



Information Technology for European Advancement

Guideline for *Validation & Verification* Real-Time Embedded Software Systems

D 1.6.2

V 01

Edited by Jens Herrmann

Software Development Process for Real-Time Embedded Software Systems (DESS)

ITEA COMPETENCES involved:

- 1) Complex Systems Engineering**
- 2) Communications**
- 3) Distributed Information and Services**

December 2001

Contents

Purpose and context of this document..... 3

1 Introduction..... 4

2 Validation, Verification, and Testing Real-time Embedded Systems 6

3 Validation & Verification 14

4 Testing..... 21

 4.1 Test Management, Planning, and Control 21

 4.2 Test Workflows 27

 4.2.1 Component Testing..... 27

 4.2.2 Integration Testing 28

 4.2.3 System Testing 31

 4.2.4 Acceptance Testing 32

 4.2.5 Regression Testing 33

 4.3 Test Activities 34

 4.3.1 From *Test Object Analysis* to *Test Evaluation*..... 35

 4.3.2 Test Case Design 40

 4.3.2.1 Normal-range, boundary, and robustness testing 40

 4.3.2.2 Equivalence Partitioning 42

 4.3.2.3 Classification-tree method..... 42

 4.3.2.4 Structure-oriented testing 45

 4.3.2.5 Procedure-free tests 50

 4.4 Test Process Improvement..... 50

5 Conclusion..... 54

Glossary..... 55

References..... 58

Appendix A: Release procedure 61

Appendix B: Embedded Software Quality Criteria 64

Purpose and context of this document

The aim of this document is to support validation and verification (V&V) of embedded real-time software-systems. The main focus thereby is testing of these systems, since testing is of great importance for the software development practice. The most important aspects of a V&V process for embedded software-systems are described in this document.

Context for this guideline is the ITEA-DESS-project (www.dess-itea.org). The goal of DESS is to define an innovative object-oriented, component-based software development methodology for embedded real-time systems. An important part of the development method defined in DESS deals with the procedure of validation & verification embedded systems. The guideline described in this document is the result of a collaborated work of DESS partners, who worked together on this subject.

1 Introduction

Welcome to this guideline for validation & verification real-time embedded systems.

From refrigerators to electric shavers, ever-greater numbers of consumer goods are being computerised to provide additional benefits to the user. In products, such as cars, mobile phones and TV-sets, this increased functionality increases the complexity of the product. The software that powers these features is embedded and has to react in real-time to events from outside (e.g. activating a car air bag in case of an accident).

An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function as part of a larger system or product. Embedded software is software designed to perform a dedicated function as part of a larger (software) system. A real-time system is able to respond to an external event within a given time.

Embedded systems form the by far the largest part of presently used computer systems. The class of embedded systems includes both, systems which are similar to the present PC, and smallest systems with a minimal memory and calculator capacity. The development of software systems for such platforms makes by far higher demands than the development of general purpose systems for the use on PC and workstations. However, in this very area, progressive development techniques such as object-oriented design and implementation are only slowly gaining ground. Much more common is to work with Assembler and C. The resulting deficits mainly affect the area of re-usability of existing solutions, and the verification and validation of embedded systems, respectively.

The goal of DESS is to define an innovative object-oriented, component-based software development methodology for embedded real-time systems, to create supporting environments by integrating state-of-the-art tools and to prove the appropriateness of the methodology by implementing several validation test cases. DESS is the acronym for *Software Development Process for Real-Time Embedded Software Systems* (see www.dess-itea.org). The methodology that is being developed in the ITEA DESS project will help to engineer high quality software at reasonable costs within the time targets set.

One important part of the development method defined in DESS deals with the procedure of **validation & verification** (V&V) embedded systems. V&V represents the totality of techniques for analysing and assessing systems. The aim of this particular part of DESS work is to provide a guideline for V&V. An important basis for the guideline development is the current state-of-the-art testing at the DESS partners which is documented in (*DESS 00*).

