VTT PUBLICATIONS 406

The reuse of tests for configured software products

Jukka Korhonen, Mika Salmela & Jarmo Kalaoja
VTT Electronics

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 2000

ISBN 951–38–5556–2 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–5557–0 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 URL: http://www.inf.vtt.fi/pdf/)

Copyright © Valtion teknillinen tutkimuskeskus (VTT) 2000

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Maini Manninen

Libella Painopalvelu Oy, Espoo 2000

Korhonen, Jukka, Salmela, Mika & Kalaoja, Jarmo. The reuse of tests for configured software products. Espoo 2000. Technical Research Centre of Finland, VTT Publications 406. 67 p.

Keywords software testing, feature-based software, regression testing, configured systems

Abstract

New efficient software production techniques are important for improving the time-to-market of software products. One example of such advanced techniques is the so-called feature-based software production which employs high-level requirements or features in finding and selecting reusable software components for the development of new products. This kind of model-driven software development shortens the production time, but the validation of configured products still remains a bottleneck.

An effort to apply regression testing techniques to configured software products shows that these techniques are not very well suited to meeting the new testing challenges. It is obvious that retesting an entire program, containing possibly only a few minor changes, is expensive. Therefore, an efficient testing approach is required for optimizing the size of the test suite. Other important issues concerning the testing approach are the design of reusable tests, the configuration and management of tests, and the automation of test execution.

In the research, the testing efficiency problem is solved by using the idea of reusable software components from the feature-based production. In software testing the idea converts into a set of reusable test components designed for a product family. From a test material repository a suitable subset of tests is selected, modified, and configured to cover the characteristics of the product being tested. In addition, the repository may contain other relevant test data to be used for configuration, such as the test plans and the test environment configuration. The technique is called the feature-based testing approach.

For identifying the relevant test data, the method proposes links to be created between the product features and the test material. The result of the test configuration depends on the automation degree, varying from a simple test

involving identification of lists useful in manual testing to executable tests in a fully automated test environment.

Often in structured testing the test case assumes that the software is in a specific state. Therefore joining test scripts arbitrarily may not produce the desired results. For that reason, we propose utilization of test specification components that are capable of using product feature data, taking care of the execution order and selecting appropriate tests for the product.

When implementing the feature-based testing approach, the issues to be emphasized are script development, test development and execution, and test management. A support system implementing the main characteristics of the feature-based testing approach has been outlined in the report. The tool is demonstrated in a case study.

Preface

This work was carried out within the KATA-project by VTT Electronics, one of the departments of the Technical Research Centre of Finland. The project was funded by the National Technology Agency (Tekes), Nokia Display Products, Semel, Suunto, Polar Electro, and Vaisala.

The objective of the KATA-project was to develop software testing techniques for configured software products. Configured products have been seen as an important constraint for the project as the trend in software production is moving swiftly towards more advanced production techniques, one example of this being the feature-based production concept. The project was initiated as a state-of-the-art study of the currently applied testing techniques and tools, and as an evaluation of their suitability for configured products. An outline for a new feature-based testing approach is proposed on the basis of the study and the survey on user needs.

The report presents the results of the survey and describes the feature-based testing approach in detail.

Oulu, Finland, October 1999.

Jukka Korhonen, Mika Salmela, Jarmo Kalaoja

Contents

Abstract 3

Preface 5

List of Symbols 8

1 Introduction 9

2 Feature-based software development 11

3 The problem and some definitions 13

4 The state of the art in feature-based testing 16

4.1 The Ovum report 16

4.2 Conference papers 20

4.3 Some examples of the tools 24

5 Testing based on features and test components 27

5.1 The footing 27

5.1.1 Regression testing 27

5.1.2 Test management 29

5.1.3 Generic reusability issues in test design 31

5.2 The feature-based testing approach 33

5.2.1 The feature-based testing concept 33

5.2.2 The test material 34

5.2.3 Organizing and linking the test material with the feature model ..35

5.2.4 The processing of the test material 37

5.2.5 The taximeter example 42

5.2.6 Generating tests 43

5.3 The optimization of test suites 44

6 Support for the feature-based testing 49

6.1 Script development 49

6.2 Test development and execution 49

6.3 Test storage and management 50

6.4 A support system for test case reuse 50

6.5 The test process..52

7 A case study 54

7.1 The system..54

7.2 The feature model of the system..55

7.3 Writing reusable test components..59

7.3.1 The variation points in testing 61

8 Summary 63

References..65

List of symbols

ERD	Entity-relationship diagram
F	Feature
FSM	Finite state machine
GUI	Graphical User Interface
HTML	Hypertext markup language
OSI	Open system interconnection
PR	Product requirement
QA	Quality Assurance
R	Test result file
S	Script
SUT	Software under test
TS	Test suite
TSRL	Test requirements specification language
TST	Test
TTCN	Tree & tabular combined notation
URL	Uniform resource locator

1 Introduction

The goal of testing is to discover problems or to make a judgment on the quality or the acceptability of a product. Testing is needed, because we - analysts, designers, coders - are fallible, and thus software systems contain faults.

Testing is not only a must, it is a laborious must. Even though automation can, to some extent, be applied to testing, there are still application domains and implementation solutions that are not amenable to the automation aspirations. This is the reason for the demand for new testing approaches and methods.

Feature-based software production, which takes advantage of reusable software components, is a fairly new method. The approach employs a feature model, containing all the possible components for defining a new product in the application domain. The desired characteristics for the product are gained by selecting the relevant components from the model. The product is then configured by means of the supporting tools.

Even though this kind of model-driven software development has shortened the production time, the validation of configured products still remains a bottleneck. Regression testing is a usual, straightforward means of validation.

In general, regression testing is initiated through the emergence of a new requirement, or when the program and its documentation have been modified and need to be tested. The goal of regression testing is to convince the maintainer that the program still performs correctly with respect to its requirements. However, the term *regression testing* is just a common name for a re-testing process for modified software. It does not give us any instructions on creating reusable test cases, or on how to configure test suites from existing test material, nor does the term tell us whether or not it is reasonable at all to strive towards reusable tests. To begin regression testing, the test organization and personnel involved need to decide on the outline of the test procedure, or to create a new one [von Mayrhauser et al. 1994]. In this matter, little help can be drawn from literature.

An effort to apply regression testing techniques to configured software products reveals that the techniques are not well suited to meeting the new testing

challenges. With the feature-based development paradigm, the production of new software is rapid, and slowing down the process with any inefficient validation procedures is not desirable. What is desired, is a method for bringing the production of the test suites to the level of the production of software.

Although there is clearly great potential for test reuse in configured products, no specific testing methods have been proposed for the purpose yet.

Extracting the essence of feature-based software production and applying this knowledge to testing could serve as a basis for an efficient testing approach. Thus we would have a set of reusable *test* components, capable of being configured to cover the characteristics of the product to be tested.

This method brings out many questions. Should we decide to apply the method, what kind of requirements would then be set on testing process and test case design? There would also be the question of test automation and tool support in the new situation. Could the benefits of feature-based software production also be gained by the feature-based testing approach? These, and other related testing issues are discussed and evaluated in the report.

2 Feature-based software development

The purpose of feature-based software development is to answer the high requirements of today's system production. System production has to be capable of addressing the needs of the customer rapidly and profitably. The list below specifies the characteristics required for keeping up with the development.

- time-based competition
- quick response
- fragmented markets
- proliferating variety
- increased customization
- continual improvement
- shortening product life cycles
- cycle time reduction

Feature-controlled product configuration is targeted at addressing most of these requirements.

The figure shows the feature-based software production process with two main sections: software delivery and software development. Marketing and delivery management use domain models developed and maintained by software development. The essential model in the reuse and production of software is the feature model, describing all the features and relations available in the domain.

Marketing specifies the product desired by the customer by means of a feature model. The feature model presents the existing features to be re-used. At best, the new product can be entirely configured by combining individual features included in previous deliveries. This being the case, the software product can be assembled simply by integrating existing software components on top of a standard software platform (Figure 1).

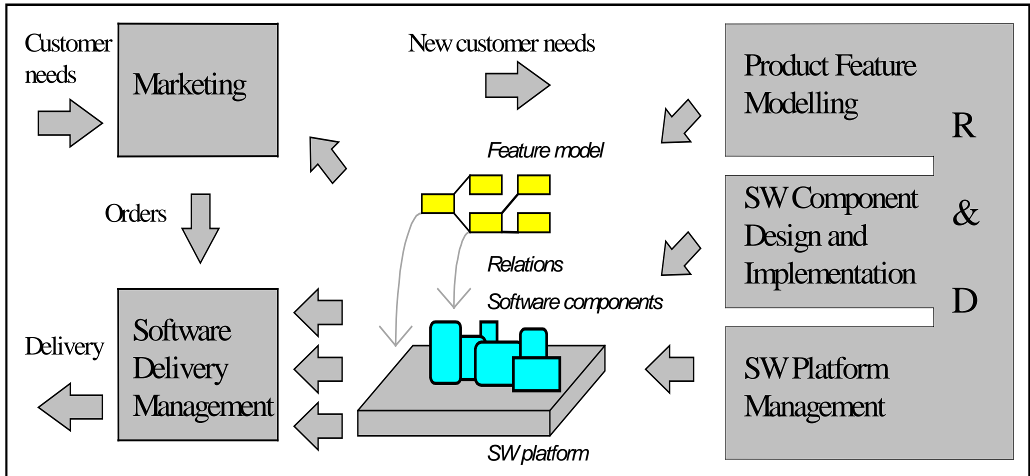

Figure 1. Feature-controlled software development and delivery process.

The feature-based production process seems to offer an excellent platform for applying regression testing principles. The feature model defines the features for the domain, thus providing a solid basis for system level testing.

The implementation of the feature-based production sets, inevitably, some requirements on the products and on the domain. The number of the different versions has to be fairly large for the feature-based approach to be profitable. On the other hand, the differences between different versions should not be too distinctive, or else the re-use rate tends to fall too low.

3 The problem and some definitions

Our intention is to find or to develop more efficient techniques for testing component-based software products. We believe that the key issue in this effort is the reuse of existing tests, including the selection of the necessary tests and the configuration of the tests for the executable test suites.

The key issue can be expressed as follows:

> How to validate a change in configured software by means of existing tests in an efficient way?

This is the most important question. If we have a novel application in a domain, it is in most cases useful to test the product as thoroughly as possible within the limits of the test resources, in respect of the requirements of the product. However, if several applications have already been generated, we will certainly wish to reduce the testing effort by reusing the tests. An obvious decision in this situation will be to concentrate on the features that have been changed compared with the earlier applications. Therefore, considering the key issue, the main purpose of the new technique is to maximize the use of the existing test material, and at the same time to reduce the size of the test suite.

The report proposes a testing approach called *feature-based testing,* which, essentially, follows the lines of feature-based production. For the term, we will use the description given in Definition 1.

Definition 1. *Feature-based testing* is a testing method for configured software products. In the method, test components and product feature descriptions are used for selecting, modifying and configuring the tests.

An outline of the feature-based testing approach is shown in Figure 2. The test design originates from the product family characteristics described in the feature model. The test design and implementation have to be carried out in consideration of the generic design constraints, which stem from the test environment, for instance. In addition, the requirements of reusability need to be paid attention to.

