product. Project organization has the following advantages:

- a clear goal (the realization of a product);
- the organization is dedicated to that goal;
- staff are recruited with the special skills necessary for the realization of the product;
- because the focus is on just one goal, management and control is easier.

Disadvantages are:

- many things have to be started from scratch, for example build-up of infrastructure, introducing standards and procedures, etc.;
- if staff leave at the end of the project (when the product is in the acceptance test) it is difficult to get them back if needed. In this stage there is not enough work for everyone, but in allowing staff to leave knowledge leaks away – this can have serious consequences for solving defects.

Many disadvantages can be avoided by starting projects with a line organization (see section 19.1.2) although this is not always possible.

In Figure 19.1 a project organization is visualized with a separate team for high-level testing. The development team carries out the low-level tests.

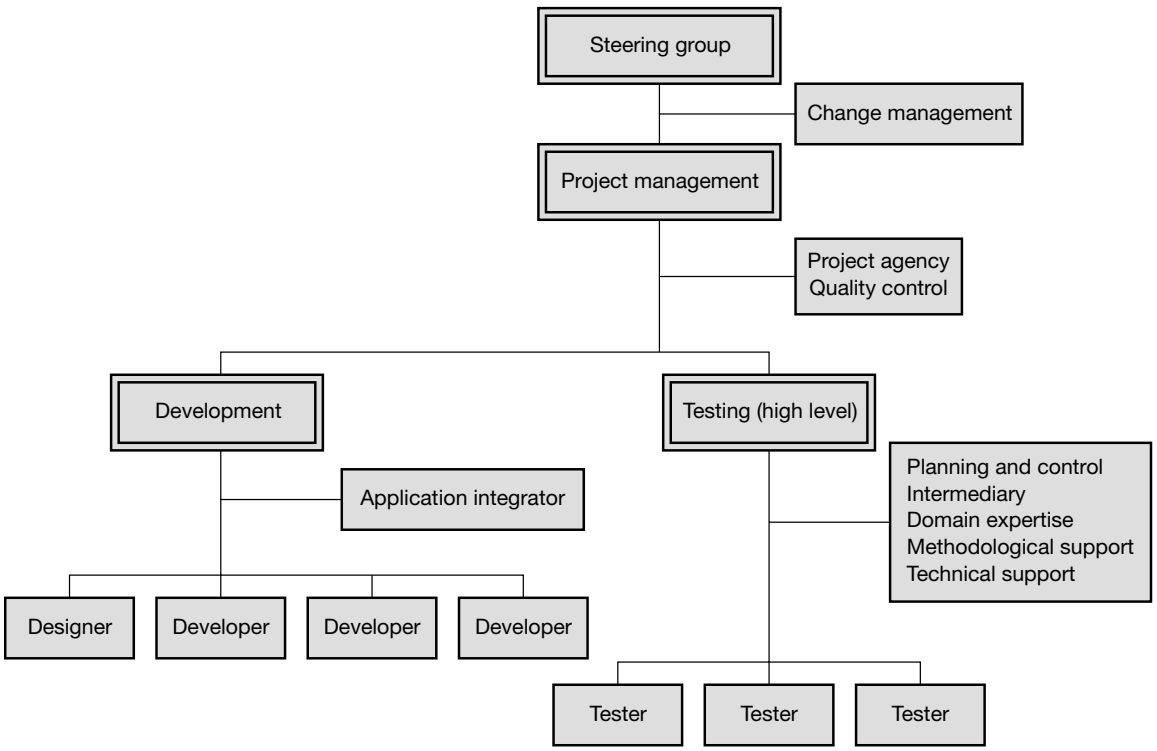


Figure 19.1 Example of a project organization

The development department executes the unit and integration test under the responsibility of the application integrator, who is responsible for reporting to the development leader on the quality status of the product. The exit criteria defined for the integration test must be the same as those for the entry criteria, defined by the test team occupied with the high-level tests. They have to accept the product officially for the high-level tests. The status reports of all the tests (low- and high-level) are subject to quality control.

The development leader has the responsibility for delivering in time and budget with a certain quality. In practice it is very difficult to realize these three objectives together. Reports have to be made regularly to the project management. If there are serious problems it is raised to the level of the steering committee – this consists of the commissioner, project leader, development leader, test leader, and other stakeholders. They ultimately decide if the product is good enough to start high-level tests, based on reports written by the application integrator. It is important that they don't approve too soon because defects are easier and cheaper to correct during low-level testing. Besides, the start of the high-level tests often coincides with the start in the reduction of developers. During high-level testing full capacity is no longer needed, although some developers are still needed to correct defects.

At the end of the high-level tests the steering group once again uses the reports of the test leader to decide whether to go into production and consequently dismantle the project organization. It should be noted that the steering group meets regularly to monitor progress and, when necessary, to take appropriate measures to ensure the progress of the project.

19.1.2 Line organization

Many companies are organized to produce new or changed products on a regular basis. Product development is then one of the departments of the organization. Within product development there are separate development and test departments. The test department is concerned with high-level testing, while the development department looks after low-level test activities. Although this sounds like line organization, in practice the development and testing of the products is carried out as a project. The line organization supports the project by delivering the right people, procedures, etc. Under this construction, many supportive tasks can be centralized and organized efficiently. The infrastructure, for example, can be used by the test and development teams for different projects and can be maintained centrally. The same is true for tool and change management, and the test automation architecture. Within the projects, staff can focus on developing and testing the new product – people outside the teams take care of providing and maintaining the right equipment.

In Figure 19.2, the projects are visualized by the different teams within the system development and test department. Depending on the culture of the organization, these teams are more or less autonomous with their own budget, staff, and goals. Or they may be under strict control of the department leader.

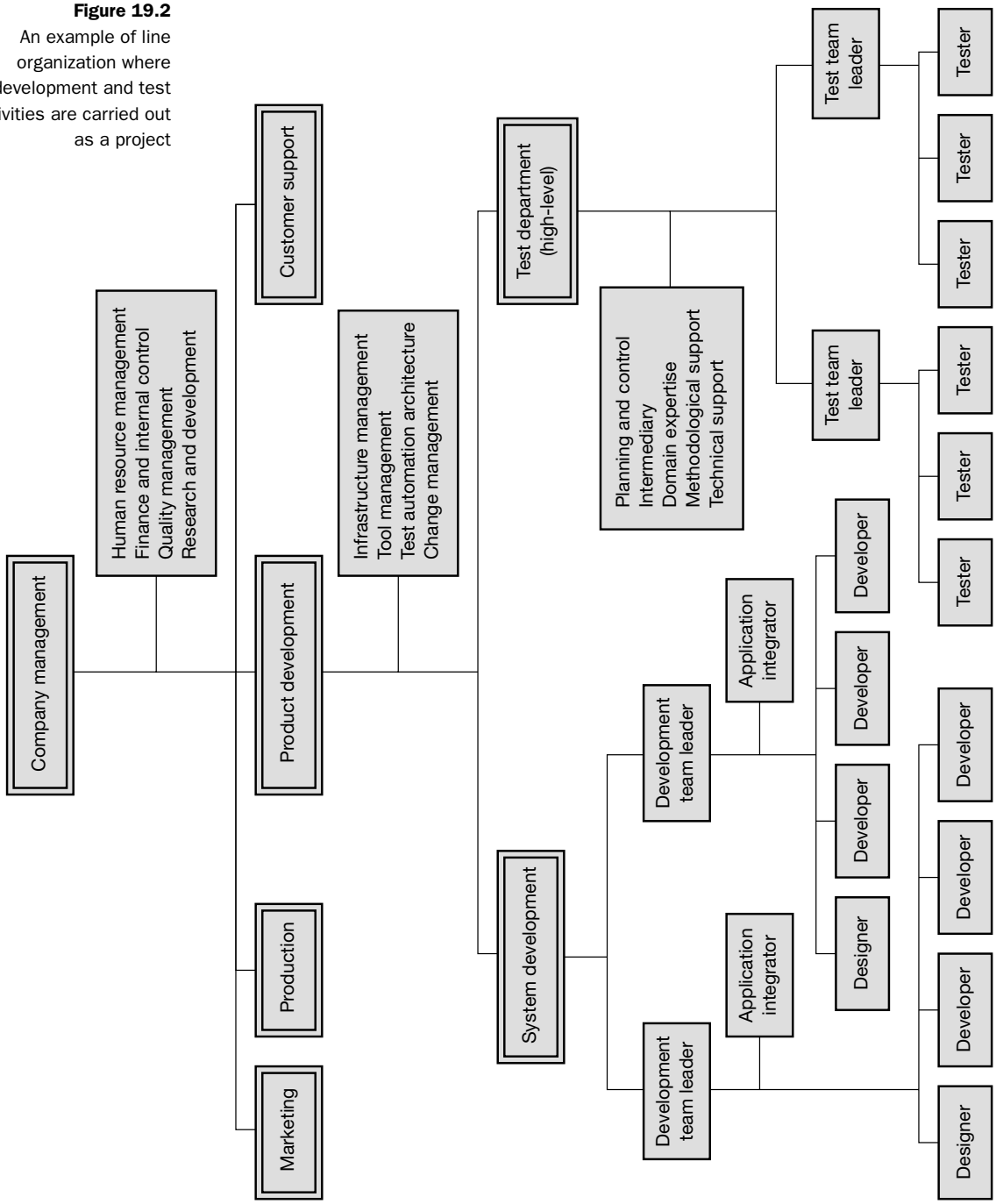


Figure 19.2
An example of line organization where development and test activities are carried out as a project

19.2 Communication structures

The company management gives approval for the development of a certain product. Before this decision can be made, much information has to be provided about development, production, marketing and support costs, time needed for development, and profitability of the product. This section deals only with the role of testing in this process from idea to product realization.

A rough estimation about the development costs, including testing, is usually made without consultating the test department. Based on experience, a certain percentage of the total development time, which includes low-level testing, is reserved for high-level testing.

At the start of the project, product development forms a steering group with representation from marketing, production, customer support, commissioner, and other stakeholders. The steering group is the decision-making forum for all manner of subjects concerning the progress of the project. This group meets on a regular basis and whenever necessary. The steering group is already confronted with testing at the start of the project when the test strategy is developed. The leader of the test department leads this process. The test strategy, together with the allocated budget and time, form the basis for the master test plan. In this process, the development team leader or development department leader is involved because decisions are being made about what should be tested at which test level.

The application integrator can make the detailed test plan(s) for the low-level tests and at the same time the test team leader prepares detailed test plan(s) for the high-level tests. The test team leader is in contact with the development team because they have to provide the test basis and knowledge about the test object. The application integrator has to define exit criteria for the integration test. These criteria must not conflict with the entry criteria defined for the high-level tests. These criteria should already be defined for high-level testing in the master test plan, otherwise the test team leader and the application integrator have to discuss it. It is their responsibility that these criteria are consistent.

The whole communication structure for testing is shown in Figure 19.3.

The low-level tests are carried out by the development team. In this case, only communication to product development about the progress is required. The progress reports are subject to central quality control. If development is finished (i.e. the integration test meets its exit criteria) the steering group decides, based on the reports of the application integrator, if the test object can be handed over to the test team for the high-level tests. This is not the formal start of high-level testing, because it is wise to check that the test object is functioning in the test environment. The test object is checked to see that the main functionality is OK and to assess whether it is sufficiently testable (see also section 6.5.1). Then the steering group gives official approval to start the high-level testing.

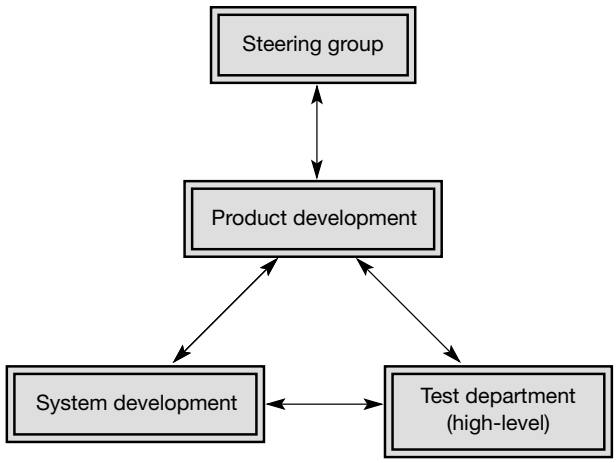


Figure 19.3
Communication structure for testing

The steering group is also the ultimate forum where incident reports are discussed and decisions made as to what to do about the incidents. Sometimes a smaller and "lighter" intermediate forum is formed to filter the difficult incidents – the test team leader and the development leader discuss what to do with each incident and only when they cannot agree is the incident raised to the level of the steering group (see also section 20.3.4).