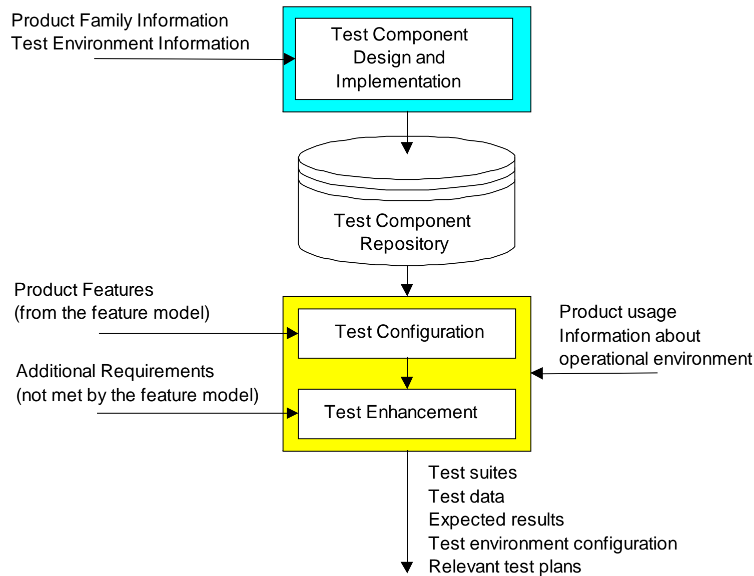

Figure 2. The feature-based testing method.

The *test component repository* is used for storing the accepted test components. A test component can, as a matter of fact, be any document relevant to testing. For instance, test plans, test data files, test result files, and the test tool configuration information are essential for efficient testing. They should be included in the test repository, in addition to the test cases and scripts.

As new products are built, the features of the product are used as a basis for selecting and configuring the test components. In most cases, a new product can not be implemented as such by using only the existing software components, which is why new software needs to be developed for the purpose. Accordingly, as it is likely that the test components of the repository do not completely cover all the product features, the test suites need to be enhanced with complementary tests. These components are potential reuse material, and if applicable, they should later be taken through the test component verification and validation processes for inserting them in the test component repository.

One of the crucial things in the outline is the question of using the product feature information for selecting the tests. Obviously, it is necessary to create some sort of connection between the features and the tests. Other issues to be

solved are, for instance, how to design tests to provide optimal reusability and how to develop executable test suites.

But what is *a feature*? In general, a feature is considered to be something that the user[1] is able to notice in the product. In other words, we take the user's point of view when analyzing a product. Thus a feature may be a system function, a subsystem name, or possibly a piece of some other qualitative information that characterizes the product. A feature might be further divided into subfeatures. We can also relate a feature to user requirements. This gives us the following definition:

Definition 2. A feature is an implementation of one or several user requirements.

This is a useful notion in case feature-based testing can not be applied. Instead, we might find that some of the conditions for feature-based testing approach are not met, and therefore we decide not to use the approach. When the relations between the features and the requirements have been defined, we are able to use the product requirements instead of the features. This method is frequently used for many commercial tools.

[1] Note that the term 'user' may include also other user groups than the end user, e.g. production, or other organisations.

4 The state of the art in feature-based testing

Test automation, supported by various test tools and environments, usually aims at improving the testing efficiency, measuring the test coverage, or formalizing the test process. Currently, the test tools are capable of supporting such test activities as:

- planning and management
- requirements analysis and validation
- test case generation
- coverage analysis
- test development
- test execution
- simulation and load testing.

Example 1. Test tool characteristics.

Considering the focus of our researchREF, we can select the items that interest us from the list in Example 1: planning and management, as well as test development. *Configuration* means that a test set (or a test suite) can be configured by using the knowledge related to the product, the requirements or the application domain. *Management* implies that the test material can be stored and retrieved in a controlled manner. The issues included in the previous topics were studied by using the Ovum report on testing tools [Ovum 1998], as well as papers and publications on the subject, published during the recent years.

4.1 The Ovum report

The Ovum report [Ovum 1998] evaluates the most popular testing tools by means of an assessment framework. The framework applies the list of characteristics shown in Example 1 in the previous chapter. Each one of the main items on the list is further structured into several subitems. Though the tools in the report are intended for various purposes, e.g. test management, source code analysis, and Windows / client-server / GUI testing, and not all of

them are relevant in this case, the framework itself provides a useful classification scheme for tool characterization.

In the Ovum evaluation structure, the most interesting item is test development, which has three relevant subitems:

- script language
- test-development support
- storage and management

The script language is in our interest (for instance, the support for variation of the tests may require parameterization of the test cases, and script language is needed for enabling this), in addition to the test development support, storage and management. According to the Ovum report it is advantageous if the script language

- supports logical condition constructs, and
- supports the calling of sub-scripts with parameters.

There are 28 other, less relevant subitems for script language, which were all considered to be of equal importance in the Ovum report.

The second item on the test development list is test-development support, listing 24 items concerning test design, debugging and documentation support. From our point of view, the list reveals some interesting issues:

- support for the linking of the script components to the test cases
- support for libraries of commonly required functions / tests
- support for a sequenced running of the scripts from a test-control script
- support for test sequencing through the knowledge of the application structure;
- sequencing by selecting an execution path from the application structure
- support for an automatic generation of a sequence for navigating to all application states

The last item, storage and management, also needs to be discussed, because the control and management of the test material, e.g. scripts, test data, test cases,

expected results, reports and test environment, is of great importance. In addition, the version control of test material and organizing the test material for a project- or product-related structure are equally important.

Through using the previous lists of items as tool selection criteria and studying fifteen tool evaluations[2] in the Ovum report, we notice that:

- the basic requirements set on the script language by us are met by all of the selected test tools (Table 1);
- the first three items of test development are well covered by the tools; however, the last three are met by very few tools (Table 2);
- the storage and the management of the test material is usually taken care of by the tools, except in version controlling (Table 3).

Table 1. Script language issues.

Script language (total number of tools is 15)	**Logical conditional constructs**	**Calling of sub-scripts with parameters**
Number of supporting tools	**15**	**15**

[2] Fifteen tools were selected out of the 42 tools in the report for closer evaluation. Only the tools that are designed mainly for dealing with embedded systems, Windows or GUI testing were included. The rest of the tools concentrate more on web testing, source code analysis etc., and were therefore less appropriate for our purposes.

Table 2. Test development issues.

Test Development (total number of tools is 15)	**Linking of scripts to test cases**	**Function / test libraries**	**Test-control scripts**	**Test sequencing based on application structure**	**Test sequencing based on execution paths**	**Sequence for navigating to all application states**
Number of supporting tools	**10**	**14**	**14**	**2**	**3**	**2**

Table 3. The storage and the management of the test material.

Storage and Management (total number of tools is 15)	**Product-related structure for test material**	**Clear representation of the test components of the application test platform**	**Automatic version control for test material**
Number of supporting tools	**12**	**10**	**3**

The issue of supporting the test libraries for commonly required functions is particularly interesting. Considering the problem definition of the study, this item is obviously of great relevance. Linking the test libraries to product functions means that by identifying a function the necessary tests are also identified and this information is available for the next products. Subsequently, the tester will be able to use the earlier function-test suite associations to configure tests for a new product.

As Table 2 shows, all the tools except one have some sort of support for the test libraries of common functions / requirements. A prompt analysis gives us the following conclusions:

- The issue is considered important by the tool developers. Majority of the relevant tools have supporting functions.

- However, the item is one among 23 other items that Ovum has found equally important in test development.
- Using libraries of test material for common functions or features (this is the solution applied by most of the tools) is a simple solution, which, in spite of its seeming plainness, is able to provide many of the potential advantages of requirements-based testing.

4.2 Conference papers

A survey of the latest conference publications on feature-based testing (or requirements-based testing) reveals little new when compared with the Ovum report. Though one would have expected a larger scope in the research papers, nearly all papers concentrate on presenting a new tool for feature / requirements-based testing. If a method is proposed, it usually concerns the size reduction or the optimization of the test suite, thus failing to address the most essential issue in this case.

The Ovum report shows that the configuration of the test suites by using the product or software requirements is supported by the majority of the testing tools. This tendency can also be noted in the conference papers. Typical examples are the articles by [Little 1996], [Liu et al. 1993], and [Mayrhauser et al. 1993], describing test techniques and test systems, all based on linking the requirements and/or functions with the test material. The general approach to the problem is quite similar in these cases, but differences can be noticed in the implementation. The most popular solution seems to be a relational database, which has been enhanced with some special, test supporting characteristics.

Liu [Liu et al. 1993] analyze regression testing, its problems, and come to the conclusion that it is important to provide information for the testers about the scope of the changes and to maintain a history of the test data. To meet these requirements, a regression testing environment should provide:

- mechanisms for storing, retrieving, and updating test material
- a system of configuration management for the material

- traceability of relevant test material by linking the material with the test objects
- support for analyzing the effects of changes in the test objects and their links.

The authors also propose that there are two kinds of useful links, *inner links* and *outer links*, which should be used for the test material. Inner links are needed for viewing the relationships within single object types (e.g. a module, or a test case). These links are useful, for instance, when several modules are needed for building up a program. The outer relationships are related to 'life-cycle objects', denoting the links between specifications, test cases and programs. Both links are needed when the impacts of changes are being traced; the effects of changes can be seen in either internally related components, in externally related components or possibly in both of them.

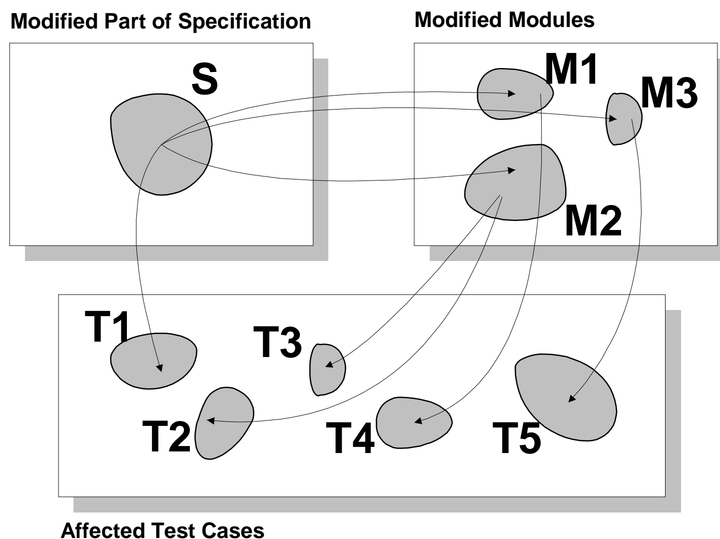

Figure 3. The SEMST concept.

To demonstrate the previous objectives, the authors implemented a prototype tool called SEMST (Figure 3). The SEMST system provides the necessary functions through four major components: the system monitor, the specification segment, the program segment, and the test case segment. The components are manipulated through a database.

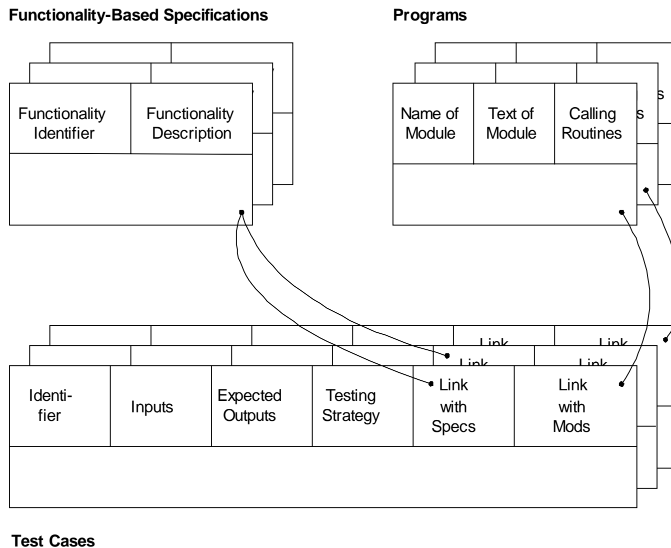

Figure 4. The logical structure of objects in SEMST.

Interestingly enough, the SEMST concept and the implementation reveal two divergent visions for implementing the linking. The concept uses modified specifications and programs to point out the test cases to be executed to validate the changes. Links between the specifications and the modules are also used. The implementation has taken a somewhat different viewpoint (Figure 4), with the links starting at the test cases and ending at the specifications and the modules. In conclusion, the SEMST concept proposes that both specifications and modules are important sources of information in selecting the relevant test cases after modifications have been made.

Von Mayrhauser [et al. 1993] suggest that test cases should be attached to requirements instead of relating them to specifications (Figure 5). In case of complex systems, the requirements have to be structured 'down to the actual function level', to the so-called *atomic requirements*. They claim that this kind of hierarchical structuring has several advantages:

- Functionally related requirements are clustered together.

- The test case suites can be associated with a chosen level of abstraction. This provides the basis for editing the requirements and the related test cases at the highest applicable level of abstraction.
- Appropriate rules can be provided for the inheritance properties of requirements and test cases.
- The representation of requirements relates to the abstraction level. According to von Mayrhauser, the use of a proper tool changes the representation of the requirements structure so that they can be readily used for testing purposes.

The different types of requirements, qualitative and quantitative, are also discussed. The solution offered involves fitting the qualitative requirements with attribute tags. The reason for this is that qualitative requirements commonly apply to more than one atomic requirement. The use of attribute vectors suggests that it is necessary to have one abstraction level, i.e. the qualitative requirements level, above the atomic requirement level. The tags provide two capabilities:

- Several instantiated requirements can be associated with a particular qualitative requirements type.
- The attribute vectors can be associated with the test suites; if an attribute value is present, the capability of executing the associated test suite exists.

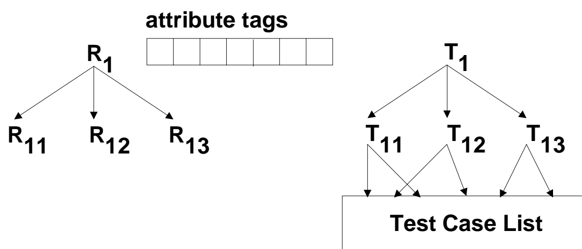

Figure 5. Requirements and test suites with attribute vectors.

In addition to the hierarchical structuring and the attribute tags, the authors propose rules for identifying a suite for regression test cases. The rules could be applied whenever the requirements are modified or new requirements are added. The problem is to find the minimum set of test cases for validating a new

product version. The rules are based on the requirements structure. An example rule is shown in Rule A.

Rule A. Select all test cases under the root that was modified.

However, these rules seem to be rather preliminary compared to the more sophisticated test suite controlling methods available (see, e.g., [Harrold et al. 1993]).

4.3 Some examples of the tools

Many companies producing test tools provide an evaluation version of their tool on the internet. Among the tools with interesting features from our point of view are, e.g., the following:

- RequisitePro at www.rational.com/products/reqpro/index.jtmpl
- Test Director at www.merc-int.com/products/testdirguide.html
- QA Director at www.compuware.com/
- WinRunner at www.winrunner.com/

RequisitePro is able to trace the product's requirements on the tests. Traceability relationships provide a direct link from a specific requirement to the tests that have been designed to ensure that the requirement is met (Figure 6). By using queries, it is possible to ensure the status of testing, what tests have been conducted and what tests still need to be executed.

The traceability from requirements (PR1 in the figure) to testing (TST1…TST10) is shown in the form of a tree. The user may also make queries about some specific requirement, and also save the results (the related test cases) for later use.

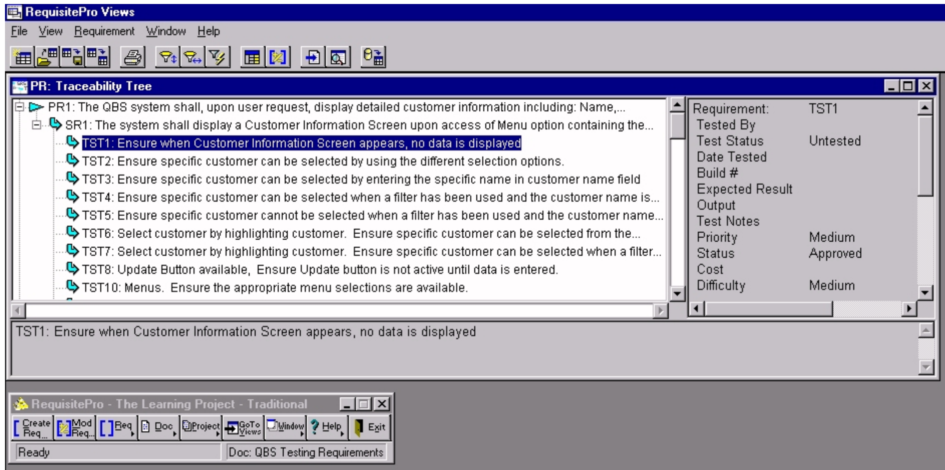

Figure 6. Traceability tree in RequisitePro.

TestDirector also links to any supplemental document related to a particular test case (Figure 7). You can access any file on disk or a specific URL over the Web. For example, the tester may create a requirements document using another program, such as Microsoft Word or Excel. TestDirector can then link to this document to provide traceability between the requirements attachment and the test.

The top level **QA Director** window provides a tree-structured hierarchical view of test plans, test case groups and test cases. In addition, it shows the history of each test plan. The test cases are marked with specific icons denoting whether they are manual tests, automated tests or test groups. You can also navigate from this view to the test results.

Test plans, test groups, and test cases are validated by a version number. By default, the references in the test plans relate to the current versions of the test cases if they are not set to a specific version.

Test plans can also be replicated to a new version and then edited in situ. A replicated version of a test plan initially contains the same versions of the test cases as the master copy. You can add test cases to a test plan by selecting them from a list of available test cases. You can not, however, reach the script itself, because it is maintained by the QA Run database.

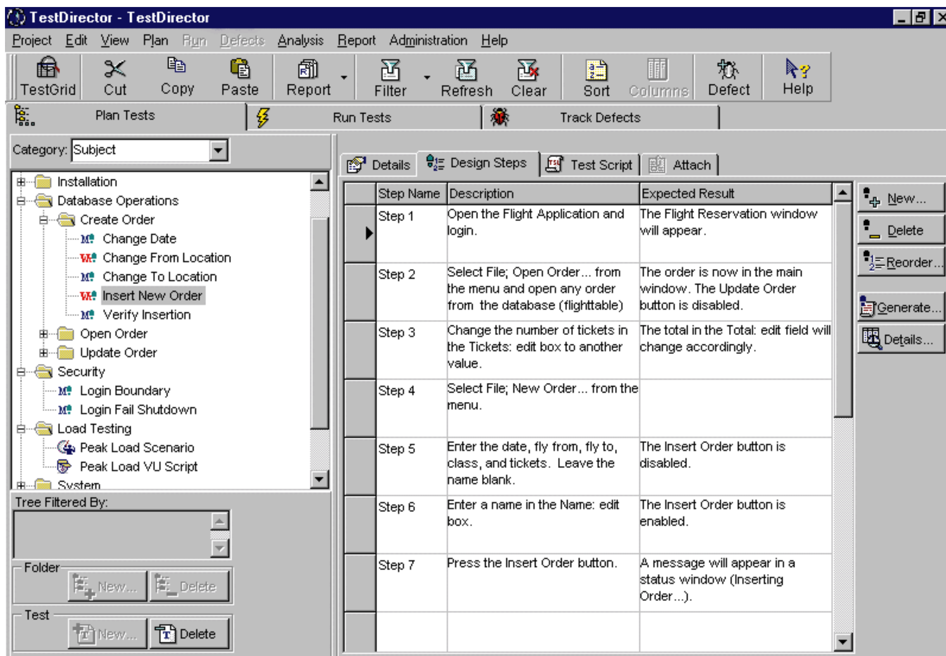

Figure 7. A view of requirements and tests in TestDirector.

Mercury Interactive has a testing environment called **WinRunner,** in which the testing is based on GUI elements. WinRunner is able to translate business processes into readable scripts, which can later be reused or replayed for verification.

To make sure that these scripts can be reused, WinRunner uses a GUI map to represent a repository of the application objects for each business process. The GUI map is created automatically when recording a test script. Each object within a test script has a minimum set of physical attributes that make it unique. As the GUI map is built, WinRunner captures the application object information and organizes it hierarchically.

5 Testing based on features and test components

5.1 The footing

Since regression testing and reuse in general constitute the foundation for the feature-based testing approach, we will take a short look at them before discussing the approach itself.

5.1.1 Regression testing

When a software product has been modified, regression testing techniques are usually applied to the testing of the software [Leung & White 1989]. Regression testing is a common name for a testing process which is applied after a change has been made to a program [Liu et al. 1993]. The change is initiated by a new requirement, which is why the program and its documentation specifications need to be modified. The goal of regression testing is to convince the maintainer that the program still performs correctly with respect to its requirements. This means that no new errors are introduced, and no unintended side-effects are caused by the change.

In this effort, regression testing reuses the test material that was used when testing the earlier products. The material may require some additions and modifications, but usually most of the material will do as such. Regression testing is thus a plain and simple process, which re-executes the slightly modified test suite.

The goals of regression testing include both testing the fix and making sure that no unintended effects have been caused. The following five steps summarize the activities of the regression testing process:

1. Identify the effects of the changes to the code (or both to the code and the specification).

2. Select the existing test cases and new test cases which will be used in testing the affected region.

3. Execute the modified program based on the selected test cases.

4. Ensure that the modified program still performs the intended function defined in the (possibly modified) specification.

5. Update the old test plan for the next regression testing process.

Figure 8 shows the regression testing steps, involving the reuse of the test case. Test cases are stored in a test repository. The definition and the process above reveal that regression testing is a method suitable for many types of testing, from unit testing to system and acceptance testing. The process requires management and tool support for submitting, retrieving, modifying and storing the test cases.

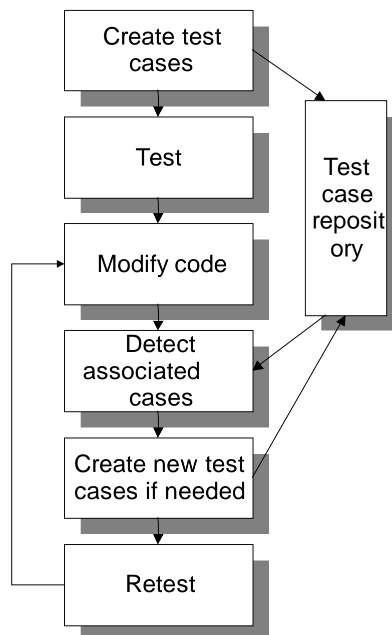

Figure 8. The regression testing steps in a software change process.

Considering the regression testing process, it is obvious that retesting the whole program, which might contain only a few minor changes, is expensive. The smaller the changes are, the greater is the desire to find only the relevant tests in the test suite. Thus the main goals of regression testing are to optimize the size

of the test suite, to produce reusable test material and to automate the execution of the testing.

However, the optimization of the test suite has its problems. Unless the reduction algorithm is not carefully considered and the optimization process supported by proper tools, the reduction of the test suite may leave out tests that will later prove necessary. This issue will be discussed in greater detail in the following chapters.

5.1.2 Test management

A management viewpoint to regression testing is presented in Figure 9. The picture outlines the activities which are involved in the process of controlling the test components [Salmela et al. 1999]. A support tool for test reuse should carry out most of these activities [Lewis et al. 1988].

Once the test components have been created they are submitted directly or through a review process to a test repository. The test components can be retrieved from the repository using queries or browse functions. The retrieved test components can be modified and assembled to form complete test sessions.