The ultimate task of the steering group is to sanction product release, based on the test reports.

Although formal communication and reporting structures must exist, informal communication is no less important. Informal contacts make life much easier and help to build an atmosphere of respect and understanding. Testing, in practice, turns out to be a process depending on much communication.

20 Test control

In a test project, many activities are performed which produce the test products (defects, test designs, test databases, etc.) within limited time and resources. To avoid chaos, control must be established over the process and its products.

In an increasing number of organizations, *proof* that testing has been carried out must be provided in explicit reports. Furthermore, it is demanded that proof is provided that defects have been dealt with. These demands are expected to increase as a consequence of government regulations. Test control should be set up in such a way as to meet the strict demands of traceability and burden of proof. This means that:

- test designs clearly indicate from which part of the test basis they have been derived;
- during test execution, proof is provided about which test cases have actually been executed;
- insight is provided into which test cases resulted in which defects;
- during retesting it is recorded which defects have been proven to be solved.

Test control can be divided as follows:

- control of the test process;
- control of the test infrastructure;
- control of the test deliverables.

They will be discussed in more detail in the following sections.

20.1 Control of the test process

The basis of process control is gathering data, drawing conclusions, and initiating corrective actions. For the control of the test process, information is required about the use of *time* and *budget* in the test project, and the *quality* of the test object. For both, the information must show:

- *current status* – providing the basis for deciding if the *product* is good enough (or, in general, if the business can still meet its goals);
- *trends* – showing problems in the development and test *processes* and providing the basis for deciding if they need to be changed.

Periodically and (ad hoc) on request, the test manager reports on the status and trends of time, budget, and quality. The frequency and form of reporting should be agreed and described in the test plan. In addition to the numerical presentation, the periodic report should also offer written insight into the activities, deliverables, trends, and possible bottlenecks.

It is important to report periodically to the commissioner from the beginning of the test process – from the moment when the test plan is committed. During the preparation and specification phases, management is inclined to be indifferent towards testing. There is interest only in progress during the execution of the tests, when testing is on the critical path of the project.

Trends are discovered during all phases of the test process. It is of importance to evaluate and report on these at the earliest possible stage. This allows corrective measures to be taken in time – this is almost always better, cheaper, and faster.

The issues of time and budget in the test project and the quality of the test object are discussed in more detail next.

20.1.1 Progress and utilization of time and budget

During all phases of the testing lifecycle, the progress of the test project overall and on a detailed level must be monitored. If necessary, timely measures must be taken to correct negative trends.

Monitoring progress is based on data collected throughout the test process. To this end, the activities and deliverables in test planning are related to time and resources. Within this framework, information can be collected in order to gain insight into the degree to which the test planning has been effected.

For a test process, the activities and deliverables (derived from the lifecycle model) shown in Table 20.1 can be used as a basis for progress monitoring.

Table 20.1 Activities and deliverables for monitoring progress

Activity	Deliverable
Review of the test basis	Test basis defects
Test design	Test cases/scripts/scenarios
Setting up test databases	Test databases
Test execution	Test results
Comparing and analyzing results	Defects
Archiving	Testware

The progress is indicated by the degree of completion of the test deliverables. The basic data that must be registered for each activity is:

- progress status (not started, busy, on-hold, percentage completed, completed);
- time and resources (planned, spent, remaining).

Many indicators can be derived from these that help to monitor whether the project is still on track or not. This belongs to project management in general and is not specific to testing – the reader is encouraged to consult text books on project management to learn more. Some examples of popular indicators are given in the following list.

- *Time (planned, spent, remaining) per activity type*. For instance testability review, design test cases, first test execution, retest, and overhead activities. When the amount spent plus the amount remaining deviates from the planned amount, corrective action is required, such as updating the schedule, changing activities, or changing staff.
- *Cost of retests per test unit*. When this amount is high, the test project is in danger of running out of time and budget due to insufficient quality of the test object. Testing is not to be blamed for the extra time required. The test manager should ask the questions "Do you want us to perform these unplanned extra retests? Then provide more test budget. Or do you want us to stick to the plan? Then we must stop further retesting."
- *Time or budget per test role*. For instance test management, testing, technology support, and methodology support. This says something about the required composition of the test organization and is especially useful for planning future test projects.
- *Time or budget per test phase*. This is related to the lifecycle used – for instance planning and control, preparation, specification, execution, and completion. This information is especially useful in estimating future test projects.

When registering progress, one can differentiate between primary activities and management support activities. For reasons of efficiency, it is recommended that all small activities are not registered explicitly. Their usefulness should be reviewed periodically (in consultation with the commissioner). For example, separate registration of management and support activities is useful if the time needed is significant (more than one day per week).

The commissioner or project managers must be offered structural insight into the progress of the test process. It is important to be able to report on progress at the level of deliverables of the test process. It is recommended that a written report complements the numerical presentation.

Registration of time and budget should not be done without tools. Test managers can, of course, develop their own spreadsheets but a wide range of commercial tools is available. In most cases, test managers can use the same time registration system as other departments in the organization – this is often required by the organization.

20.1.2 Quality indicators

During the whole test process, the test team provides information on the quality of the test object. This information is based on the defects found during the execution of test activities. It is used for periodic and ad hoc reporting, and for the compilation of a formal release advice. When reporting on quality, there is always a danger of starting fruitless discussions based on personal preferences about what constitutes acceptable quality. To facilitate a constructive follow-up to a quality report, it must contain the following parts, forming a logical chain of thought:

- *Objective facts*. This typically consists of the measurement data relating to defects and time. The value of these measurements should not be subject to discussion.
- *Subjective conclusions*. The test manager draws conclusions about the state of affairs, founded on the objective facts previously reported. Other people may disagree with these therefore the test manager must carefully formulate the conclusions and support them with facts.
- *Recommendations*. When the conclusions are agreed by (most of) management, then possible corrective actions must be initiated to solve the observed problems and risks. The test manager can help the organization by proposing certain solutions, often of the following kinds:
  - *change in schedule* – the release will be postponed to allow extra time to solve defects; sometimes the opposite can be proposed, when testing shows very few or just minor defects;
  - *change in release content* – the product can perhaps be released on the planned date with reduced functionality;
  - *change in test strategy* – extra tests will be performed on vulnerable parts of the system, or test effort will be reduced for areas that have shown very few or just minor defects.

Defects can be found in the test basis, the test object, the infrastructure that was used, and the test procedures. Section 20.3.4 describes in more detail the procedures for recording and managing defects and which data to record. The set of data to be collected and the quality indicators derived from them is agreed, preferably beforehand in the test plan.

The following quality indicators can be useful *during the test project* to monitor what's going on and to initiate corrective action.

- *Number of still open defects per severity category at a certain time*. This could be called a "snapshot of the quality of the product." It is easy to measure and provides immediate information about the current quality of the product.
- *Stability of the product release*. This is an indication as to whether the product is no longer subject to change. If many changes still need to be made to the

product, confidence in its quality is low. A simple measure that can be used for this quality indicator is

$$\text{number of defects found} + \text{number of defects solved}$$

measured over a certain time period and per severity category.

- *Number of defects found per man-hour of testing*. This is an indication of the length of time it takes to find one more defect. For instance, a number of "1.4 defects of high severity per man-hour" would translate into "With a team of three testers working 30 hours per week on average, we would still find $3 \times 30 \times 1.4 = 126$ severe defects next week". To draw these kind of conclusions, it is better to measure only the time spent on actual *execution* of test cases.
- *Number of retests per product component (test unit)*. This is an indication of how many tries it takes to deliver a component of sufficient quality. It can be an indication of the maintainability of the product or the quality of the development process.
- *Turnaround time per defect*. This is the time between the recording of a defect and when the solution to that defect is delivered. As with the previous indicator, it may say something about the maintainability of the product or the quality of the development process.
- *Number of defects per cause of the defect*. The causes of defects can be categorized – for instance test basis, test object, infrastructure, test design, and test procedure. This indicator points out where corrective action is most effective.

The first three of these indicators are especially suited for use as acceptance criteria. If any of these is significantly above zero, it seems unwise to stop testing. It is relatively easy to represent these indicators graphically over time. This provides a clear and simple picture for management and, if all goes well, the graphs for all three should move towards the zero baseline. Examples are shown in Figures 20.1 and 20.2.

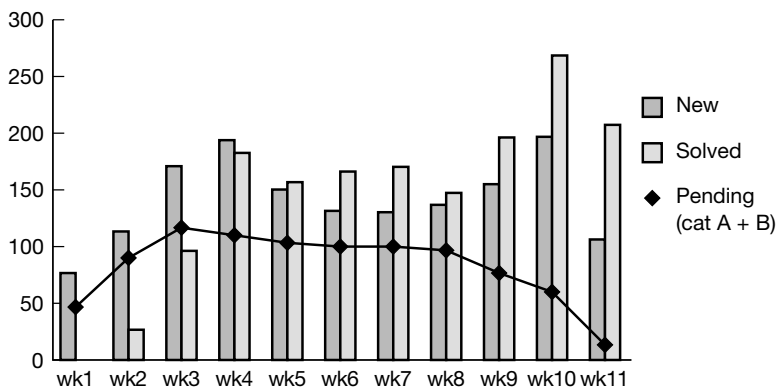


Figure 20.1
Graphical representation of the number of pending defects (line) and the stability of the product (sum of the bars)

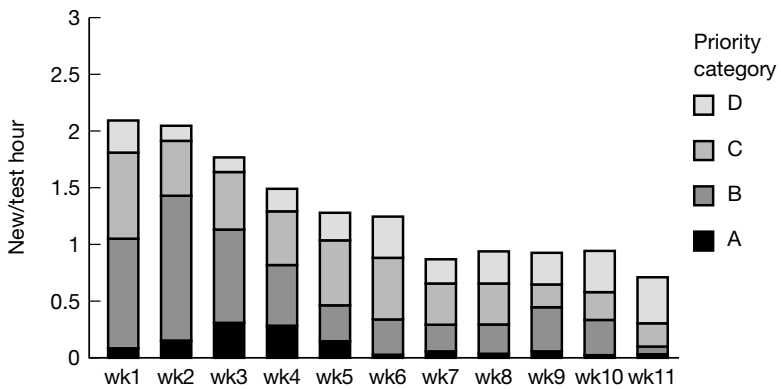


Figure 20.2
Graphical representation of the test productivity

The following quality indicators are especially useful at the *end* of the test project for analyzing the quality of the test object and the development and test processes.

- *Distribution of defects over the test units (or components)*. This identifies the weak parts of the product.
- *Test defects per type*. For instance the following types can be distinguished: test design error, execution error, result evaluation error, and environment error. This gives an indication which testing skills or procedures need to be improved in the team.
- *Defects per unit size of the product*. This indicator compares the quality of this product (release) with the quality of other products (releases). To measure this, the organization must decide on how to measure the size of the product. Examples of such a measure are lines of code, function points, and number of classes.

In addition to these, the quality indicators previously mentioned that are measured during the test project can be used as well. They can serve as historical data for the purpose of estimating and planning future test projects and for the evaluation of future test projects and products.

20.2 Control of the test infrastructure

During the early phases of the test project, the test infrastructure is specified and ordered. After installation and acceptance, the test management is responsible for controlling the test infrastructure.

The test infrastructure can be divided into the facilities:

- test environment;
- test tools.

With respect to the division of control tasks, the various aspects of the test infrastructure can be divided into two groups.

- Technical control:
  - test environment (hardware, software, control procedures);
  - test files (physical);
  - networks for the test environment and the office environment;
  - technical office organization;
  - test tools.
- Logistical control – the non-technical component of the office environment such as:
  - canteen facilities;
  - transport;
  - admission passes.

The tasks to be carried out in the framework of logistical control are part of the control and/or facilitative services support roles. These tasks are not discussed in this book.

The technical control tasks to be carried out are part of the technical support role. When these tasks are carried out, the supplier or system management may offer support. The most important control tasks of the technical support role are:

- organization;
- provision;
- maintenance;
- version management;
- troubleshooting.

These tasks are described in more detail in section 17.2.6.

20.2.1 Test environments and test tools changes

During the project, the infrastructure is subject to changes because of a variety of internal and external causes, including:

- phased delivery and changes in the infrastructure;
- delivery or redelivery of (parts of) the test object;
- new or changed procedures;
- changes in simulation and system software;

- changes to hardware, protocols, parameters, etc.;
- new or changed test tools;
- changes in test files, tables, etc.:
  - converting test input files into a new format;
  - reorganization of test files;
  - changes in naming convention.