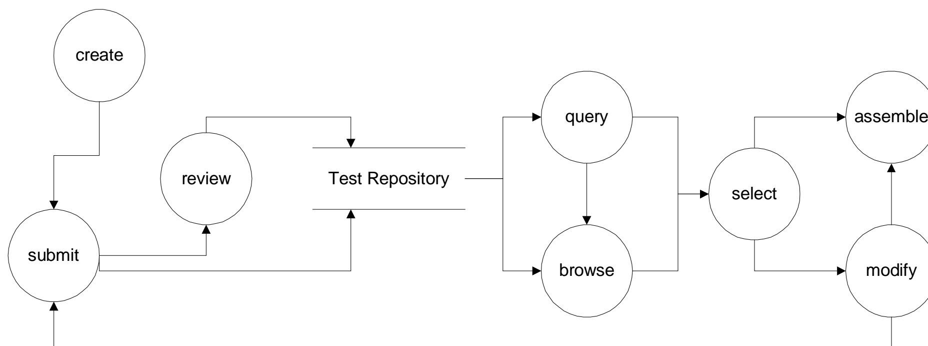

Figure 9. The test management process.

Test cases are usually developed either manually or by means of a test case generator, a capture tool or a recording tool. Generators usually produce test cases from code or from definitions (e.g. diagrams). Capture tools are used mainly in GUI testing to capture the user's actions on the screen. Also other message-based data exchange can be captured and recorded. The flaw in using

record/playback is that the system under test has to be brought to a definite state before the testing can be started. The expected results must also be programmed. The benefit of the record/playback approach is that after recording it is easy to use the tests in regression and load testing.

The test repository is the core of the support system, storing all testing-related items. A common way to implement the repository is to use a configuration management tool as a platform. The repository should be able to perform the retrieving and the submitting of functions, identity control, version control and linkage management (for instance, a link from designs to implementation and further to test documents). The configuration management system should also include a test repository. This method should facilitate the linkage management between software components and tests.

The tester should be able to assemble the test items to the test sets. A test suite manager is usually applied. It shows the available scripts, while the tester selects the relevant scripts for the new test compilation. For the compilation, the tester needs to arrange the scripts to form the various testing hierarchies and to define how many times a specific script is to be performed. The tester may also wish to select whether to playback one script or an entire suite of scripts.

Test cycle management is considered another important testing aspect in the Ovum project [Ovum 1998]. A test cycle describes the test steps as a process and the tool should be capable of assisting the user in implementing the process. The tools which advocate test-cycle management typically base their testing approach on the vendor's definition of the test cycle. The test cycle may thus define the entire mapping of the testing against the development lifecycle. Alternatively, it may simply describe the events associated with carrying out a discrete testing task.

The following list summarizes the features considered essential for the test management tools in general:

- Repository
    - Directory structure
    - Test suites
- Linkage control
- Configuration management
- Version control
- Automatic logging of test result
- Search facilities
- Maintenance facilities like adding new tests, modifying existing ones, and deleting redundant or useless tests.

List 1. The features considered essential in test management tools

5.1.3 Generic reusability issues in test design

Since regression testing concerns reusing existing tests, the issue of creating reusable tests is inescapable. The techniques for developing reusable designs share a common feature: they distinguish between a common part and a specific part. For instance, the ISO/IEC 9646-2 standard (OSI conformance testing methodology and framework - Part 2: Abstract test suite specification) mentions the possibility of splitting a test suite to a common and a specific part when using the method (Figure 10). The common part is dedicated to the elements or features which are always included in the software under test, while the specific part concentrates basically on the elements that may change or be totally excluded.

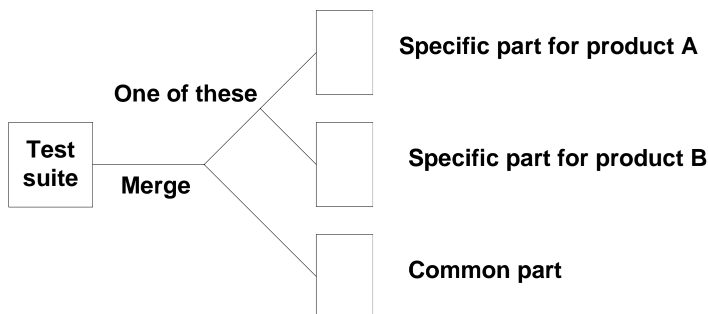

Figure 10. Combining common and specific parts.

The idea of common and specific parts can and should be introduced at the test case level already. Test cases should therefore be written in such a manner that the test body contains only sequences which are valid for testing the target software. The behavior related to the environment, e.g. software or hardware, should be hidden in preambles, postambles, or in other test components [EWOS 95]. The behaviour of the environment generally implies that the system has to be brought to a certain state before the actual testing can be initiated. These test initiation and ending sequences should be separated into their own specific test chunks.

Further general requirements on reusable tests include the following:

- Generality: a basic issue when reusability is being considered. Design for reusability demands an increased amount of resources. Is the suite generic enough to be used later so extensively that the extra cost can be recovered?

- Compatibility: the enhanced properties brought by reusability in the test suites should not cause any compatibility problems with the existing test material.

- Simplicity: the needed extensions and their application rules should be easily adapted to the current testing procedures. Any complicated solutions should be avoided.

- Modularity: the reusable design should support the modularity, for instance, in the description of how to reuse the testing material. Modularity is necessary for making test documentation readable and easy to maintain.

- Tool-applicability: the reusable design should be applicable to the test tools. The transition to a new testing procedure should not be too expensive for tool developers.

- Maintainability: the design should fit in with the maintenance and the classification of existing test material, and the design should also be open to future extensions.

The generic rules are often completed with application-specific rules, such as the following examples provided in the TTCN style guide [ETR 141 1994].

- Keep the number of the formal parameters small.
- Use parameterized test steps to ensure the reuse of the test steps within the test cases for different needs.

Although the rules mentioned above do give some assistance for designing the reusable tests, some important issues, such as the configurability of the tests, are not covered by the rules. Therefore, it is necessary to create a new kind of testing approach capable of coping with the validation challenges of configured software products.

5.2 The feature-based testing approach

The main idea with feature-based testing is to combine the potential advantages of feature-based production with those of regression testing. The purpose is to provide means enabling an efficient configuration of the tests and the testing environment. To fulfill the task, the feature-based testing method has to be able to answer the following questions:

- How to take the application domain knowledge into consideration in test design?
- How the information of the feature model can be used in adapting the tests to a desired software configuration?
- How the features of the software product can be used for selecting appropriate test material?
- How to take mutually dependent features into consideration in the test suite structure and in the test implementation?
- How the identified test material is developed into executable tests?

5.2.1 The feature-based testing concept

To clarify the feature-based testing concept, at first we need to define the relation between testing and product featuring. The relation has to be made clear, so as to be able to integrate the tests with the feature-based production process. We assume here that we already have a feature model and are able to configure

software products by using the model. Thus some basic facilities already exist for inserting the test material into the process.

The ERD-diagram in Figure 11 puts forward a proposal for defining the relationships. The relations between feature-based production and the different testing concepts are shown in the figure. There is a new term involved, *feature-model occurrence*, defining the properties for a single product. A feature model occurrence is a feature model from which the features not included in the product have been removed.

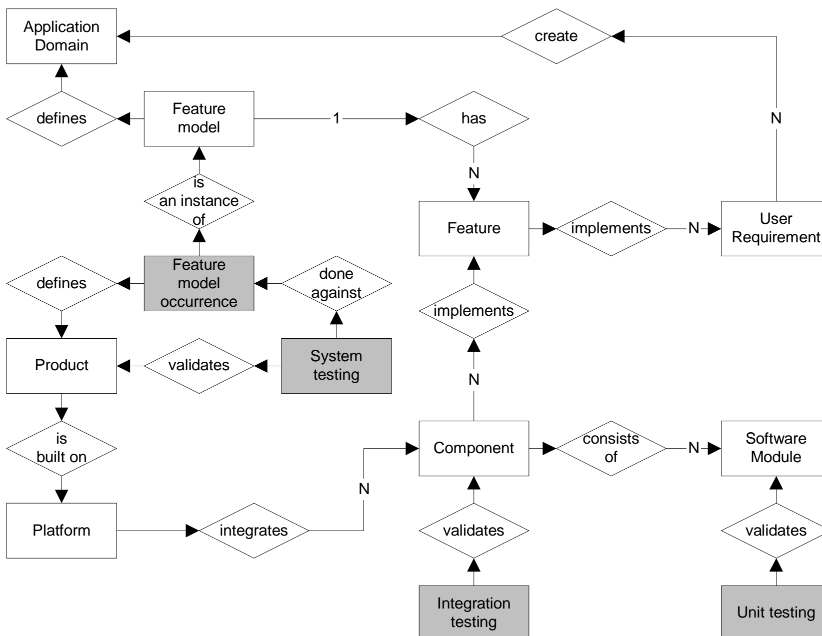

Figure 11. An ERD of the feature-based production and testing.

5.2.2 The test material

The material for testing a software product should include at least (the following list defines the minimum contents of the test material that is to be identified and managed during testing):

- A test script - a mechanism for automating the execution of the application in a controlled and repeatable manner

- test data – the input data required for making the application follow a specific execution path
- expected results - the state of some aspect of the application after a specified test case has been executed.

In addition to the previous items, the test material usually includes test plans and specifications, test environment configuration data etc. All the test data has to be stored and managed in a consistent manner. This is a major concern for any test organization, as the volume of the by-products of the testing process will grow rapidly by the progression of the testing project. Currently, very few testing tools employ the capabilities of version management or configuration management. However, many of the required controls can be implemented by using the conventional configuration management processes and tools.

5.2.3 Organizing and linking the test material with the feature model

Linking the test material with the feature model is easier if the material has been systematically arranged. There are several grounds for organizing the test material, for example,

- the test type (e.g. these tests are intended for *system testing*);
- the feature or the requirement linked with the test (e.g. this test is associated with the Display Refresh *feature*
- the type of the feature or the requirement linked with the test (e.g. these tests are intended for the *performance testing* of the Display Refresh feature).

All of the previous test organizing methods can be applied to the test hierarchy levels. Thus, for instance, the tests associated with the Display Refresh feature may be structured further. One possibility is the following, in which the tests are classified on the basis of system modes:

Test Suite (TS) for the Display Refresh feature

TS for initiating the feature in different system modes

TS for initiating the feature in normal operation

TS for initiating the feature in the sleep mode

TS for validating erroneous user input during the execution of the feature

Some other justification for classification, such as functional structuring, can also be applied. This means that the functional descriptions of the feature are used for structuring the tests. Tracing the procedure leads to a test repository structure resembling the one presented in Figure 12.

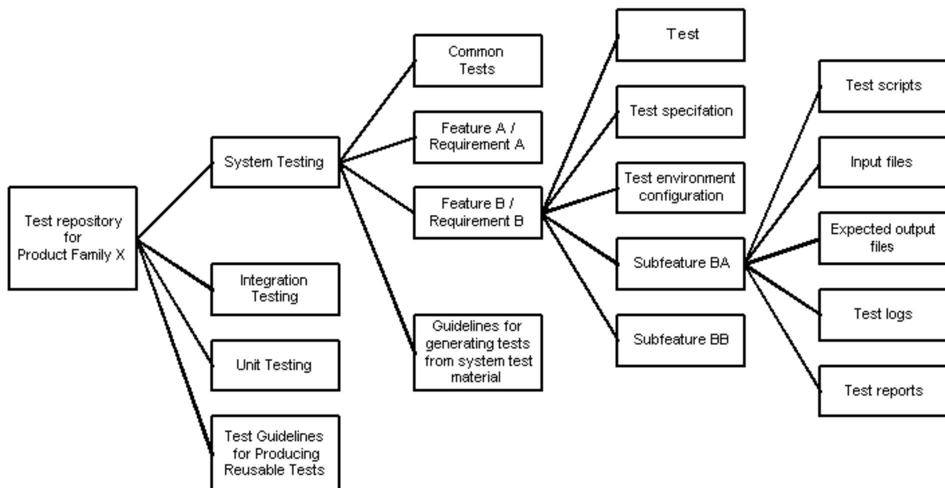

Figure 12. A test repository structure.

There are many ways to organize the test material. This is why the test method and the tools should have only few limitations on linking and defining the organization of test data.

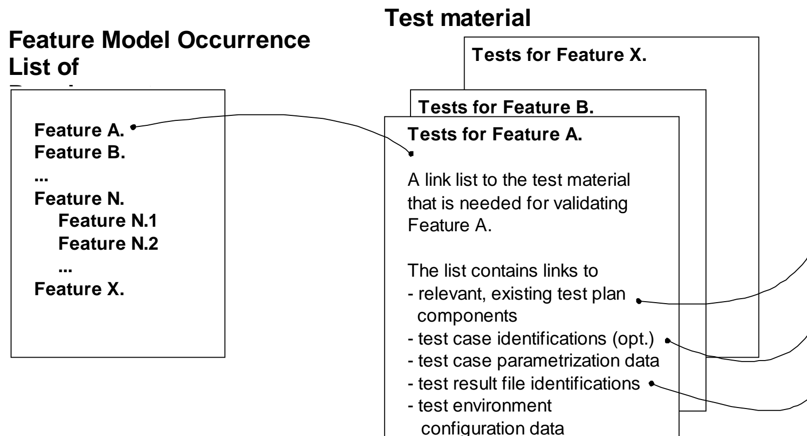

Figure 13. Linking the test material with the feature model or with the requirements.

As Figure 13 proposes, the user may find it necessary to link the test plans with the feature model or with the test environment configuration data. A free building of links should be made possible by the support system.

The features and other objects associated with the tests are subject to a continual change. This leads to changes in the application itself and, therefore, also in test plans, test cases, scripts and the data itself. This maintenance burden can be eased if the supporting system is capable of generating reports revealing the test material that may be affected by the changes. Updating the links and adding new material should likewise be easy.

5.2.4 The processing of the test material

The test material, now structured and linked with the feature model (or with other suitable objects, e.g. the requirements), has to be developed further for generating executable tests.

There are, in fact, several choices for processing the test material, depending on the test execution environment. It may well be that there are no appropriate test facilities available for employing executable tests. In this case, the only support for the testers is provided by a list of relevant tests and other test material. The testers will then execute the necessary tests *manually* (Figure 14).

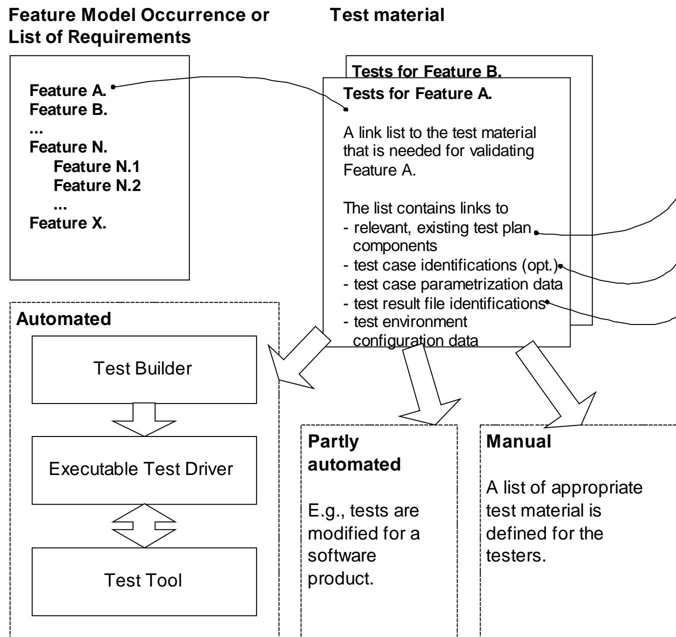

Figure 14. Solutions for test development.

If the test material processing is driven a bit further, the test supporting system could, for instance, provide modified test scripts and test environment configuration data for the testers. This could be called a *partly automated* solution.

In the most advanced case, there is a test environment capable of executing the tests. This being the case, the test material has been *automatically* configured in such a manner that the features of the product are considered by the tests and these can be executed without any additional, manual processing.

For automating the test execution, it is necessary to consider the implications a feature model has on the test material. It may well be that the features have, for instance, mutual dependencies, and therefore a solution involving a single *static* test suite used for testing one feature is not reasonable. The feature model occurrence has to be used for selecting and modifying the test suites for the product.

Definition 3. The features are mutually dependant if the selection of one feature changes the behaviour of the other.

Let us assume that we have included feature *F1* and some other features in a product configuration and the resulting product needs to be validated. Test scripts *S1... Sn* and corresponding result files *R1... Rn* have been developed for the domain. The domain analysis has revealed that the feature *F1* changes its behaviour depending on the feature combination. In test design, this could appear as follows:

- If *F1* is selected, the script *S1* is run for testing *F1* and results defined by *R1* are expected.

- If both *F1* and *F2* are selected, the script *S1* is run for *F1*, but the expected results are now defined by *R2*.

- If both *F1* and *F3* are selected, the script *S2* is run for *F1* and *R3* is expected.

The idea is that testing of a feature may require several alternative test scripts and test result files. The selection of appropriate tests depends on feature configuration; *F2* and *F3* in the example above. The implementation of the test material structure and the test configuration has to be able to handle the conditions described above.

Compared with other software models, feature-based software production offers certain advantages regarding test development. According to [Nilson et al. 1994] domain analysis in the software process includes the following tasks: "carefully bound the domain being considered, consider the ways the systems in the domain are alike (which suggests required characteristics) and the ways they differ (which suggests optional characteristics), organize an understanding of the relationships between various elements in the domain, and represent this understanding in a useful way". The next step is domain modeling, which "provides a description of the problem space in the domain that is addressed by software" [Krut & Zalman 1996]. The results of these tasks are applicable to the test design as well, especially for identifying the instances for reusable test components. The models define the user point of view to the product and in

some cases the test components can be directly extracted from the Use Case descriptions.

In fact, use cases and test cases work well together in two ways: if the use cases for a system are complete, accurate and clear, the process of deriving test cases is straightforward. And if the use cases are not in a good shape, the attempt to derive test cases will help to debug the use cases [Collard 1999].

Some software processes apply the commonality analysis of the Use Cases during the definition of the requirements. The analysis surveys and identifies the common parts of the Use Cases as the use case steps. If this information is available, the common parts are already known and the test component design is simple and straightforward. If the Use Case descriptions do not exist, it is reasonable to start by creating the descriptions and proceeding with the test component design from that point on.

Figure 15 illustrates how test cases are extracted from a use case. For each action in the use case there is a corresponding test step. Common actions (e.g. related to hardware or software environment) can be hidden in preambles, postambles, or in other test components. These actions are needed for bringing the system to a desired state and to initiate or to end the actual test. Specific, feature-related actions are separated in specific test steps or components.

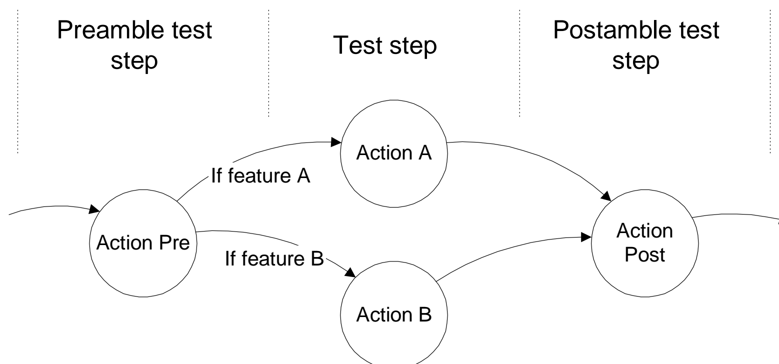

Figure 15. Abstracting test steps from a use case.

Use case descriptions are practical in modeling human user sequences. However, systems have various types of interfaces and are seldom stimulated merely by

human users. A sensor or some other system is possibly used as a communicating device. In simple cases the external device can be modeled using a static input-file, but devices with complicated behaviour need executable, dynamic models. These input-files and even system models can be handled like the other test components; e.g., they can be parameterized and included in the test configuration when needed [Haapanen et al. 1997].

Joining test case scripts arbitrarily may not produce the desired result. When testing integrated components, one component presumes that the next component is in a specific state. Therefore, *test controlling scripts* need to be generated on the basis of the events and/or test steps which form a continuous line of different states [Jeon & von Mayerhauser 1994].

Test components can be structured as presented in Figure 16. The purpose of the method is to hide the numerous test steps in the test specifications and to make test planning easier. The specification (i.e. test controlling script) defines one use case step, but does not contain the actual test script. The actual script is located in the component implementations. The alternative implementations are called by the specification component, which uses feature selections to determine the right implementation. Direct calls to implementations are not recommended, as they may lead to a loss of the reusability of the components and thus increase the maintenance load. This solution is similar to the "information hiding" principle, according to which unnecessary details are hidden from the user.

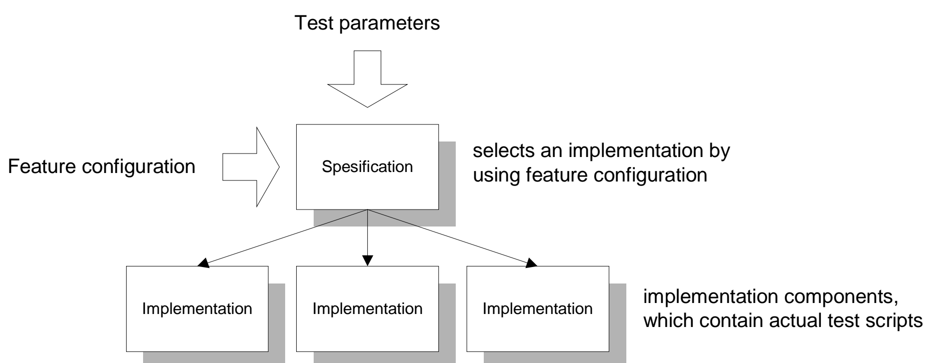

Figure 16. The test case componentation hierarchy.

The selection of a test component implementation, i.e. the test script, is defined in the specification using *if-then* clauses. The most general configuration is put into the last *if*-block and the most infrequently used one on top of the list. When the *if* -clause is used in this way, a new feature can be added into the system as a new specialization, while the existing mappings/expressions remain the same.

```
If (most specific feature configuration) then
   select script file 4
else if (specific feature configuration) then
   select script file 3
else if (common feature configuration) then
   select script file 2
else if (most common feature configuration) then
   select script file 1
else
   select default script file component
endif
```

Example 2. Selecting a proper test implementation.

Figure 16 presents only one level in the test hierarchy. The test structuring may be continued to the desired level by applying the principle of sub-scripting. A level may contain only the identifications for the next, lower level test scripts.

5.2.5 The taximeter example

Figure 17 describes a sample taximeter test case, *Cash payment 100 units*, which simulates driving a taxi trip with *fare 1* charge. *Cash payment* is simulated at the end of the trip. The test case specification file is the top-level script that calls the feature selection file and the three test component specification files.

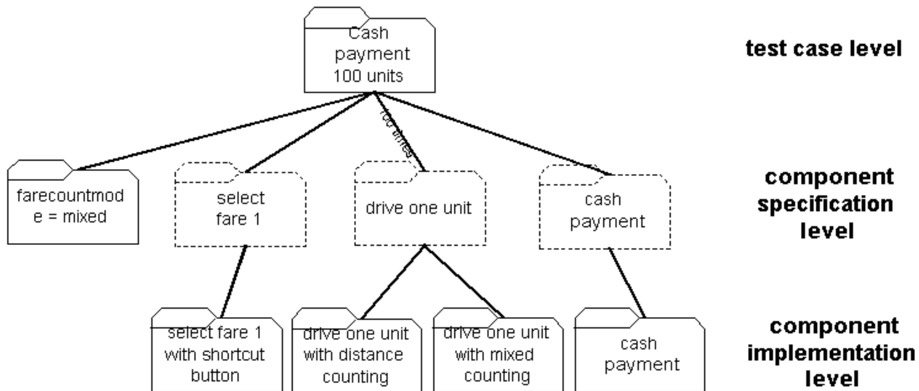

Figure 17. Sample taximeter test case.