Changes in the technical infrastructure may only be introduced after permission from the test management (possibly delegated to the technical support official). The manager registers the changes with the aid of the test control procedure. The test team will be informed immediately of any changes that will effect its work.

20.2.2 Availability of test environment and test tools

The technical support officer is responsible for keeping the infrastructure operational. The infrastructure and recovery procedures are maintained by creating backups so that the environment can be restored to a recent sound state.

The tester must report defects in the test environment and test tools to the technical support officer who will then analyze the defect and, depending on the severity, correct it immediately. For those defects that cannot be corrected immediately, the tester must write a formal defect report.

The technical support staff provide data, periodically and on request, for reports on the availability of the test environment and test tools, and (possibly) the progress and duration of the defect that has occurred.

20.3 Control of the test deliverables

In this process, external and internal deliverables are distinguished. External deliverables include the test basis and the test object. Internal deliverables include testware, the test basis (internal), the test object (internal), the test documentation, and test regulations.

20.3.1 External deliverables

Control of external deliverables is an external responsibility, although it is often the test project that suffers most from a lack of external control of the test basis and object. It is therefore important to set demands as a test team in relation to these procedures.

20.3.2 Internal deliverables

Testware

Testware is defined as all test deliverables produced during the test process. One of the requirements is that these products must be used for maintenance purposes and must therefore be transferable and maintainable. Testware includes the following:

- test plan(s);
- logical test design;
- test scripts;
- test scenario;
- traceability matrix (showing which test cases covering which requirements);
- test output;
- test results (including defects) and reports.

Test basis (internal)

The documentation marked as the test basis is saved in its original form. Control provides duplicates for the test activities and registers the assigned version numbers, etc.

Test object (internal)

The software and hardware are controlled by procedures and other means available within the organization. The documentation is managed in the same way as the test basis.

Test documentation

During the test process various documents are received or written that are meant only for this particular project. In principle, they have no value after completion of the test project, apart from being a recorded history of the project. This kind of documentation includes:

- project plans (other than test plans);
- reports from meetings;
- correspondence;
- memos;
- standards and guidelines;
- test, review, and audit reports;
- progress and quality reports.

The control support role can assist in proper archiving and retrieval of test documentation.

20.3.3 Test deliverables control procedure

The aim of this procedure is to manage the internal deliverables. The tasks to be carried out in this framework are part of the control support role. The specific control of changes in the test regulations is often a task for the methodological support official. In this procedure, the term "deliverables" is used for all products delivered during the test process.

The test deliverables control procedure consists of four steps:

- *Delivery*. The testers supply the deliverables that must be controlled to the controller. A deliverable must be supplied as a whole, for example it should contain a version date and a version number. The controller checks for completeness.

- *Registration*. The controller registers the deliverables in the administration on the basis of data such as the name of the supplier, name of the deliverables, date, and version number. When registering changed deliverables, the controller should ensure consistency between the various deliverables.
- *Archiving*. A distinction is made between new and changed deliverables. Broadly speaking, this means that new deliverables are added to the archive and changed deliverables replace previous versions.
- *Distributing*. The controller distributes copies of the deliverables at the request of project team members. The controller registers the version of the deliverables that has been passed on, along with to whom and when.

20.3.4 Defect management

Through defect management, a test manager is able to keep track of the status of defects. This will help the report about the progress and quality of the product tested so far. Defect management helps the leader of the development team to assign defects to individual developers and to watch the progress and workload across the team. A distinction is made between internal and external defects. Internal defects are those caused by errors made by the test team whereas external defects are those in the test basis, test object, test environment, and tools.

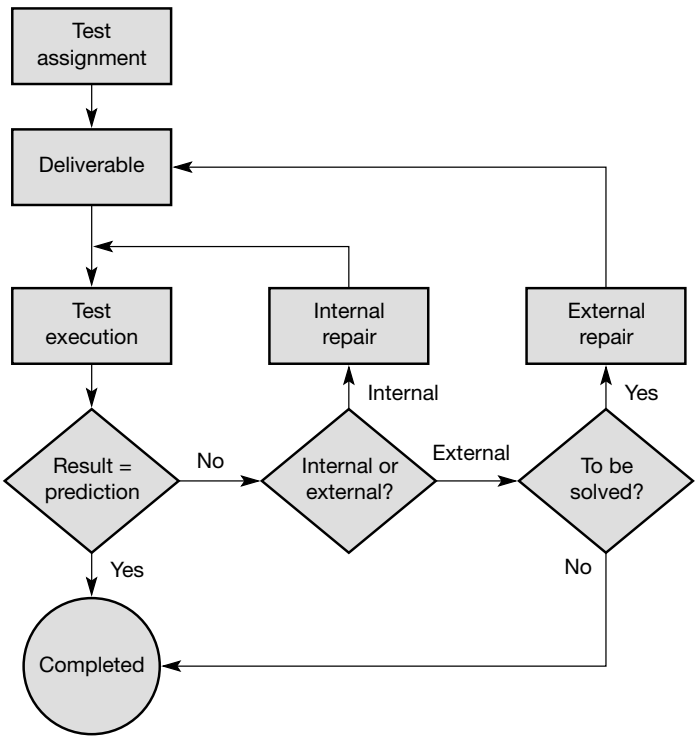


Figure 20.3
Deliverables and defects

Figure 20.3 shows the relationship between the deliverables and defects. When a deliverable is delivered for testing, a formal clearance for the test has to be given. An entry check is performed to determine whether the deliverable has sufficient quality to be subjected to the test. If the quality is insufficient, the deliverable is not accepted and is returned to its supplier. If it is accepted, the test is carried out and the result will be as expected or not. The latter means that an analysis has to be carried out to determine the cause of the difference. If the cause is a test error, the test team can solve the problem. This does not always mean that the test has to be carried out again, usually when the error turns out to be a "wrong expected result".

If the cause of the difference between the expected result and the actual result is external, a decision must be made whether to solve the defect or accept it (as a known error). The detailed procedure showing how to reach such a decision is visualized in Figure 20.4.

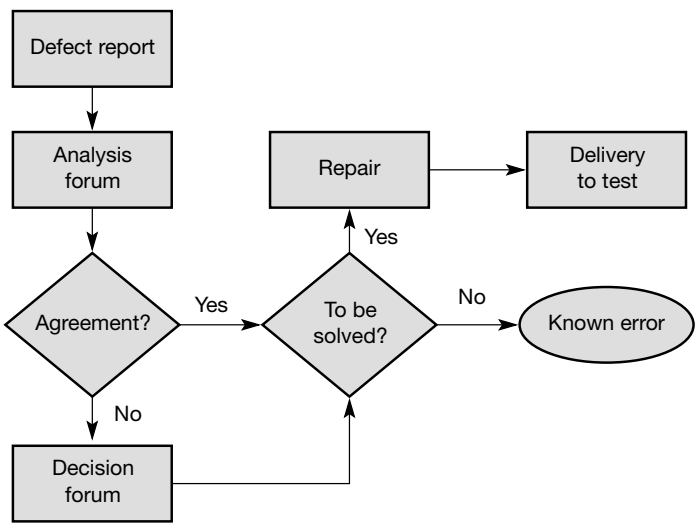


Figure 20.4
Procedure for deciding on defect reports and change requests

All defect reports concerning external defects are discussed in the analysis forum. The task of this forum is to accept or reject the defects raised and to classify them. This forum should also decide whether the defects have to be repaired or accepted as known errors. The repair of these defects can be postponed to the next release. By classifying, a decision has already been made about solving or not solving a defect. The defects in non-serious categories are usually not solved due to shortage of time and staff. Another reason for deciding not to solve a defect is the major impact that repair may have. This type of defect is mostly related to early design decisions. If the analysis forum does not agree about certain defects, a decision forum has to make a formal decision on these.

Defect report

The defect report is the central item in defect management. The efficiency and effectiveness of defect management is strongly related to the quality of this report. Every defect report should contain at least the following items:

- unique identification;
- type of defect;
- test object;
- test script;
- version number of test basis used;
- description of the defect and the preliminary steps that led to the defect or a reference to the steps in the test script;
- name of the tester who raised the issue;
- date;
- reference to the test environment used.

The defect report should provide enough information to solve defects quickly. This means that during unit and integration testing a defect report is used only for registration purposes. The defect should be corrected immediately by the person who detects it. A defect found by independent testers should be accompanied by much more detailed information as it has to help the developer to reproduce and solve the problem.

Many tools are available for defect management and writing defect reports. Commercial tools usually have the facility to define security and access levels, and offer workflow management.

PART VI

Appendices

Appendix A
Risk classification

The risk classification table is used to determine the risk of system failure. The risk is determined with the endangerment of life in mind. The risk classification table shows every possible combination of severity and probability.

If a company is involved in designing and/or producing safety-critical systems, it must have a standard for failure probabilities, damage, and risk classes. A company can use or develop its own standards, or it can use an industry standard. Sometimes it is forced by law to use industry standards. An example of a industry standard is the RTCA DO-178B (1992) used by the aviation authorities to certify airborne software. The example of risk classification in this chapter is loosely based on the RTCA DO-178B standard. The definition of risk classes, failure probability, and damage is part of establishing safety management.

Risk is defined as:

$$\text{risk} = \text{chance of failure} \times \text{damage}.$$

To make the prioritization of risks easier, risk classes are formed (see Table A.1). Most of the time it is not possible to determine the chance of failure exactly. For this reason, probability ranges are used (see Table A.2). The damage is also categorized and is a translation into the harm done to humans (see Table A.3).

Table A.1
Risk classification table – class A is the highest risk class

Risk class	Description
A	Intolerable
B	Undesirable, and only accepted when risk reduction is impracticable
C	Tolerable, with the endorsement of the project safety review committee
D	Tolerable, with the endorsement of the normal project reviews
E	Tolerable under all conditions

Table A.2 Probability ranges

Accident frequency	Occurrence during operational life
Frequent	$10\,000 \times 10^{-6}$; likely to be experienced continually
Probable	100×10^{-6}; likely to occur often
Occasional	1×10^{-6}; likely to occur several times
Remote	0.01×10^{-6}; likely to occur sometime
Improbable	0.0001×10^{-6}; unlikely, but may occur exceptionally
Incredible	0.000001×10^{-6}; extremely unlikely to occur

Table A.3 Accident severity categories

Category	Definition
Catastrophic	Multiple deaths
Critical	A single death, and/or multiple severe injuries or severe occupational illnesses
Major	A single severe injury or occupational illness, and/or multiple minor injuries or minor occupational illnesses
Minor	At most a single minor injury or minor occupational illness
Negligible	No effect

The chance of failure and damage are now combined in a new table (see Table A.4). For every combination of probability and severity category, it is determined to which risk class it belongs. This determination depends on the project, organization, usage of the system, etc. This table also shows that a detailed discussion about which category the chance of failure or the damage belongs to is not necessary because it is the combination of these two that matters. Only the risk classification of Table A.1 is used in FMEA.

Table A.4 Risk classification table – A represents the highest risk class

	Catastrophic	Critical	Major	Minor	Negligible
Frequent	A	A	B	D	E
Probable	A	A	B	E	E
Occasional	A	B	C	E	E
Remote	A	B	C	E	E
Improbable	B	C	D	E	E
Incredible	C	D	D	E	E

Appendix B
Statecharts

B.1 States

The different system conditions in state-based systems are depicted as states. A state can be defined as a clearly distinguishable disjunct condition of a system that persists for a significant period of time. The term "disjunct" means that at any one moment only one state is possible, and exactly one state applies always.

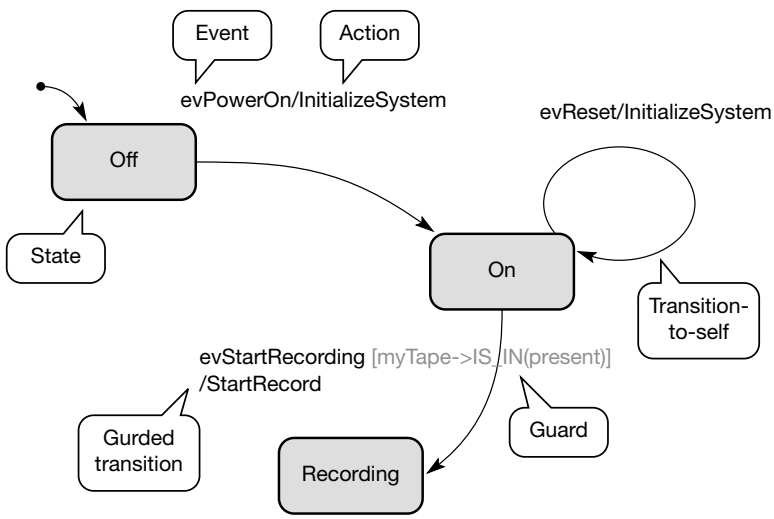


Figure B.1
Part of the statechart of a tape recorder

The *initial* or *default* state is that in which the first event is accepted (in Figure B.1 this is state *Off*). The state in which a system resides at a certain moment is defined as the *current state*.