Select fare 1 is a test component specification with one implementation file. The test component simulates the situation of a customer entering a taxi and the driver selecting manually *fare 1*. The *drive one unit* component simulates driving until a fare unit is counted. The component will be executed 100 times so that the total fare will be 100 units. The component specification has two alternate implementations, which are selected depending on the feature selections. The cash payment component simply simulates normal cash payment.

5.2.6 Generating tests

The need of generating tests usually depends on test stopping criteria. If the system is simple or the test set compact, the testing is usually stopped when all the tests have been executed successfully. In some cases, e.g. if a statistical requirement on test stopping exists, the static test sets are out of the question. In this case the test specifications can be enhanced as dynamic user models. Usage profiles or other coverage criteria for implementing dynamic control in test specifications can be applied.

Another way of generating test is to employ usage modeling, by using the Markov chains or the Finite State Machines, for instance. These approaches can be applied for generating numerous stochastic tests imitating the way the system would be used in an actual situation.

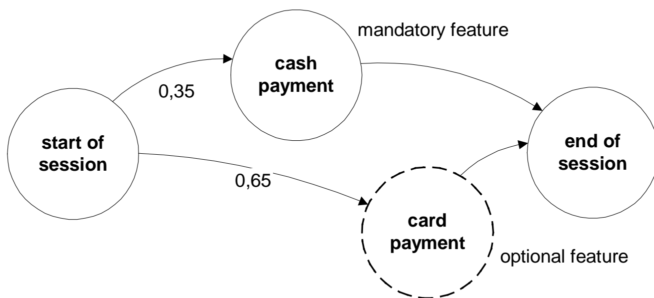

Figure 18. A usage model with probabilities.

Figure 18 shows an example of a usage model. The graphical model can, if done by using a suitable description method, be compiled or executed as such. It is evident that Markov models are very similar to use case descriptions, with the exception of probabilities being included in the graphs. Thus use cases can easily be enhanced to Markov diagrams (Figure 15 and Figure 18). The model can be used as a top-level test controller. The probability conditions can also be manually coded in the test specification (see Example 3).

The label *0,65* on the arrow means that the probability for the transition is 0,65 if the optional feature has been selected. The implementation of probabilities is easily accomplished by using a random generator with an even distribution between [0, 1].

5.3 The optimization of test suites

We have noticed that the test suites for validating a product always contain *all* the test cases associated with the selected features. Depending on the product and the application domain, this may involve a considerable amount of time and resources. One may wonder if the testing of the earlier, relatively similar products could somehow reduce the effort of testing the new products. Even though various optimization solutions have been proposed, the problem is not easily solved.

Optimization needs are highly dependent on the application domain. A good example of a potential optimization domain in regression testing is data communications, where the context may include more than 20 million lines of

code with over 50´000 discrete components. In such a case, the time needed to run a fully automated regression test would be several weeks. A manual analysis of which components need to be included and which ones can be left out safely is simply not feasible. In this context, even the early algorithms by Leung and White [1990] will have a tremendous payoff. The most recent reports state that in the case above it has been possible to cut the total of tests down to 5% - 10% of fully repeating the tests [Beizer 1999].

Since feature-based testing is relatively similar to ordinary regression testing, the idea of applying test reduction techniques becomes very tempting. Situations may arise where all the software components have already been used in earlier products. A full testing of such a product seems waste of time (assuming that the product is large and complicated enough, hence requiring a large test set). But is this actually the case? What are the requirements for the test reduction techniques and are they easy and straightforward enough to use?

Von Mayrhauser [et al. 1993] have proposed that the hierarchical structured tests linked with the requirements could be used as a source of rules for identifying a suite of regression test cases. The rules could be applied whenever the requirements are modified or new requirements are set. The problem then is to find the minimum set of test cases for validating the new product version. The following are examples of the rules to be applied when an addition has been made:

Rule 1. Select all test cases under the root that was modified.

Rule 2. Select all test cases related to the requirements that trace to the code affected by the new code.

However, these kind of rules are not reliable and, furthermore, they may even prove harmful to test set reduction. A similar heuristic might suggest that we skip all the tests that are 'far from' the modified code. The new product might, for instance, contain features used in earlier products with only one additional feature selected. The situation could possibly make us concentrate on the tests directly related to the new feature and omit the tests used for verifying other features.

However, there is no guarantee of the *safety* of this kind of procedure. In this context, the term *safe* should be interpreted as follows:

Definition 4. A safe regression test selection technique selects all the test cases that can reveal a fault in the modified system. [Rothermel & Harrold 1996]

In general, the safety requirement leads to analyzing the program's internal structure and execution paths, i.e. to various kinds of coverage measurements. As attractive as the possibility might be, we can not safely reduce the regression test suite by examining the program through its external interfaces by using it as if it was a black box.

The knowledge on earlier systems can, however, be used to some extent in testing. The information is generally exploited in the assessment of the systems that have to meet stringent reliability requirements. The tendency in reliability assessment is shifting from human judgment towards mathematical approaches. An example of this is the study of Littlewood and Wright [1997], who estimate the reliability of a new product by using the evidence of previous comparable products[3]. The authors have applied the Bayesian model to calculate the increase in the probability of a failure-free operation for a new system when previous, similar systems have operated faultlessly. They have proved, for instance, that if we have three similar systems that have operated without failure 10^7 times, the new system will have a 0,75 probability to survive the same amount of inputs, i.e. 10^7. This piece of data is useful when estimating the test stopping criteria for a new system.

However, the mathematical approach has some obvious drawbacks. How should we define the similarity between two comparable products? And, even if we could quantify the similarity, e.g. relying on experience, the increase in confidence is still rather small. Furthermore, the procedure is valid for large,

---

[3] Littlewood and Wright discuss the reliability assessment, which seems to have no direct relation to testing. However, reliability assessment is based on system testing, using test results in predicting reliability.

stochastic test suites and may not be applied without encountering problems in general.

It is obvious that the heuristic and the mathematical approach both suffer from obscurities, due to which a better method is called for. An excellent overview and analysis of the current regression test selection techniques can be found in [Rothermel & Harrold 1996]. In the report, the popular test selection techniques (13 in total) are analyzed and it is shown that many of the techniques do not meet the requirement of safety (see Definition 3). The evaluation is useful in the sense of clearly identifying the strengths and weaknesses of the various techniques. This provides help in selecting an appropriate test reduction technique for a specific purpose. The referenced techniques analyze the structure of the code by using various methods, which is also indicated in the names of the techniques: *Path analysis*; *Dataflow*; or *Program dependence graph*.

However, many of the techniques have not been implemented and even fewer have been subject to empirical studies. Thus, there are no commercial tools to be readily applied. Some companies have implemented tools by themselves. This can be done, as the techniques are openly presented, while the required effort is still considerable. Furthermore, as discussed at the beginning of this chapter, the test selection techniques are suitable for certain application domains, where the size of the systems and the number of the software components is large. However, commercial solutions are well on their way and probably within the next ten years there will be some off-the-self systems available for test selection and configuration.

From the discussion above, the following conclusions can be drawn:

- Heuristics is a commonly applied method for test reduction, but it is not safe. (We may associate test cases to a certain feature, while we can not guarantee that one feature does not affect the other features; thus additional testing is needed.)

- A safe test reduction method requires verifying the internal structure of the program (usually not reasonable to carry out manually).

- The safest test reduction method is to do no reduction at all; this is feasible in cases where the size of the software is fairly small and automated test execution environment is available.

6 Support for the feature-based testing

Let us assume that our application domain is suitable for feature-based testing and we wish to apply the described testing procedures; what would this require? Some of the requirements have been discussed earlier in the document, but a more concise formatting is shown in this chapter. The list of the requirements is by no means final, since different aspects can be emphasized or otherwise modified depending on the user's needs.

6.1 Script development

A script can be described as a sequential set of commands which mimics the normal controlling input to the application. Thus a script can be a reproduction of a keying sequence from the user for screen manipulation.

A script language needs certain attributes, listed below, to be able to implement hierarchical and modifiable test suites:

- logical condition constructions
- data manipulation
- external file input/output commands
- procedure calls with parameters

Since most script languages simply adapt to the existing languages, e.g. C or Visual Basic, the requirements above are not hard to meet.

6.2 Test development and execution

Test development denotes the task of converting the test scripts into executable tests. At this point, we have a set of test scripts to be transformed to an executable format. Thus the test development has to support

- the linking of test scripts to features, and
- sequenced running of the test scripts from a test controlling script.

A test controlling script may be necessary, e.g. for controlling the execution order of the scripts. The test execution tools typically support this requirement. However, the first requirement has to be implemented otherwise, since no tool support for that requirement is available. The issue is dealt with in detail in [Salmela et al. 1999].

Joining the test case scripts arbitrarily may not produce the desired results. Individual test cases always focus on one feature, function, object or component. The components have inner states and variables. When testing integrated components, one component presumes that the second one is in a specific state. Therefore, the test controlling scripts need to be built up using the events/test steps which form a continuous line of the different states [Jeon & von Mayerhauser 1994].

The test execution may also contain other functions, such as recording capabilities or the comparison between the expected test results and the actual response, but these issues are not of great relevance from our point of view.

6.3 Test storage and management

Test execution tools usually have no support for normal configuration management features. Even though this is not an essential characteristic in feature-based testing, this issue should be brought into the implementation when considering real-life applications. If no support for configuration management exists, it might be a good idea to integrate the test tool with a configuration management system (see for instance [Desai 1994]).

6.4 A support system for test case reuse

A test case reuse and management system is at its most useful when it can be flexibly integrated with other testing and programming systems. In general, the support system for reuse should offer an easy and effortless connection to the test repository, from which the test designer is able to retrieve the necessary components [Griss et al. 1997].

Requirement The support tool needs to be built on a version management system which takes care of test repository maintenance and linkage control.

If the tool has only one user, or if the test components are used only by one person or on one workstation at a time, the repository can have a lighter structure. The test components can be stored, for instance, in the directory structure, which is organized according to the primary search structure (project, release, delivery, and component).

If several persons need to use the same test repository, the support system should be implemented in a client/server environment. A simplified solution is to implement the directory structure on a shared hard disk. However, controlling and delivering the files with a dedicated server software is a safer solution.

Requirement The user can search test cases using the different search methods and browse the test cases.

The test components should be archived with additional information to facilitate the searching and maintenance tasks. The hierarchy of the repository should be arranged so that it corresponds to the primary search structure. For instance, the main levels could be named after the project or product.

Requirement The user can modify and rearrange the retrieved tests using a suite manager.

A searching and browsing tool is needed for retrieving the test components from the repository. The search tool may use categories, keywords, application/SUT and project names as search criteria. After searching or viewing the user receives a result page, which shows links to the test objects and to other related information, such as how to use the items. The search engine can be implemented by using HTML, Java, or other web-based technologies.

In addition to the previous generic characteristics, the support system needs to have the ability to link test scripts, and test material in general, with the features of the product family. An equally important issue is the capacity of linking tests with product requirements.

Requirement The test development has to support the linking of test scripts to features.

Requirement The test development has to support a sequenced running of the test scripts from a test controlling script.

The latter statement means that the supporting tool is able to execute series of tests in a controlled manner, taking into consideration the mutual dependencies the features may have. Another equally important issue concerning the test controlling script is the control of the execution order of the scripts.

6.5 The test process

The so-called V model is a commonly referenced software development model at present. According to Figure 19, both system testing and acceptance testing utilize the various requirement documents in test design. Tests are developed on the basis of the usage models presented in development documents, e.g. as use case descriptions.