A system has several states. In Figure B.1 for example, the states involved are *Off*, *On*, and *Recording*. A system changes state in reaction to events (such as *evPowerOn* in Figure B.1). These changes are called *transitions* (in Figure B.1 for example, the transition from *Off* to *On*). If an event results in a transition, the

old state is called the *accepting state* – the new state is called the *resultant state*. If a system is in a state that is susceptible to events, then this is also known as the *active state*. The opposite is called the *final state* – a state in which the system is not susceptible to events.

B.2 Events

An event is an occurrence inside or outside a system that can result in a transition. Four types of events exist:

1 *Signal event*. An asynchronous occurrence that takes place outside the scope of the state machine.
2 *Call event*. An explicit synchronous notification of an object by another object.
3 *Change event*. An event based on the change of an attribute.
4 *Time event*. The expiration of a timer or the arrival of an absolute time.

When testing, it is important to recognize that a system can respond to an event in three different ways:

1 A system can respond to an event by going through a transition.
2 An event is ignored when the system isn't in the right state or when the condition isn't met in order to carry out the transition. Because it is not specified how the system should react on such events, the information concerning this event is lost to the system.
3 When a synchronization pseudostate is part of the statechart, the reaction of the system is more complicated. In a synchronization pseudostate, information about certain events is stored. If the system is in State 1a and Event 3 takes place (see Figure B.2), the information about Event 3 is stored in the synchronization pseudostate (circle with the 1 in it) and a transition is made into State 3a. The behavior in orthogonal region A is not influenced by the synchronization pseudostate. The behavior in orthogonal region B is less straightforward. If the system is in State 1b and Event 5 takes place this only results in a transition to State 2b when the synchronization pseudostate is filled. A consequence of this transition is that the information about Event 3 in the synchronization pseudostate is erased. If Event 5 takes place and there is no information in the synchronization pseudostate, the system stays in State 1b and the information of Event 5 is lost to the system.

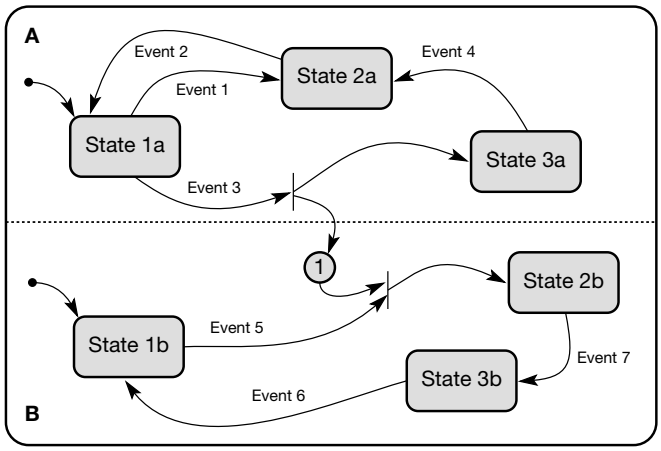


Figure B.2
Synchronization between orthogonal regions with storage of event information

B.3 Transitions

A transition is a change from one state to another. If the resultant state and the accepting state are the same, the transition is called *transition-to-self*. In Figure B.1, the event *evInitializeSystem* changes the system from state *On* to state *On*. If the transition takes place within the state, the transition is called *internal transition*.

It is possible that a condition is attached to a transition. This is called a *guarded transition*. An example of this type of transition is the transition caused by the event *evStartRecording* in Figure B.1. This event only triggers a transition whenever a tape is present, as demanded by the guard *myTape-IS_IN(present)*. If an event can lead to several transitions from one state, then guards separate these transitions from each other. These guards should not overlap because in the overlapping area the system's behavior is undefined. To enhance the clarity of such a chart, UML has the ability to use a conditional connector. In this case a transition labeled with the event leaves the state and terminates at the conditional connector, from where the guarded transitions leave for the different resulting states.

B.4 Actions and activities

An **action** (in Figure B.1, *InitializeSystem* and *StartRecord*) can be defined as non-interruptible behavior executed at particular points in state machines, whereby this behavior is considered to take a non-significant amount of time. An action can also be a string of actions that cannot be interrupted as a whole. An action can be assigned to the entry or exit of a state (using the *entry* and *exit* state labels) or to the execution of a transition (e.g. *StartRecord* on the transition from *On* to *Record* in Figure B.1).

Activities differ from actions because they take a certain amount of time and can therefore be interrupted. Because of that, activities can only be defined within a state. An action, for example, can be the starting of rewinding a tape. The rewinding itself is an activity. If the activity is completed while the state is still active, this will result in the generation of a completion event. In the case of an outgoing completion transition, the state will be left. If a tape is rewound then the rewinding activity comes to an end. The system gives a signal that the tape cannot be rewound any further. This signal makes sure that the rewinding will be stopped and that the system shifts from the state *Rewind* to the state *Standby* (see Figure 11.1).

Actions and guards can reference to the same variable. For instance, in a guard a certain variable is evaluated which must be less than a certain threshold. The action related to the transition changes the variable. Although some have the opinion that these constructions are not correct (bad programming practice), they are common in synchronous time models for modeling counters (see Figure B.3).

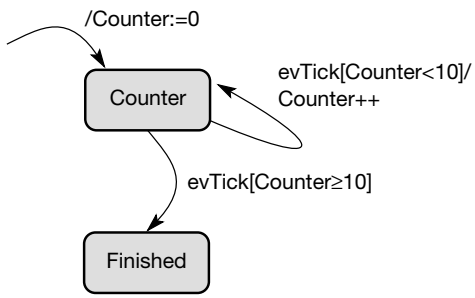


Figure B.3
Transition-to-self where a variable is referenced by the guard and an action

B.5 Execution order

Actions are carried out in a certain order. First the actions defined in the exit label of the accepting state are performed, followed by the actions appointed to transitions, and finally by the actions defined in the entry label of the resultant state.

Basically the same applies to nested states, however the situation involved here is a bit more complex. Figure B.4 shows a number of transitions in combination with nested states. The entry actions are carried out in the same order as the nesting.

If transition *t1* is executed, first action *w*, then *x* and *y*, and finally action *z* will be performed. For transition *t2*, the execution order of the actions is from the inside to the outside. So first action *p*, then *n*, and finally action *m* will be performed. If transition *t3* is executed, first actions *p* and *n* are performed, followed by action *g*, and finally by action *s*. If transition *t4* is executed, first action *q* is performed and, on "reaching" the state again, action *s* will be performed.

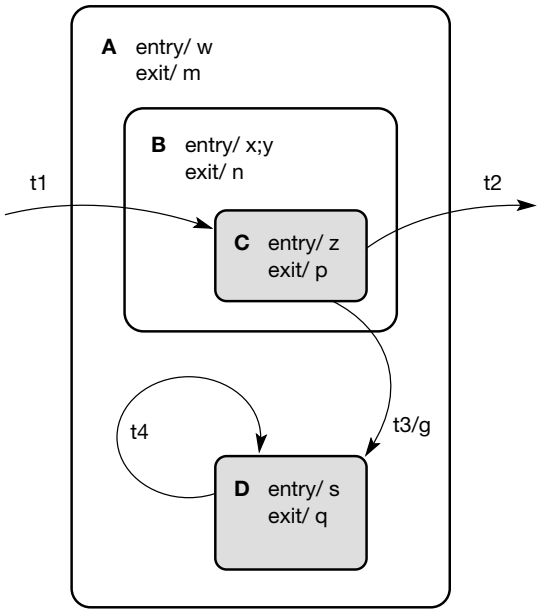


Figure B.4
Actions and nested states

If an internal transition is executed, the entry and exit actions associated with its state are not performed, only those actions directly connected to the transition (appearing after the '/' part of the transition label) are performed.

B.6 Nested states

Statecharts encourage the use of abstraction. Details can be concealed by working with superstates. These superstates are states that themselves consist of a number of states, called substates. These substates are described either in the same chart, as in Figure B.5, or the superstate is described in detail in another statechart. In this way a structure is formed in which a number of levels of abstraction can be identified.

In Figure B.5, the states *On* and *Display* are the superstates involved. The superstate *On* consists of two orthogonal regions. The left region contains two substates (*CD* and *Radio*), and the right contains the substate *Display*. This substate is in turn a superstate that breaks down into several substates.

The states *On* and *Display* consist of two partitions called orthogonal regions. When the car radio resides in superstate *On*, then it is situated in both the left and right orthogonal regions. Orthogonal regions are called *AND-states* because the system is then positioned in both regions. However, in the left orthogonal region of superstate *On*, *OR-states* are present. The car radio is in the CD- or radio-state and can't be in both at the same time. If the car radio leaves state *On*, then all orthogonal regions of superstate *On* will be left.

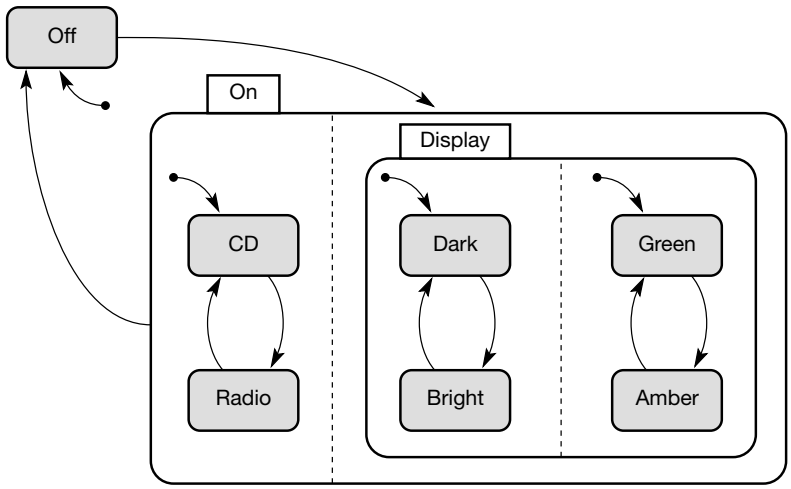


Figure B.5
Example of a nested statechart of a car radio

C Appendix Blueprint of an automated test suite

C.1 Test data

The test data are stored in a database or a file system. The data are exported in a readable format for the test kernel. This can either be a flat file or via the test kernel being connected to the database with an ODBC-link. There are two types of test data. The test scenario contains information about which test scripts have to be executed, and sometimes when they have to be executed (see Table C.1) – it is the schedule for the test kernel. The other type is the test script – this contains information on what has to be tested and how (see Table 15.1).

Table C.1 Layout of the test scenario

Test scenario 1: testing address book of electronic organizer			
Test script ID	Name test script	Schedule time	Comments
Script_1	Script_Address_1	12:01 2001-03-03	Only adding address
Script_2	Script_Address_2	12:03 2001-03-03	Only deleting address

There are several ways of checking the result of a certain action.

- *The check is part of the action*. The action starts with the input as defined in the test script. When the input parameters have been specified in the test script, the expected output is defined. The last step is to check if the result agrees with the prediction.
- *An action that can also be used as a check*. For instance, the action *View_Address* can be used to find an address or can be used to check if an address exists in the address book in a valid way.
- *The check is defined as a separate action*. The action is checking a certain state of the system under test (system under test) or a certain output.

The test scenario and script are usually exported as comma-separated files.

C.2 Start

During the start of the test suite, its environment is initialized. All parameters are initialized, the correct modules are loaded, if necessary, and a connection with the system under test is established. The next step is the initialization of the system. Part of this can be the start of the system, or else the initialization only establishes the initial system state. The start module also opens and closes the progress report and stops the test cycle smoothly.

C.3 Planner

The planner reads the test scenario. This delivers information about which test scripts should be executed may not be specific as to when they should be executed. If the test scenario contains no scheduling information, the scripts are executed in the same order as they are listed in the scenario.

The planner and the test scenario are together the driving force for the test suite. If no new information is provided by the test scenario (the end of the list is reached or all scheduled tasks are executed), the test suite produces its progress and status reports and then stops.

In essence, the planner is simply a short loop that reads the test scenario (see Figure C.1). It can be enhanced with report functions and a scheduling mechanism. When the planner is finished it gives back the control to the start module.

Figure C.1
An example of a planner in pseudocode

```
Planner(test_scenario){
open_file (test_scenario);
while NOT_END_OF_FILE
{
      read_line (test_scenario, line);
      split (line, scr, ",");   Splits a line by its separator and stores its parts in an array
      if scr[1] !=""            If the first column is empty the line is interpreted as comment
            Reader (scr[2]);    The second column of the scenario stores the name of the script to execute; the reader will be described in section C.4.
}
close_file (test_scenario);
return;
}
```

C.4 Reader

The reader retrieves its information from the test script. The planner provides the name of the test script to be opened. The reader gives control back to the planner after the last test case has been executed.

The reader also triggers error recovery when something unexpected happens during test execution (see Figure C.2).

```
Reader(test_script){
last_testcase = "";
error = NULL;
open_file (test_script);
while NOT_END_OF_FILE{
      if error == 1{   Scroll to next test case if a fatal error has occurred
               while read_line (test_script, line) !=
               NOT_END_OF_FILE {
                      split (line, case, ",");
                                          Skip next actions
                                          of the same texture
                      if case[1] != last_testcase{
                              last_testcase = case[1]);
                              error = NULL;
                              break;
                      }
               }
      }
      else {
               read_line (test_script, line);
               split (line, case, ",");
               last_testcase = case[1]);
      }
      i-f *case != ""{
               if Translator(case) == ERROR The translator is
                                          described in section E.5
                      Write_to_status_report ();
                      Error_recovery ();   Error recovery is
                                          described in section E.9
                      error = 1;
               }
      }
}
close_file (test_script);
return;
```

Figure C.2
An example of the reader in pseudocode

C.5 Translator

The translator links the action as mentioned in the test script to the corresponding function in the library of the test suite. After the execution of a function, the translator gives control back to the reader together with the return value of the executed function. See Figure C.3 for an example.

Figure C.3
An example of a translator – the default return value is error

```
Translator (action){
switch (tolower((action[2])
{
        case "action1":
               return (Action1(action));
               break;
        case "action2":
               return (Action2(action));
               break;
        default: Write_to_status_report ("Test action does
        not exist!");
               return -1;
               break;
}
}
```

C.6 Test actions

A test action is implemented by a "system specific function," which contains a standard set of elements (see Figure C.4). System specific functions can be developed to cover elementary functionality as well as aggregated functionality or complete processes. A "high-level action" that covers a complete process can be composed of "lower-level actions" that cover elementary functions. Low-level actions make the test suite flexible, while high-level make the test suite easy to use.

```
system_specific_function(){
synchr_func()        At least one synchronization function, mostly the first function
<synchr_func>        Synchronization and check functions must have a return value; this value should be returned with the return statement to the translator
<check_func>
<common_func>        Commonly used self-built functions such as data manipulation
<tool_func>
return return_value}
```

Figure C.4
These are the elements of a system specific function – the functions between < > symbols are not mandatory

C.7 Initialization

There are four types of initialization.

- Initialization of the test suite.
- Start up of the system and setting the initial state.
- Initialization of additional test equipment.
- Recovery of the initial state (after a faulty situation or execution of a test case).

Initialization of test suite

This initialization is executed at the start of the test suite. It sets all the environment variables. All hidden, and possibly interfering, processes are terminated. The variables for the test scenario and test script directory are set. The directories for the progress and defect reports are also set. The relevant modules are loaded.

Start up of the system and setting the initial state

The system under test is started – the test suite can do this automatically or this can be done manually. After this, the initial state has to be established. "Initial state" here means the state of the system where the test cases in general start (in state-based systems this is usually the same as the initial state of the system).

Initialization of additional test equipment

Additional test equipment can mean signal generators or monitors, for example. Such equipment has to be started and must also be brought in the correct state. This can be done manually, but if the equipment is fully controlled by the test suite it can also be done automatically.

Recovery of the initial state

During testing, the system does not always behave as expected. If it shows unexpected behavior it has to be initialized for the execution of the next test case. Sometimes this means the system must be completely restarted. This function is called by the error recovery. On the other hand, sometimes there are test cases that have to be executed on a system that has been reset.

C.8 Synchronization

Synchronization is the co-ordination between the test suite and the system under test such that the test suite generates only input and checks for output when the system under test is ready for it. Bad synchronization is often the cause of malfunctioning of the test suite. Possible methods of synchronization include the following.

- *GUI indicators*. If a GUI is used, most of the time there are indicators that can be used as synchronization points. For example, the changing of a window name or the appearance of a certain object.
- *Signals*. The appearance or disappearance of a certain signal on a port, or a change of a signal.
- *Time-out*. The test suite waits for a predefined length of time during which the system is guaranteed to be ready. This implies that such a time limit can be determined and is valid under all load conditions. To guarantee synchronization, the test suite must wait for the maximum possible time-out length. In general, this solution should be avoided since it usually results in a serious degradation of performances of the test suite.

C.9 Error recovery

If something unexpected happens during test execution, the test suite must be able to reset the system under test and resume with the next test case. Every new test case has to start at the initial state of the system. The error recovery function handles unexpected situations. Error recovery has several tasks:

- add the proper information to the status report and the progress report;
- return the system to its initial state when necessary by calling one of the initialization functions;
- give control back to the reader.

Several types of error are possible. For every type, error recovery has to act differently. The following types are identified.

- *An error condition which is not fatal for further execution of the test case*. The error has to be logged and then the test execution can continue.
- *The realized output does not correspond with the expected output*. The error is logged and the system is brought to its initial state.
- *Unexpected behavior*. The test suite is losing of synchronization for an unknown reason. This is logged and the system is brought to its initial state. It is possible that the only way to do this is to restart the system.
- *Fatal error – the system does not respond any more*. This behavior is logged and the system is restarted. If the system is not going to restart any more the test cycle is stopped.

C.10 Reporting

The layout of a report can be completely customized to the needs of the organization. The progress report is related to the test scenario and gives an overall view of the test scripts executed and the result of this execution.

Typical items in such a progress report are:

- the date, time, description test environment, and test suite description (when necessary);
- the test scripts executed during this test cycle;
- overall execution time;
- which test script led to detected defects;
- total number of executed test scripts and test cases;
- total number of defects detected;
- if there is a classification system of defects it is also possible to order defects by category and link them to the test scripts involved.

There are several ways of producing a status report. One is to produce an individual status report per defect detected. Another is to produce a status report per test script. The latter is, more or less, a progress report at test script level. Typical items in defect report can be:

- the date, time, description test environment, and test suite description (when necessary);
- the case which produced the defect (or an OK if the test was passed);
- the expected situation;
- the detected situation;
- the differences between the expected and detected situation;
- the severity category of the defect;
- the part of the system where the defect was detected.

The functionality required to produce these reports depends very much on the form of the reports. Some of these forms are identified below:

- *ASCII file*. The reports are produced in flat file format – this is probably the easiest form to implement.
- *RTF document*. The test suite uses an RTF parser to produce a report in the standard layout of the organization.
- *HTML document*. This can be placed on a server where the relevant people can read it. Another possibility is to place this document directly on a web server and make it accessible via an intranet.
- *E-mail*. The progress report is sent directly to the test manager. The defect report is sent directly to test manager and head of development. It also possible to use a workgroup application equipped with a document tracking and task scheduling system.
- *XML document*. This can be exported in different formats. For instance, an elaborate report for management and a basic report with no frills and the focus on information for testers.
- *Connection to defect management system*. The defects are added directly to the defect management system. Subsequently the functionality of the defect management system is used to process the defects further.
- *Entries in a database*. It is possible to produce reports in hard copy when necessary.

C.11 Checking

Checking is absolutely essential for testing. Automated checking makes life a lot easier – much more can be checked and it is not error prone. However, this can be a pitfall because there is a tendency to test too much.

Checking means comparing the output with the expected output. The output can be a signal, a certain state, an image, or data. A regular check consist of three actions.

1 Retrieving the output.
2 Comparing the output with the expected output.
3 Writing the outcome to the status report and returning the correct value.

The expected output can be given as parameter of a test action or it can be stored in a file. The comparison can be made automatically online or after the test run with a specialized tool or by hand. Sometimes the output is too big to handle in run-time. In this situation the output has to be stored so that it can be analyzed after the test run is executed.

Some signal analyzers can compare the output stream with the expected stream. The comparison of data is then situated outside the test suite. The analyzer has to provide the result of this comparison in a form which the test suite can read. This information can then be used to write the result to the reports and, if necessary, to start error recovery.

C.12 Framework

A big maintenance problem is caused by redundancy of code and once in a while there has to be a search for this. All the redundant code is replaced by a function. These functions are stored in supporting modules. By storing the supporting functions in separate modules, the framework assumes the character of a pyramid where the higher layers are built up with the functionality of the layers beneath.

The framework stores not only code but also the design of the test suite. In the design, the same hierarchy can be seen as for the modules.

C.13 Communication

Communication between the system under test and the test suite can be established by a hardware layer and/or a software layer (see Figure C.5). Sometimes, software has to be added to the system under test in order to establish stable communications. In other situations this is not appropriate, certainly not if the real behavior of the system under test should be tested. A communication approach in which the communication does not claim any resources of the system under test and in which no additional code is introduced that is not present in the end product is described as non-intrusive.

There are three possible ways of establishing communication:

- with hardware – non-intrusive;
- with software – intrusive;
- with software – bypassing the external interface.

Hardware – non-intrusive

In this situation, communication with the system under test is only possible with a special hardware layer. This layer can be built with commercially available equipment, or with dedicated self-built components, or a combination of both. There are also many tools available in which this hardware layer is already a part of the tool.

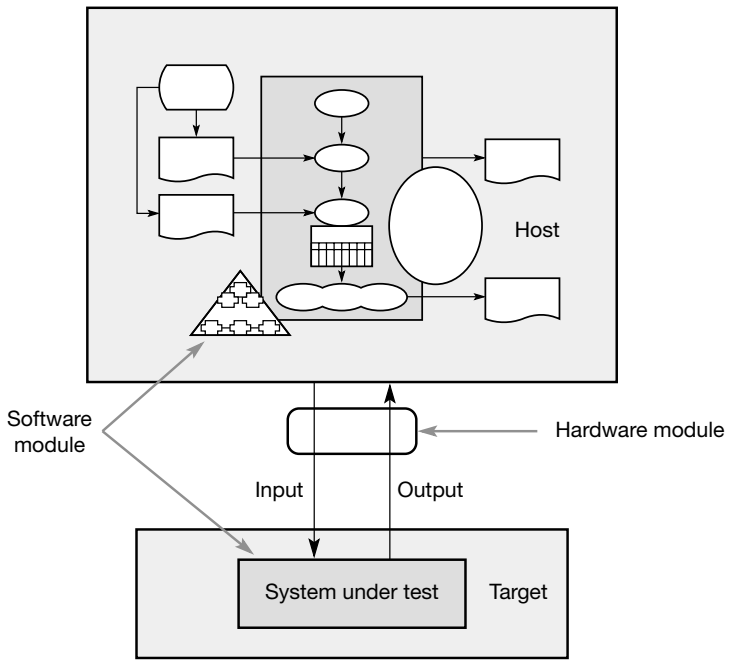


Figure C.5
The different options for communication between a test suite and a system under test

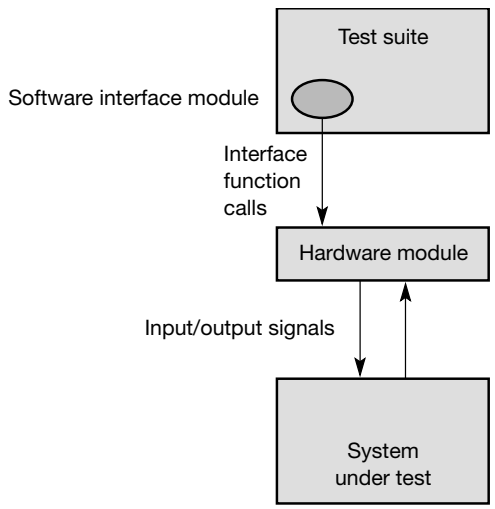


Figure C.6
Schematic description of a test suite with the use of a non-intrusive hardware layer

The hardware layer establishes communication with the system under test (see Figure C.6). It also generates the necessary input suitable for the system to process. The system under test generates output monitored by the hardware layer – or it transitions from one state to the other and generates state information monitored by the hardware layer.

The test suite controls the hardware layer (see Figure C.6). The test suite tells the hardware layer what to do based on the test script. Sometimes, if the hardware layer is able to do a comparison between the actual and the expected output, it also gives information about the expected output or where this information is stored. In this situation, the hardware layer should provide information about the success of this comparison.