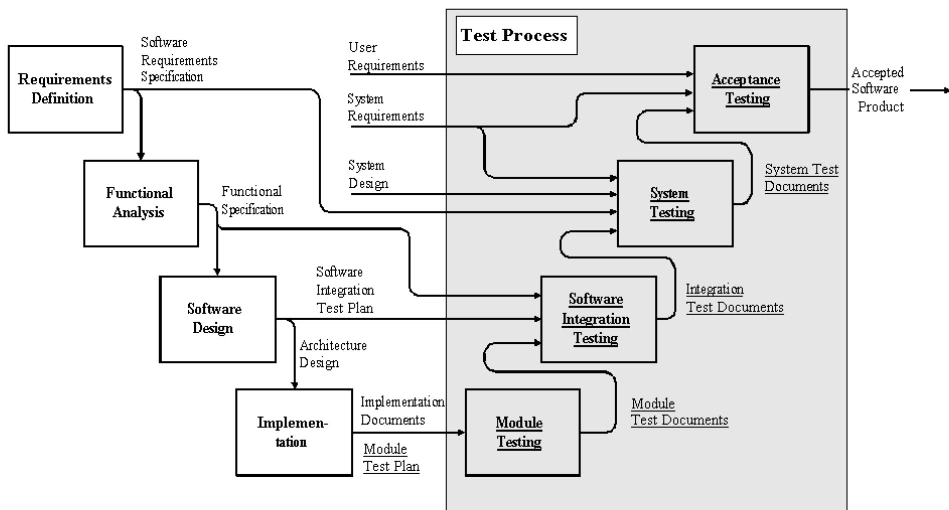

Figure 19. The V-model.

This kind of test development can also be applied to feature-based testing. However, a test domain analysis may be necessary for identifying the generic test sequences (i.e. components) before entering the test design. Otherwise, the process is similar to the test processes mentioned, e.g. in Atkins and Rolince [1994]. The Test Requirements Specification Language standard (IEEE P 1029.3 TSRL), which they refer to, defines the following tasks for a generic test process:

- the characterization of the test subject
- the definition of the test objectives
- the definition of the test requirements
- the selection of the test method and the test resources
- the generation of the test procedures.

These activities can be applied as such to the feature-based test process.

7 A case study

This case study puts into practice some of the ideas presented in the previous chapters. For the purpose, we constructed a simple demonstration system, consisting of the typical IO devices included in an embedded system: a keyboard, a display and a card/bank card reader. The case is loosely based on two real embedded systems [Kalaoja et al. 2000]. The main emphasis of the case study is to show in practice how the test material configuration can be implemented.

Thus, the demo system aims at presenting

- how the test data is managed and organized;
- how the feature model is used for test material selection;
- how the feature model is taken into consideration in the test design;
- how the selected features are brought and applied to the test scripts;
- how the mutually dependent features are taken into consideration in the test suite structure and the test implementation; and
- how the identified test material is developed into executable tests.

7.1 The system

The system was expected to include characteristics which would bring out potential problems in the implementation of the ideas, and, at the same time, the system would provide a suitable 'test environment' with enough functional variety. Since the demo system was fictive, only a limited specification of its functionality was needed. The system is intended for placing bets, e.g. in horseracing. The keyboard of the system contains a few function keys and a numeric keypad. The system has a resemblance to automatic teller machines (Figure 20).

The system has two main operation modes: maintenance (e.g. for display adjustments) and bet placing.

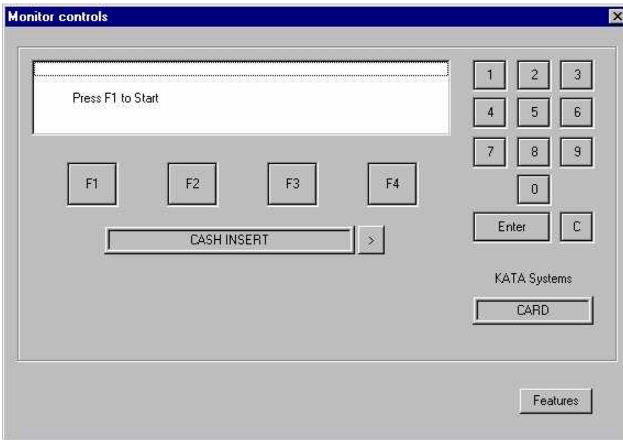

Figure 20. A view of the system window.

There is a set of functions for display adjustments linked with the various displays. This is the reason why several display models are available for the system. The act of bet placing consists of a sequence of actions, including bet entering, the selection of the payment method, as well as the payment actions themselves. The features can be combined fairly freely, while a number of different products can be built by using the feature-based production technique.

7.2 The feature model of the system

The KataSystems demo environment consists of three components: a betting device, a tester and a configurator (Figure 21). The configurator is used for modifying the outlook and the behaviour of the slot machine. When the user configures the slot machine, the tester is loaded with tests that are connected to the selected features. The tester is used to execute and to monitor the selected test.

The possible variations of the slot machine are shown in a structured tree-like feature model (see Figure 22). The EON (Embedded Object Notation) method [Kalaoja et al. 1997] was applied to the feature model development. In the diagram, the features are shown as boxes with an identification or, optionally, with a value. The relationships between the features, e.g. feature and its subfeature, are represented as arrows. The mutual dependencies are also shown as arrows containing the *req* attribute.

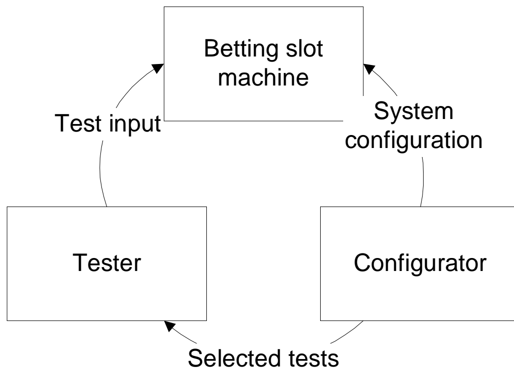

Figure 21. The KataSystems demo environment.

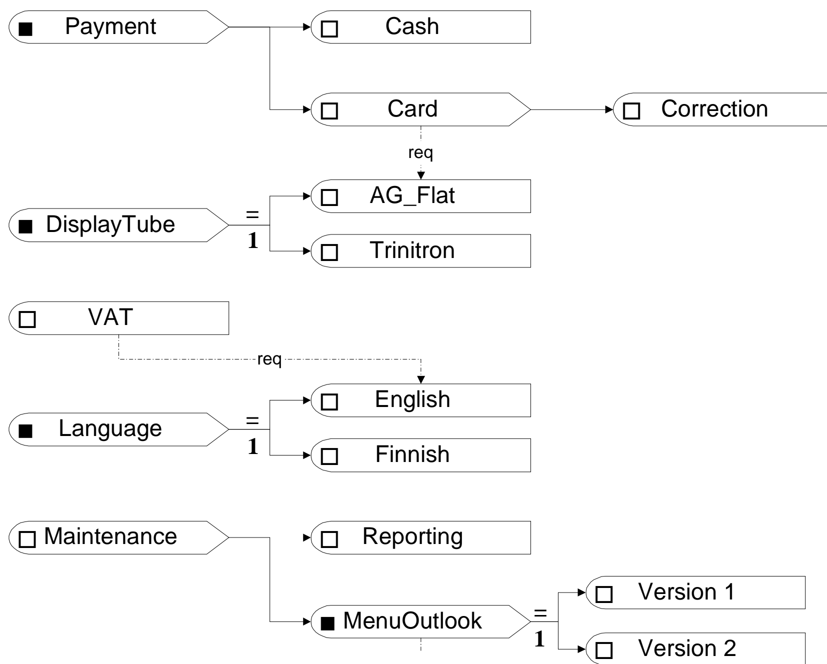

Figure 22. The feature model of the slot machine.

The model shows that the slot machine always has a cash payment feature, but there is an option for credit card payment. The credit card payment has an additional option for payment correction, i.e. the bet can be changed. In addition, there are further options for, e.g., display types and language.

The feature model is converted to the feature selector, which is implemented by using check boxes and radio buttons. After the user has configured the system, the tester is loaded with tests corresponding to the selected features. The tester employs only the tests needed to carry out the testing of the configured product.

Figure 23 presents the layout of the configurator. Mutual dependencies are shown as arrows referring to the other features. The user or the one configuring the system can create different kinds of systems by clicking the desired features and pressing the OK button.

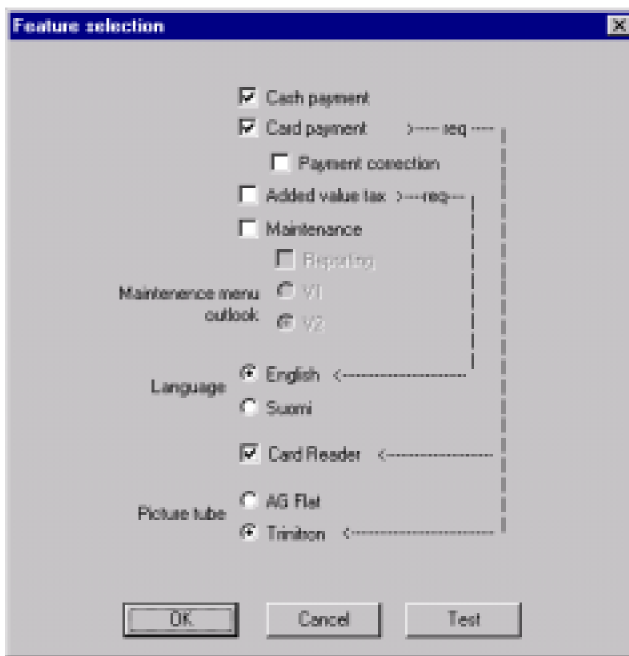

Figure 23. The feature configurator for the demo system.

The selected features are imported to the test script as an *include* statement. The control of the testing of the correct features is gained by using the data of the *feature_model.h* file in the conditional statements. Test data is stored in files and it can be made readable by using special routines. Note the implementation of a usage profile using a random generator (see also Figure 18).

```
/*  The feature configuration of the application  */
#include "feature_model.h"

/*  Test parameters  */
get_test_params( PAYMENT_TESTING, number_of_tests, *bet_sums );

/*  Test execution  */
for ( i=0; i < number_of_tests; i++)
     {
     /*  Preamble  */
     start_user_session;
     enter_bet( *bet_sum, i );

     /*  Feature configuration controls test selection  */
     /*  If both features included, apply usage profile */
     if ( CARD_PAYMENT && (uniform_distribution(0,1) <= 0,65 ))
          { card_payment; }
     else
          { cash_payment; }
     /*  Postamble  */
     enter_keycode( STOP_PAYMENT_SESSION );
     }
```

Example 3. A part of the main test controller for the bet placing feature.

In the demo system, we used the C programming language as a test scripting language. C is powerful enough for creating reusable test components and functions. It also has an ability to pass parameters to the test components, which was one of the main requirements for the script language.

Table 4. Test suite correspondence to features.

Feature	Test suite
Payment	payment_test
Cash	cash_pay_test
Correction	payment_correction_test
DisplayTube	display_test
AG_Flat	ag_flat_test
feature list continues...	

The test suite hierarchy for the demo follows the structure of the feature model. The list in the left column of Table 4 contains all the features, and on the right you will find the corresponding test suites.

A simple way of implementing links between the test material and the feature model is to use a test index file. The index file maintains links from the features / functions / issues to the test files, e.g. the test specifications and scripts. When the user makes selections in the feature selector, the necessary tests are searched from the test index file. The test index file represents 'a small database' maintaining links between the features and the tests.