The hardware layer can, for instance, be a:

- signal generator/analyzer;
- simulation environment;
- special equipment with a printed circuit board to plug in the target.

Software – intrusive

This approach is used when it is impossible or very difficult to test the system using its external interface. To control the system under test, a software module is placed inside the system (see Figure C.7). This software module makes it possible to communicate with the test suite. It uses the resources of the system under test. This means that this solution is not suitable for every test system because the resources of some are too limited to guarantee proper functioning of the software module and system. This solution is also inappropriate for performance testing. The use of the resources of the system under test by the software module can have a negative influence on the performance of the system.

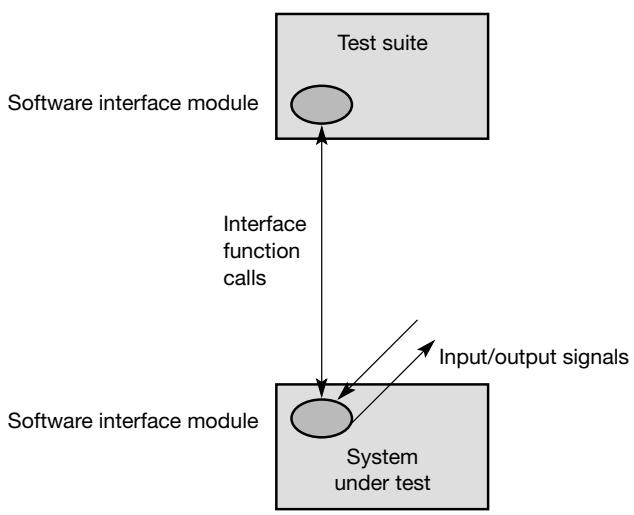


Figure C.7
Schematic description of the communication between a test suite and a system under test by using a software module implanted in the latter

The software module in the system under test should control that system. It should also provide the test suite with the necessary information. Conversely, the test suite must control this software module.

The easiest way to implement communication between a system under test and a test suite is to use an existing port of the system under test. A communication module in the test suite takes care of the communication between the two (see Figure C.7).

Software – bypassing the external interface

This approach provides a solution whenever:

- the external interface is too complicated to use for automated testing;
- there is no external interface, or one is hardly accessible;
- the external interface does not provide enough information to decide whether or not the system under test is behaving correctly.

The external interface (when it is there) is bypassed and an internal interface is used to control the system under test (see Figure C.8). An example of this approach is in gaining direct access to classes without using the interface that covers it from the outside world. A communication module inside the test suite provides the communication. There should also be a connection between the system under test and the test suite. If there is no external interface, a connector must be made to get access to the system under test. This connector preferably has no influence on the function of the system under test.

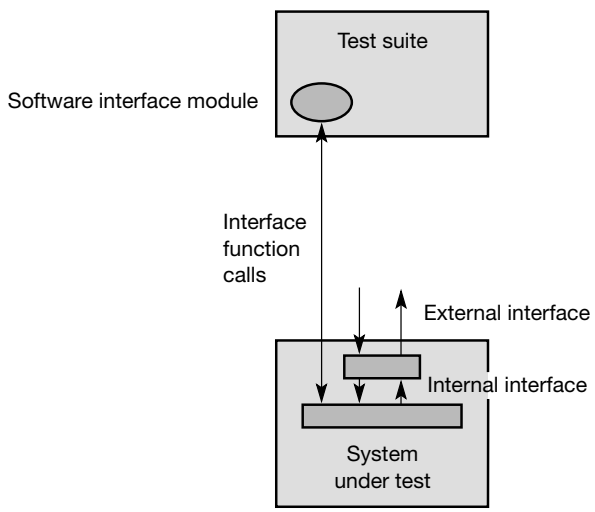


Figure C.8 The schematic description of communication with a system under test by bypassing the external interface

Appendix D
Pseudocode evolutionary algorithms

D.1 Main process

```
Initialize P(1)
t = 1
while not exit criterion
       evaluate P(t)
       selection
       recombination
       mutation
       survive
       t=t+1
end
```

D.2 Selection

```
Calculate for each individual the accumulated fitness;
for P_sz loop
       rand = random number;
       for i loop
              if rand <= f_i,accu then
                     select individual;
                     exit loop;
              end if;
        end loop;
end loop;
```

D.3 Recombination

```
For all parents loop
        rand = random number;
        if rand <= P_c
                rand = random number;
                parents interchange at random position and
                create
        offspring;
        end if;
end loop;
```

P_c is the crossover probability. For single crossover this equals to 1/(number of elements in test case).

D.4 Mutation

```
P_m = 1/(number of elements in test case)
For offspring size loop
       rand = random number;
       position = rand div P_m + 1;
       mutate individual at position;
end loop;
```

D.5 Insertion

```
Calculate the normalized accumulated fitness for offspring
population;
Calculate the reciprocal normalized accumulated fitness for
parent population;
for all parents loop
         rand = random number;
         if rand < P_s then
                 rand = random number
                 for all offspring loop
                         if rand <= f_i,accu offspring then
                                 select individual;
                                 exit loop;
                         end if;
                 end loop;
                 rand = random number
                 loop for parent population
```

```
                        if rand <= f_{i,recaccu} then
                                select individual;
                                replace with selected
                                offspring individual;
                                exit loop;
                        end if;
                end loop;
        end if;
end loop;
```

Appendix E
Example test plan

This is a complete example of a test plan that can be used as a template for other test plans. Although it is purely fictional, it contains all the necessary elements. It helps to give the reader a more concrete idea of what should be recorded in a test plan. The level of detail depends on the specific situation and must be decided by the test manager. All the names of personnel and products in this example are fictitious – any similarity with existing elements is unintentional.

The table of contents of the test plan is:

E.1 Assignment

E.1.1 Commissioner

The commissioner of this assignment is J. Davis, product manager of Solidly Embedded Ltd.

E.1.2 Contractor

The assignment is executed under the responsibility of T. Testware, test manager of Solidly Embedded Ltd.

E.1.3 Scope

The scope of the acceptance test is:

- "James" the electronic butler: version 1.0;
- "Mother" base station: version 1.0.

E.1.4 Objectives

The objectives are:

- to determine if the system meets the requirements;
- to report the differences between observed and desired behavior;
- to deliver testware such that it can be reused for future releases of "James."

E.1.5 Preconditions (external)

The preconditions are:

- the system documentation will be delivered on 14 August;
- the system will be delivered for acceptance testing by the development team on 10 September;
- the tests should be finished on 26 September.

E.1.6 Preconditions (internal)

The following preconditions are needed for the test team to complete the assignment.

- The test execution starts when the test object has passed the entry check successfully.
- Delivery delays of (parts of) the test basis or of the test object must be reported immediately to the test manager.
- All the necessary tools and the test infrastructure must be available during test execution.
- During test execution, the immediate support of the development department must be available to solve blocking problems which seriously hamper the progress of test execution.
- Changes in the test basis must be communicated immediately to the test manager.

E.2 Test basis

The test basis consists of:

- product specifications;
- standards;
- user manuals;
- project plans;
- planning.

Product specifications

- General functional design 1.1.
- Detailed functional design 1.0, which will be delivered on 14 August.
- User guide James 1.0, which will be delivered on 14 August.
- User guide Mother 1.0, which will be delivered on 14 August.

Standards

- Internal standards for test products 2.0.
- The book "Testing Embedded Software."

User manuals

- User manual test environment, which will be delivered on 14 August.
- User manual test tools, which will be delivered on 14 August.

Project plans

- Project plan James: version 1.0.
- Project plan Mother: version 1.0.

Planning

- Planning development team James: version 1.0.
- Planning development team Mother: version 1.0.

E.3 Test strategy

The test strategy of the acceptance test is based on the assumption that the module test and integration test are performed by the development team.

The test strategy is the result of meetings with the commissioner, project leader, and test manager.

E.3.1 Quality characteristics

Table E.1 shows the quality characteristics that should be tested and their relative importance.

Table E.1 The relative importance of the quality characteristics

Quality characteristic	Relative importance (%)
Functionality	50
Usability	30
Reliability	20
Total	100

E.3.2 Strategy matrix

Based on the functional design, the system is divided into five parts.

- *Part A*. Order intake/processing containing the following processes:
  - voice recognition as the order entry mechanism;
  - processing the order.
- *Part B*. Motion control: all the movements of "James" and the co-ordination between these movements – also its positioning in relation to its environment.
- *Part C*. Anticipated behavior – learning to understand the behavior of the owner (customer) and anticipate from that.
- *Part D*. Base station – includes service/monitor software and internet link to maintenance provider.
- *Total system*. "James" and "Mother" together.

Table E.2 shows the test importance for each subsystem – quality characteristic combination.

Table E.2 Strategy matrix for the acceptance test

Relative importance		Part A	Part B	Part C	Part D	Total system
	100%	40	20	10	15	15
Functionality	50	++	+	+	+	+
Usability	30	++			+	
Reliability	20	+	++			+

E.3.3 Test design techniques for system parts

Table E.3 shows the test design techniques used for the different system parts. The choice of test design techniques is based on the characteristics of the system parts, the strategy decisions, and the quality characteristics that should be tested for these system parts.

Table E.3 Test techniques assigned to subsystems

Applied test design technique	Part A	Part B	Part C	Part D	Total system
STT	+	+		+	+
ECT	+				
CTM	+				
Statistical usage		+	+	+	+
Rare event	+	+			

STT = state transition testing
ECT= elementary comparison test
CTM = classification-tree method

E.3.4 Estimated effort

Table E.4 shows the estimation of the time needed for the different phases of the test process in this project. Estimation is based on the starting and finishing dates, and also on strategy decisions.

Table E.4
Estimated effort for the test process

Activity	Effort (hours)
Test plan	32
Planning and control phase	120
Test management	48
Test configuration management	28
Methodological support	44
Preparation phase	32
Specification phase	186
Execution phase	250
Completion phase	16
Total	636

E.4 Planning

Table E.5 shows the planning of the test project. The starting and finishing dates of the activities are mentioned, including the budget reserved for the activities.

Table E.5
Planning of acceptance test "James" and "Mother"
TM = test management, MS = methodological support, TS = technical support, DE = domain expert, TCM = test configuration manager TST = tester

Activity	Start	End	TM	MS	TS	DE	TCM	TST
Test plan	07 08	11 08	32					
Planning and control	14 08	04 10						
Test management			48					
Test configuration management							28	
Methodological support				44				
Preparation phase	14 08	15 08						32
Specification phase	16 08	08 09						186
Executionphase	10 09	26 09			30	50		170
Completion phase	27 09	28 09						16

E.5 Threats, risks, and measures

The following threats can be identified.

- *It is possible that delivery to test is delayed*. If the end date is fixed, this means that the test strategy has to be used to figure out which tests must be omitted.
- *The development is disbanded when the products are delivered to test*. This means that correcting defects will be more difficult and immediate support will not be available during testing. At least one experienced developer must be available during test execution.
- *The domain expert is only available during the planned weeks of test execution*. A delayed delivery of the test object will mean that the domain expert can't be used and additional staff must be hired. Not only is the expert's support not available, but also some of the tests will take more time and be more error prone.
- *The designers are still working on the functional design and the user guides*. These must be available on time, otherwise part of the test is not specified at the start of execution. Reproducibility of the tests is then difficult to achieve. If this happens, capturing tools will be needed to record the executed test actions to make it possible to reproduce the tests.

E.6 Infrastructure

E.6.1 Test environment

The test team comprises five persons. During the test project, a PC must be available with standard hardware configuration and the standard software as used by Solidly Embedded Ltd. (for special software, see section E.6.2).

At least 20 Gb of disk space must be available on the product management directory.

E.6.2 Test tools

The following test tools are essential:

- defect management tool "DefectTracker;"
- change management tool "ChangeMaster;"
- planning software;
- additional hardware to trace the internal behavior of "James!"

E.6.3 Environment

The standard test environment is used. The testers use their normal office space during the test project.

E.7 Test organization

E.7.1 Test roles

The following test roles are required in this test project:

- test engineer;
- test manager;
- methodological support;
- technical support;
- domain expert;
- test configuration manager.

Test engineer

The responsibilities of this role include:

- reviewing the test basis (specifications);
- producing a specification of logic and physical test cases and the start situation;
- executing the test cases (dynamic testing);
- registering defects;
- archiving testware.

Test manager

The test manager is responsible for planning, and the management and execution of the test process within time and budget, and according to the quality requirements. The test manager keeps records of the progress of the test process and the quality of the test object.

Methodological support

This includes:

- establishing testing techniques;
- making test regulations;
- giving advice and assistance in the implementation of all possible testing techniques.

Technical support

This includes:

- establishing the test infrastructure;
- supervising the test infrastructure;
- physical configuration management;
- technical troubleshooting;
- ensuring reproducibility of the tests;
- giving assistance and advice.

Domain expert
This role provides assistance and advice on the functionality of the test object whenever necessary.

Test configuration manager
This role has responsibility for:

- controlling progress;
- controlling test documentation;
- keeping a record of defects and collecting statistics;
- change control of the test basis and test object;
- change control of the testware.

E.7.2 Organization structure

Figure E.1 shows the structure of the organization of the test project

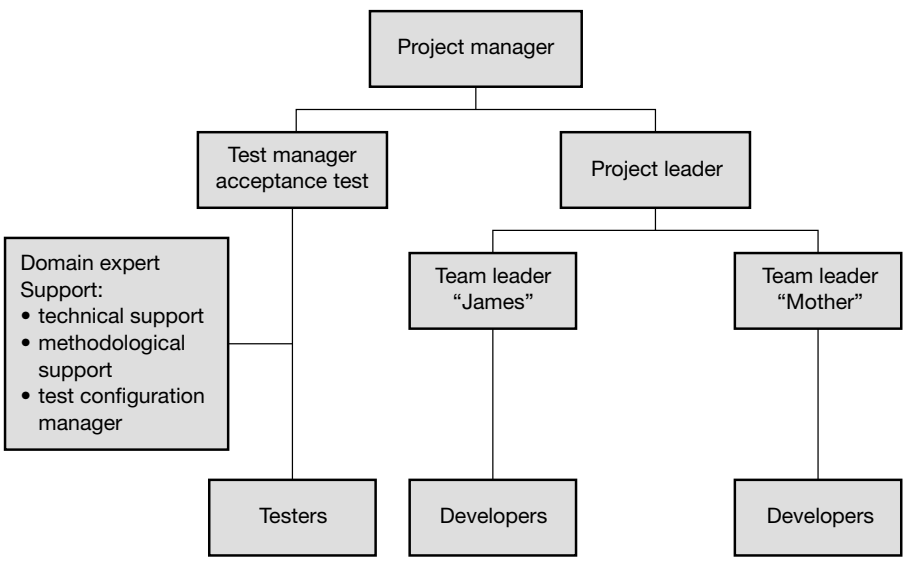


Figure E.1
Organization structure

The test manager reports directly to the product manager. During the test project, weekly progress meetings are planned. The product manager, the project leader, and the test manager attend this meeting. The team leaders of "James" and "Mother" are invited when necessary. Ad hoc meetings are held whenever necessary.

E.7.3 Test staff

Function	Name	Full-time equivalents
Test manager	T. Testware	0.20
Tester	P. Testcase	1.00
Tester	F. Testscript	1.00
Support	G. Allround	1.00
Domain expert	A. Expert	0.60 (during test execution)

Table E.6
Test staff functions and full-time equivalents

E.8 Test deliverables

E.8.1 Project documentation

The following documents will be produced during the test project.

- *Test plan*. The inital document and all its previous/future versions.
- *Defect reports*. All observed defects are reported.
- *Weekly reports*. Progress reports are made by the test manager and distributed a day in advance of the weekly progress meetings to all participants.
- *Recommendation for release*. This is formally the end of the test execution phase.
- *Review report*. This reports gives an evaluation of the test process during the test project. The objective is to improve the test process for the next release.

E.8.2 Testware

The following documents are deliverables of the test project.

- *Test script*. A description of how the testing is done. It contains test actions and checks, related to test cases, indicating the sequence of execution.
- *Test scenario*. A micro-test plan that co-ordinates the execution of several test scripts and allocates testers to the test scripts.
- *Initial data set*. Files and data sets necessary to start certain tests.

E.8.3 Storage

The directory structure shown in Table E.7 is as implemented on the central server of Solidly Embedded Ltd. The directory is stored on the directory of product management: \\PROD_MANAG

Table E.7
Directory structure

ACC_TEST_JAMES_MOTHER	
PROCDOC	Project documentation
WORK_TESTWARE	Working directory testware
FINAL_TESTWARE	Directory for archiving testware
DEFECTS	Database for storage of defects
OTHER	All other documents used or produced

E.9 Configuration management

E.9.1 Test process control

The test progress and the depletion of budget and time are monitored and reported during the test project. This is done on a weekly basis and the results are reported to the weekly progress meeting.

Defect management

For defect management, tool "DefectTracker" is used. The standard defect procedure (SolEmb 76.1 "Defect procedure") is followed.

Metrics

The test manager keeps track of the following metrics:

- number of open defects per severity category at a times;
- number of solved defects in a period per severity category;
- total number of raised defects;
- number of retests per defect;
- total number of retests.

E.9.2 Configuration management items

The test plan is subject to configuration management, starting with the first complete version. The other testware becomes subject to configuration management after the completion phase.

Changes in the test infrastructure are subject to configuration management of the technical support department of Solidly Embedded Ltd.

Glossary

The standard *Glossary of terms used in software testing* as developed by BCS SIGIST (BS7925-1) is used as a basis for this glossary.

Acceptance testing	Formal testing conducted to enable a user, customer, or authorized entity to decide whether to accept a system or component.
Actor	A module dedicated to manipulating the environment, controlled by output signals from the embedded system.
Actual result	The observed behavior of a system as a result of processing test inputs.
Behavior	The combination of input values and preconditions, and the required response for a function of a system. The full specification of a function would normally comprise one or more behaviors.
Black-box testing	Test case selection based on an analysis of the specification of the component without reference to its internal workings.
Boundary value	An input value or output value which is on the boundary between equivalence classes, or an incremental distance either side of the boundary.
Boundary value analysis	A test design technique for a component in which test cases are designed which include representatives of boundary values.
Bug	See *fault*.
Certification	A process of confirming that a system or component complies with its specified requirements and is acceptable for operational use.
Checklist	A list of questions that can be answered only by yes or no.

Code coverage	An analysis method that determines which parts of the software have been executed (covered) by test cases, and which parts have not been executed and therefore may require additional attention.
Component	A minimal software or hardware item for which a separate specification is available.
Coverage	The relationship between what is tested by the test set and what can be tested – it is expressed as a percentage.
Debugging	A process of finding and removing defects.
Defect	See *fault*.
Driver	A skeletal or purpose-specific implementation of a software module used to develop or test a component that is called by the driver.
Dynamic testing	A process of evaluating a system or component based on its behavior during execution.
Embedded software	Software in an embedded system, dedicated to controlling the specific hardware of the system.
Embedded system	A system that interacts with the real physical world using actors and sensors.
Emulator	A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.
Entry check	A check whether the test object is delivered in full and with sufficient quality to start the next phase.
Equivalence class	A portion of the component's input or output domains for which the component's behavior is assumed to be the same from the component's specification.
Error	A human action that produces an incorrect result.
Evaluation	The reviewing and inspecting of the various intermediate products and/or processes in the system development cycle.
Expected result	Behavior predicted by the specification of an object under specified conditions.
Failure	A deviation of the system from its expected delivery or service.

Fault	A manifestation of an error in software. A fault, if encountered, may cause a failure.
Functional requirement	The required functional behavior of a system.
Functional specification	A document that describes in detail the characteristics of a product with regard to its intended capability.
High-level testing	A process of testing whole, complete products.
HiL (hardware-in-the-loop)	A test level where real hardware is used and tested in a simulated environment.
Initial situation	The state which a system must be in at the start of a test case. This is usually described in terms of the values of relevant input data and internal variables.
Initial state	The system state in which the first event is accepted.
Input	A signal or variable (whether stored within a component or outside it) that is processed by a component.
Input domain	The set of all possible inputs.
Input value	An instance of an input.
Inspection	A group review quality improvement process for written material.
Integration	A process of combining components into larger assemblies.
Integration strategy	A decision about how the different components are integrated into a complete system.
Integration testing	Performed to expose faults in the interfaces and in the interaction between integrated components.
Iterative development	An iterative lifecycle based on successive enlargement and refinement of a system through multiple development cycles of analysis, design, implementation, and testing.
Known errors	Defects that have been found but not solved (yet). (Note that the word "error" is used incorrectly here – "known defects" would be more accurate.)
Lifecycle	A lifecycle structures a process by dividing it into phases and describing which activities need to be performed and in which order.
LITO	The four cornerstones of structured testing: lifecycle, infrastructure, techniques and organization.

LITO matrix	The relationship between system characteristics and specific measures (subdivided by the four cornerstones).
Logical test case	A series of situations to be tested, running the test object (e.g. a function) from start to finish.
Low-level tests	A process of testing individual components one at a time or in combination.
Master test plan	Overall test plan which co-ordinates the different test levels in a development project.
MiL (*model-in-the-loop*)	A test level where the simulation model of the system is tested dynamically in a simulated environment.
Mixed signals	A mixture of digital and continuous signals.
Model-based development	A development method where the system is first described as a model. The code is then generated automatically from the models.
Output	A signal or variable (whether stored within a component or outside it) that is produced by the component.
Quality (ISO 8402)	The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs (ISO 8402, 1994).
Quality assurance (ISO 8402)	All the planned and systematic activities implemented within the quality system, and demonstrated as needed, to provide adequate confidence that an entity will fulfill requirements for quality (ISO 8402, 1994).
Quality characteristic	A property of a system.
Physical test case	Detailed description of a test situation or logical test case containing the initial situation, actions to be taken, and the expected results explicitly defined in concrete values. The level of detail is such that when the test is executed at a later stage, it is done as efficiently as possible.
Plant	The environment that interacts with an embedded system.
Precondition	Environmental and state conditions which must be fulfilled before the component can be executed with a particular input value.

Rapid prototyping	A test level where a simulated embedded system is tested while connected to the real environment.
Real-time system	A system where correctness of the system behavior depends on the exact moment when the output is produced.
Regression testing	Retesting of a previously tested test object following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.
Result	See *actual result* or *expected result*.
Risk	Risk = chance of failure × damage.
Risk reporting	A description of the extent to which a system meets the specified quality requirements and the risks associated with bringing a particular version into production, including any available alternatives.
Sensor	A module dedicated to receiving stimuli from the environment and transforming them into input signals for the embedded system.
Simulation	A representation of selected behavioral characteristics of one physical or abstract system by another system.
Simulator	A device, computer program, or system used during software verification that behaves or operates like a given system when provided with a set of controlled inputs.
SiL (*software-in-the-loop*)	A test level where the real software is used and tested in a simulated environment or with experimental hardware.
Start situation	See *initial situation*.
State machine	A system in which output is determined by both current and past input.
State transition	A system change from one state to another.
Static testing	A process of evaluating a system or component without executing the test object.
Stub	A skeletal or purpose-specific implementation of a software module used to develop or test a component that calls or is otherwise dependent on it.
System testing	A process of testing an integrated system to verify that it meets specified requirements.
TEmb	A method for testing embedded software.

Test action	Part of a test case which defines the actions that must be performed on a test object.
Test automation	The use of software to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and test reporting functions.
Test basis	All system documentation used as the basis for designing test cases.
Test bed	Software/hardware that offers stimuli to a test object and records outputs from it.
Test case	A set of inputs, preconditions, and expected results designed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
Test infrastructure	The environment in which the test is performed, consisting of hardware, system software, test tools, procedures, etc.
Test level	A group of test activities that are organized and managed together – a division can be made into *high*- and *low-level tests*.
Test depth level	Indicates to what extent the dependencies between successive decision points are tested. At test depth level *n* all dependencies of actions before a decision point, and after *n*–1 decision points, are verified by putting all possible combinations of *n* actions in test paths.
Test object	A system (or part of it) to be tested.
Test organization	All the test roles, facilities, procedures, and activities including their relationships.
Test plan	A project plan which contains all the essential information for managing a test project.
Test process	The collection of tools, techniques, and working methods used to perform a test.
Test role	A responsibility, assigned to one or more person(s), concerning the execution of one or more task(s).
Test scenario	A micro test plan which co-ordinates the execution of several test scripts.
Test script	A description of how testing is done. It contains test actions and checks, related to test cases, and indicates the sequence of execution.