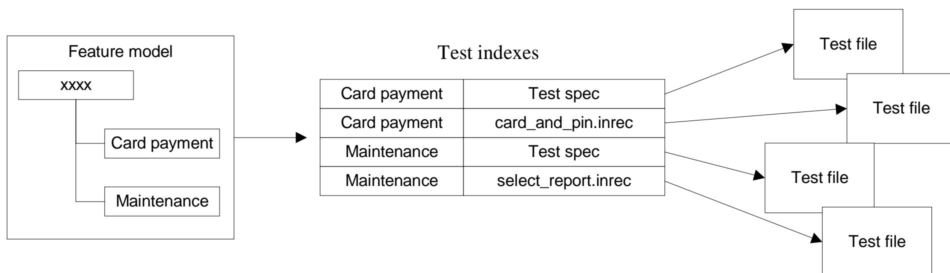

Figure 24. The test index file.

7.3 Writing reusable test components

The C programming language has been used as a test scripting language. Using C makes it is easy to write reusable test components and functions. Furthermore, C has the ability to pass parameters to the components. This approach indicates

that the scripting language should have structural programming capacities. This chapter contains some examples of the test design.

Example 4 presents a lowest-level testing component, implementing the connection to the betting slot machine. The component simulates manual button pressing. The button to be pressed is indicated with the input parameter.

```
void press_button( int key )
{
    SendMessage( GetDlgItem( hWnd, key ),
            BM_SETSTATE, TRUE, 0L );
    Sleep( 200 );
    SendMessage( GetDlgItem( hWnd, key ),
            BM_SETSTATE, FALSE, 0L );
    SendMessage( hWnd, WM_COMMAND,
            MAKEWPARAM( key, 0 ), 0L );
}
```

Example 4. A low-level component.

After developing the lowest-level components we can define more generic and readable components using such basic components as *press_button*. Example 5 shows a few higher-level test components.

```
void press_number( int num )
{   switch( num )
    {
    case 1:
        press_button( IDC_B_1 );
        break;
    case 2:
        press_button( IDC_B_2 );
        break;
   ...
}

void start_betting( void )
{   press_button( IDC_F1 );  }

void press_enter( void )
{   press_button( IDC_B_E ); }
```

```
void type_number( int num )
{   int     len, i;
    char    szNum[ 7 ], c;

    itoa( num, szNum, 10 );
    len = strlen( szNum );
    for( i = 0 ; i < len ; i++ )
    {   c = szNum[ i ];
        press_number( atoi( &c ) );
    }
}
```

Example 5. The test components.

In the previous example, the *press_number*, *start_betting* and *press_enter* components are implemented by using the *press_button* component. *Type_number*, a further component, is constructed by using the *press_number* component.

If the script language of the tool offers this kind of structural programming abilities, it is likely to provide a very efficient method for creating reusable testing components. Furthermore, the ability to pass the parameters is essential for the reusability and versatility of the components.

7.3.1 The variation points in testing

This example shows how to make implementation-specification pairs. In the example, the function *display_test* is a component in the specification which selects the correct implementation component according to the desired configuration. In this case, the implementation components are *trinitron_test* and *ag_flat_test*.

```
void display_test( void )
{
    if( valid.iTube == TRINITRON )
    {   trinitron_test();  }
    else
    {   ag_flat_test();    }
}
```

Example 6. A specification component for testing the display.

In the following example, a variation point is used for selecting the correct test input file. At the *bet_test* the test input is written in separate files. Using the separate input files provides an efficient way of writing large amounts of test inputs. In this case, different inputs are needed, because the maximum bet sum and the amount of the bets vary between the displays.

```
if( valid.iTube == TRINITRON )
{
    // Open Trinitron test data file
    create_file_name(
        "testit\\bet_test_trinitron.dat", &file );
}
else
{
    // Open AG Flat test data file
    create_file_name(
        "testit\\bet_test_agflat.dat", &file );
}
```

Example 7. Loading the test input files.

8 Summary

In general, reuse has been connected with software development and production through software component design. In testing, the method of reuse seems to be more of an ad-hoc nature, inspired by practical considerations. More formal reuse procedures can be found in telecommunications, where the standards give generic advice for reusable test design.

The feature-based testing method proposed in this research report is intended mainly for systems which apply features for defining their properties or which at least build products upon software components. The approach outlines the essential characteristics for test reuse. Among these issues are test suite structuring, test script design and test material management. Some of the ideas have been presented earlier, even though not necessarily implemented or evaluated as we did. This paper integrates the relevant ideas and reviews the requirements for a test environment to support the approach.

The key issue of the approach is how to structure tests for the test components. The components have to be configured in such a manner that the tests match the features of the product. The test domain analysis, which can clearly draw upon the feature domain analysis, has to be done prior to the structuring of the test, since the test components are identified through the analysis.

An empirical evaluation of the results is essential, especially since the case has a strong practical point of view. A demonstration system was constructed to implement the essential features and ideas presented in the paper. Another, equally important, purpose of this study was to evaluate the functionality and the relevance of the proposed techniques and solutions. For the demonstration system, the designing of tests was straightforward and easy, and the configuration of the tests revealed no major problems in the approach. At best, presuming that a new product is based on already existing software components, all the necessary tests can be identified and configured by using the feature-based testing approach.

Unfortunately, we did not have the opportunity to evaluate the approach with a real product family. However, we believe that the ideas presented in the report can be used as such to a great extent. We presume that a practical

implementation of the technique demands a more professional approach. The demonstration tool described here is merely an exercise, having only little practical value. We are looking forward to the ideas being brought forward in new research programs to be initiated in the near future.

References

Atkins, R. & Rolince, D. TRSL standard supports current and future test processes. In proceedings of AUTOTESTCON '94 conference. IEEE. 1994.

Beizer, B. Personal correspondence. 1999.

Collard, R. Developing test cases from use cases. In Software Testing & Quality Engineering, July/August 1999.

Desai, H.D. Test case management system (TCMS). In proceedings of 1994 IEEE GLOBECOM. Communications: The global bridge. IEEE. 1994.

ETR 141. Methods for testing and specification (MTS) Protocol and profile conformance testing Specifications: The Tree and Tabular Combined Notation (TTCN) style guide. 1994.

EWOS/TA. Methods for testing and specification (MTS) partial & multi-part abstract test suites (ATS), rules for the context-dependent reuse of ATSs. EWSO/ETG 057. 1995.

Griss, M.L., Favaro, J. & d'Alessandro, M. Featuring the reuse-driven software engineering business. A draft for Object Magazine. September 1997.

Haapanen P., Pulkkinen, U. & Korhonen, J. Usage models in realiability assessment of software-based systems. Finnish Centre for Radiation and Nuclear Safety. STUK-YTO-TR 128. April 1997.

Harrold, M.L., Gupta, R. & Soffa, M.L. A methodology for controlling the size of a test suite. In ACM Transactions on Software Engineering and Methodology, Vol.2, No.3, July 1993.

Jeon, T. & von Mayrhauser, A. A knowledge-based approach to regression testing. First Asia-Pacific Software Engineering Conference. IEEE Comput. Soc. Press, 1994.

Kalaoja, J., Rytilä, H. & Salmela, M. The final report on the Semel pilot case. A KATA-project report. To be published in 2000.

Kalaoja, J., Toivanen, J., Okkonen, A., Niemelä, E. & Ihme, T. Configurable feature-based application software. KOMPPI-project report. VTT Electronics. 1997.

Krut, R. & Zalman, N. Domain analysis workshop report for the automated prompt and response system domain. Special report. CMU/SEI-96-SR-001. May 1996.

Leung, H.K.L. & White, L. A study of regression testing. In Proceedings of the 6th International Conference on Testing Computer Software. USPDI. 1989.

Leung, H.K.L. & White, L. Insights into testing and regression testing global variables. Journal of Software Maintenance. No 2 December 1990.

Lewis, R., Beck, D.W. & Hartman, J. Assay - a tool to support regression testing. British Telecom Research Lab/Dept. of Computer Science, University of Durham. 1988.

Little, L.A. A new approach to managing project requirements and system testing. In proceedings of the Fourteenth Annual Pacific Northwest Software Quality Conference. 1996.

Littlewood, B. & Wright, D. A Bayesian model that combines disparate evidence for the quantitative assessment of system dependability, In: Mathematics of Dependable Systems, II, edited by V Stavridou, Oxford: Clarendon Press, 1997 pp. 243-258.

Liu, L., Robson, D.J. & Ellis, R. A data management system for regression testing. In proceedings of the 1st International Conference on Software Quality Management. March 1993.

Nilson, R., Kotgut, P. & Jackelen, G. Component provider's and tool developer's handbook central archive for reusable defense software (CARDS). (STARS-VC-B017/001/00). Reston, VA: Unisys Corporation. 1994.

Ovum. An evaluation of software testing tools. Ovum 1998.

Rothermel, G. & Harrold, M.J. A safe, efficient regression test selection technique. In ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, April 1997.

Rothermel, G. & Harrold, M.J. Analyzing regression test selection techniques. In IEEE Transactions on Software Engineering, Vol. 22, No. 8, August 1996.

Salmela, M., Korhonen, J. & Kalaoja, J. Support system for feature-based testing. A KATA-project report. To be published in 1999.

von Mayrhauser, A. & Olender, K. Efficient testing of software modifications. In Proceedings of the International Test Conference. IEEE. 1993.

Published by

Vuorimiehentie 5, P.O.Box 2000, FIN-02044 VTT, Finland
Phone internat. +358 9 4561
Fax +358 9 456 4374

Series title, number and report code of publication

VTT Publications 406
VTT–PUBS–406

Author(s)

Korhonen, Jukka, Salmela, Mika & Kalaoja, Jarmo

Title

The reuse of tests for configured software products

Abstract

New efficient software production techniques are important for improving the time-to-market of software products. One example of such advanced techniques is the so-called feature-based software production which employs high-level requirements or features in finding and selecting reusable software components for the development of new products. This kind of model-driven software development shortens the production time, but the validation of configured products still remains a bottleneck.

An effort to apply regression testing techniques to configured software products shows that these techniques are not very well suited to meeting the new testing challenges. It is obvious that retesting an entire program, containing possibly only a few minor changes, is expensive. Therefore, an efficient testing approach is required for optimizing the size of the test suite. Other important issues concerning the testing approach are the design of reusable tests, the configuration and management of tests, and the automation of test execution.

In the research, the testing efficiency problem is solved by using the idea of reusable software components from the feature-based production. In software testing the idea converts into a set of reusable test components designed for a product family. From a test material repository a suitable subset of tests is selected, modified, and configured to cover the characteristics of the product being tested. In addition, the repository may contain other relevant test data to be used for configuration, such as the test plans and the test environment configuration. The technique is called the feature-based testing approach.

For identifying the relevant test data, the method proposes links to be created between the product features and the test material. The result of the test configuration depends on the automation degree, varying from a simple test involving identification of lists useful in manual testing to executable tests in a fully automated test environment.

Often in structured testing the test case assumes that the software is in a specific state. Therefore joining test scripts arbitrarily may not produce the desired results. For that reason, we propose utilization of test specification components that are capable of using product feature data, taking care of the execution order and selecting appropriate tests for the product.

When implementing the feature-based testing approach, the issues to be emphasized are script development, test development and execution, and test management. A support system implementing the main characteristics of the feature-based testing approach has been outlined in the report. The tool is demonstrated in a case study.

Keywords

software testing, feature-based software, regression testing, configured systems

Activity unit

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN-90571 OULU, Finland

ISBN	Project number
951–38–5556–2 (soft back ed.) 951–38–5557–0 (URL: http://www.inf.vtt.fi/pdf/)	E8SU00158

Date	Language	Pages	Price
January 2000	English	67 p.	B

Name of project	Commissioned by
KATA	Nokia Display Systems Oyj, Semel Oyj, Polar Electro Oyj, Suunto Oyj, Vaisala Oyj

Series title and ISSN	Sold by
VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)	VTT Information Service P.O.Box 2000, FIN-02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374