Test set	A collection of test cases.
Test design technique	A standardized method of deriving test cases from a test basis.
Test strategy	A description of the relative importance of the system parts and quality attributes leading to decisions about desired coverage, techniques to be applied, and resources to be allocated.
Test team	A group of people responsible for executing all the activities described in a test plan.
Test technique	A standard description of how to execute a certain test activity.
Test tool	An automated aid that supports one or more test activities, such as planning and control, specification, building initial files and data, test execution, and test analysis.
Test type	A group of test activities aimed at checking a system on a number of related quality characteristics.
Test unit	A set of processes, transactions and/or functions which are tested collectively.
Testability review	The detailed evaluation of the testability of a test basis.
Testing	The process of planning, preparation, execution and analysis aimed at establishing the quality level of a system.
Testware	All products produced as a result of a test project.
Unit test	The testing of individual software components.
White-box testing	Test design techniques that derive test cases from the internal properties of an object, using knowledge of the internal structure of the object.

References

Beck, K. (2000) *Extreme Programming Explained*, Addison-Wesley.

Beizer, B. (1990) *Software Testing Techniques*, International Thomson Computer Press.

Beizer, B. (1995) *Black-box Testing. Techniques for Functional Testing of Software and Systems*, John Wiley and Sons.

Bienmüller, T., Bohn, J., Brinkmann, H., Brockmeyer, U., Damm, W., Hungar, H. and Jansen, P. (1999) 'Verification of automotive control units.' In: Olderog, E.-R. and Steffen, B. (Eds.): *Correct System Design*. LNCS Vol. 1710, pp. 319–341, Springer-Verlag.

Binder, R.V. (2000) *Testing Object-oriented Systems*: *Models*, *Patterns*, *and Tools*, Addison-Wesley.

Boehm, B.W. (1981) *Software Engineering Economics*, Prentice Hall.

BS7925-1 – *Glossary of terms used in software testing*, British Computer Society – Specialist Interest Group in Software Testing.

BS7925-2 – *Standard for software component testing*, British Computer Society Specialist Interest Group in Software Testing.

Cretu, A. (1997) *Use Case Software Development and Testing Using Operational Profiles*, Concordia University.

Conrad, M. (2001) 'Beschreibung von Testszenarien für Steuergeräte-Software – Vergleichskriterien und deren Anwendung' (Description of test scenarios for ECU software – comparison criteria and their application, in German). In: Proc. of *10th Int. VDI-Congress: Electronic Systems for Vehicles*, Baden-Baden.

Conrad, M., Dörr, H., Fey, I. and Yap, A. (1999) 'Model-based generation and structured representation of test scenarios.' In: Proc. of *Workshop on Software-Embedded Systems Testing*, Gaithersburg, Maryland.

Douglass, B.P. (1999) *Doing Hard Time – developing real-time systems with UML, objects, frameworks, and patterns*, Addison-Wesley.

dSPACE (1999) *ControlDesk Test Automation Guide For ControlDesk* – version 1.2, dSPACE GmbH.

Erpenbach, E., Stappert, F. and Stroop, J. (1999) 'Compilation and timing of statechart models for embedded systems', *Cases99 Conference*.

Fagan, M.E. (1986) 'Advances in software inspections,' *IEEE Transactions on Software Engineering*, SE-12.

Ghezzi, C., Jazayeri, M. and Mandrioli, D. (1991) *Fundamentals of Software Engineering*, Prentice Hall.

Gilb, T. and Graham, D. (1993) *Software Inspection*, Addison-Wesley.

Graham, D., Herzlich, P. and Morelli, C. (1996) *Computer Aided Software Testing*: *the CAST Report*, Cambridge Market Intelligence Ltd.

Grochtmann, M. (1994) 'Test case design using classification-trees,' *Star '94*, Washington D.C.

Grochtmann, M. and Grimm, K. (1993) 'Classification-trees for partition testing.' *Software Testing, Verification, and Reliability*, 3 (2), pp. 63–82.

Hsu, W., Shinoglu, M. and Spafford, E.H. (1992) *An Experimental Approach to Statistical Mutation-based Testing*, SERC TR-63-P, www.cerias-purdue.edu/homes/spaf/wwwpub/node10.html

Information Processing Ltd. (IPL) (1996) 'Testing state machines with AdaTest and Cantate', published on the internet, www.iplbath.com/products/-library/p1001.shtml

ISO/IEC 9646 (1996) – *Part 3: Tree and Tabular Combined Notation*, 2nd edition.

ITU Recommendation Z.120 (1996) *Message Sequence Charts*, International Telecommunication Union – Telecommunication Standardization Sector (ITU-T).

Kim, S-W., Clark J.A. and McDermid, J.A. (1999) *Assessing Test Set Adequacy for Object-oriented Programs Using Class Mutation*, Department of Computer Science, The University of York.

Kruchten, P. (2000) *The Rational Unified Process: an introduction*, Addison-Wesley.

Lutz, R.R. and Woodhouse, R.M. (1999) 'Bi-directional analysis for certification of safety-critical software,' *1st International Software Assurance Certification Conference proceedings*, Feb. 28 – Mar. 2, Washington D.C.

Mitchel, D. (1996) 'Test bench generation from timing diagrams'. In: Pellerin, D. and Taylor, D (Eds.): *VHDL Made Easy*, Appendix D, Prentice Hall.

MOD (1996a) *Safety Management Requirements for Defence Systems*. *Part 1: Requirements*, Def Stan 00-56, Ministry of Defence.

MOD (1996b) *Safety Management Requirements for Defence Systems*. *Part 2: Guidelines*. Def Stan 00-56, Ministry of Defence.

Musa, J.D. (1998) *Software Reliability Engineering: more reliable software, faster development and testing*, McGraw-Hill.

Myers, G.J. (1979) *The Art of Software Testing*, John Wiley and Sons.

OMG (1997) *UML Notation Guide Version 1.1*, OMG document ad/97-08-05, Object Management Group.

OMG (1999) *OMG Unified Modeling Language Specification Version 1.3*, published on the internet www.omg.org/uml

Pohlheim, H. (2001) *Genetic and Evolutionary Algorithm Toolbox*. Chapters 1–7, http://www.geatbx.com/docu/algindex.html

Pol, M., Teunissen, R. and van Veenendaal, E. (2002) *Software Testing: a guide to the TMap® Approach*, Addison-Wesley.

Reid, S. (2001) 'Standards for software testing', *The Tester* (BCS SIGIST Journal) June, 2–3.

Reynolds, M.T. (1996) *Test and Evaluation of Complex Systems*, John Wiley and Sons.
RTCA/DO-178B (1992) Software Considerations in Airborn Systems and Equipment Certification, RTCA.
Sax, E. (2000) *Beitrag zur entwurfsbegleitenden Validierung und Verifikation elektronischer Mixed-Signal-Systeme*. PhD Thesis, FZI-Publikation, Forschungszentrum Informatik an der Universität Karlsruhe.
Schaefer, H. (1996) 'Surviving under time and budget pressure,' *EuroSTAR 1996 Conference Proceedings*.
Simmes, D. (1997) *Entwicklungsbegleitender Systemtest für elektronische Fahrzeugsteuergeräte* (in German), Herbert Utz Verlag Wissenschaft.
Spillner, A. (2000) 'From V-model to W-model – establishing the whole test process', *Conquest 2000 Conference Proceedings*, ASQF.
Sthamer, H.H. (1995) *The Automatic Generation of Software Test Data using Genetic Algorithms*, Thesis at the University of Glamorgan.
Untch, R.H. (1995) *Schema-based Mutation Analysis: a new test data adequacy assessment method*, Technical Report 95-115, Department of Computer Science, Clemson University.
Wegener, J. and Grochtmann, M.(1998) 'Verifying timing constraints of real-time systems by means of evolutionary testing,' *Real-Time Systems*, Vol. 15, No. 3, pp. 275–298.
Wegener, J. and Mueller, F. (2001) 'A Comparison of static analysis and evolutionary testing for the verification of timing constraints,' *Real-Time Systems*, Vol. 21, No. 3, pp. 241–268.
Woit, D.M. (1994) *Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules*, Queen's University, Kingston, Ontario.

Further reading

Burton, S. (1999) 'Towards automated unit testing of statechart implementations.' University of York.
Dornseiff, M., Stahl, M., Sieger, M. and Sax, E. (2001) 'Durchgängige Testmethoden für komplexe Steuerungssysteme – Optimierung der Prüftiefe durch effiziente Testprozesse' (Systematically consistent test methods for complex control systems: enhancement of testing depth through test automation Control Systems for the powertrain of motor vehicles) in German. In: Proc. of *10th Int. VDI-Congress: Electronic Systems for Vehicles*, Baden-Baden.
Grochtmann, M., Wegener, J. and Grimm, K. (1995) 'Test case design using classification-trees and the classification-tree editor CTE'. In: Proc. of *8th International Software Quality Week*, San Francisco.
Isakswn, U., Jonathan, P.B. and Nissanke, N. (1996) 'System and software safety in critical systems,' The University of Reading.
Lehmann, E. and Wegener, J. (2000) 'Test case design by means of the CTE XL.' In: Proc. of *EuroStar 2000*, Copenhagen.
Liggesmeyer, P. (1990) *Modultest und Modulverifikation*: *state of the art*. BI-Wiss.-Verlag Mannheim.

Low, G. and Leenanuraksa, V. (1988) 'Software quality and CASE tools.' In: IEEE Proc.: *Software Technology and Engineering Practice*.

Ostrand, T. and Balcer, M. (1988) 'The category-partition method for specifying and generating functional tests,' *Communications of the ACM*, 31(6), pp. 676–686.

Sax, E. and Müller-Glaser, K.-D. (2002) 'Seamless testing of embedded control systems.' *3rd IEEE Latin-American Test Workshop*, Montevideo.

Shankar, N. (1993) 'Verification of real-time systems using PVS,' *Lecture Notes in Computer Science*, Vol. 697, pp. 280–291.

Spafford, E.H. (1990) 'Extending mutation testing to find environmental bugs', *Software Practice and Experience*, Vol. 20, No. 2, pp. 181–189.

Tracey, N., Clark, J., Mander, K. and McDermid, J. (1998) 'An automated framework for structural test data generation. In Proc. *13th IEEE Conference in Automated Software Engineering*, Hawaii.

Tsai, B-Y., Stobart, S. and Parrington, N. (1997) 'A method for automatic class testing object-oriented programs using a state-based testing method'. In: Proc. of *EuroStar 1997*, Edinburgh.

Wegener, J., Baresel, A. and Sthamer, H. (2001) 'Evolutionary test environment for automatic structural testing'. *Information and Software Technology*, Vol. 43 pp. 841–854.

Wegener, J. and Pitschinetz, R. (1995) 'Tessy – an overall unit testing tool.' Proc. of *8th International Software Quality Week*, San Francisco.

Company Information

Software Control

Software Control is a client-oriented organisation which takes a structured approach to research and development. A team of over 400 staff help to ensure that the company remains a trend setter and market leader, offering services such as the implementation of structured testing, test management, quality control, process improvement, auditing, and information security. These are closely matched to the needs of the client. Additional services include:

- effective automated testing;
- structured performance testing;
- setting up and exploiting testing in an organisation;
- testing embedded software.

Software Control's generic model for process improvement takes into consideration both the 'hard' and 'soft' aspects. The 'Quality Tailor Made' approach is used for setting up quality control in projects. If you want to learn more, a number of books are available on the internationally recognised market standards developed by the company. These are TMap®, the approach for a structured testing, and TPI®, the approach for test process improvement. Established in 1986, Software Control is a part of **Sogeti Nederland B.V.** and is ISO-9002 certified. **Sogeti Deutschland GmbH**, founded in 1999, is the German representative of Software Control. It offers testing and quality assurance services.

Gitek nv

Gitek nv, founded in 1986, designs and develops software, as well as specialising in software testing and customised IT solutions. Gitek employs over 130 people, including 50 professional software Test Engineers. Its customers include the pharmaceutical and telecoms industries, as well as insurance companies. The company takes a personal approach, ensuring that services are adapted to the customer's specific requirements. Gitek is the exclusive distributor of TMap®, TPI® and TAKT © in Belgium. Gitek's services offer a complete solution for testing. They include:

- Participation in the operational test process;
- Test advice and support, training and coaching;
- Defining and implementing a structured test process;
- Selection and implementation of test tools;
- Improvement of the test process.

Contact details

Sogeti Nederland B.V.
Software Control
Postbus 263
1110 AG DIEMEN
The Netherlands
http://www.gitek.be
gitek@gitek.be

Sogeti Deutschland GmbH
Schiessstrasse 72
40549 DUSSELDORF
Deutschland
www.sogeti.nl
info@sogeti.nl

Gitek nv
St. Pietersvliet
3, B-2000 Antwerp
Belgium
www.sogeti.de
kontakt@sogeti.de

Index