The DESS method describes the procedure with object-oriented, component-based development of embedded real-time systems (*DESS 01-2*). As basis serves the V-model (*V-model 97*) which was adjusted and extended within DESS to the particular properties of the systems under consideration in DESS (embedding, real time, components, and object orientation). More precisely, the DESS method consists of three proceeding mod-

ITEA CONFIDENTIAL

els, or three Vs: a *Realisation-V*, a *Validation-V*, and a *Requirements-V*. The *Realisation-V* describes the creation of embedded real-time systems, and has phases such as requirements definition, design, implementation, and integration. The *Validation-V* describes the analysing activities which can be applied parallel to construction, such as requirements review, model checking, component testing, integration testing, system and acceptance testing, respectively. The *Requirements-V* describes requirements management aspects to be applied during the development.

This document describes the guideline for validation & verification component-based, real-time embedded software-systems. This guide is the result of a collaborated work of DESS partners, who worked together on this subject. The document is structured as follows:

- **Chapter 2** introduces the topic of *analysing component-based, embedded real-time systems*. This chapter provides an overview on the topic, and explains, among other things, the basic terms and concepts in use, e.g. validation, verification, and testing.
- **Chapter 3** explains the terms *validation and verification* with regard to the process model as developed in DESS for the development of embedded real-time systems. The V&V techniques which may be employed parallel to system construction, e.g. review, model checking, and testing, will be explained in detail.
- **Chapter 4** concentrates on the *testing* of embedded real-time systems. Since testing is of great importance within validation and verification, we allow testing to take a relatively large part in this guideline. The model for testing includes four aspects, each being explained in separate chapters: *test management*, *test workflows*, *test activities*, and *test process improvement*.

The document concludes with a summary, assessing the results as being described in this paper, and gives an outlook on possible subsequent works.

2 Validation, Verification, and Testing Real-time Embedded Systems

The amount of embedded software in technical systems is steadily increasing. Even with the present state of the art, it is not possible to guarantee error-freedom of this software. Still, quality management is confronted with the task to ensure, by means of constructive and analytical methods, the development of a high-quality product with a lowest possible amount of errors.

Together with the increasing complexity of software-based systems, the quality management requirements will increase likewise. The term *quality* stands for a number of characteristics, such as functionality, safety, reliability, real-time ability, usability, and re-usability. These properties are usually divided into more concrete ones until they result in quantifiable properties. This way, so-called quality models develop which are noted in the form of trees. Not all quality properties may reach quantitative measurability. Thus, for instance, the perception of the property *usability* depends on subjective notions and the respective experience of the users.

Constructive approaches aim at organising the development process of software with the help of organisational measures and the use of suited constructional techniques in such a way that the development of quality defects or errors is minimised from the beginning. Examples of constructive measures are the deployment of established process models such as the V-model (*V-Model 97*) and specific techniques such as component deployment and precise interface specifications as developed in DESS. According to experience, the constructive approaches are not sufficient to achieve the desired quality of a product. Despite their deployment emerging defects cannot be precluded.

Analytical measures aim at showing the accordance of the developed software with its requirements and at detecting existent errors. In order to guarantee this development-concomitant tests are carried out. Analytical measures can be roughly subdivided into two classes: *validation* and *verification*. Validation responds very much to the wishes of customers or users and operates in respect to these requirements. In the case of verification developers examine whether they work correctly during construction (more on V&V in chapter 3).

The analytical technique used most often in practice is the test which permits a systematic search for errors and in which the correct behaviour can be proven in certain cases. Validation as well as verification can be carried out by testing. With the test, a random sample will be selected from the input domain of the test object which is then executed with these chosen input values. After that, the results obtained by this execution are compared with the expected values. Thus, testing is as a dynamic technique, i.e. a technique which contains the execution of the test object. Testing is a very important analytical method as it permits the analysis under real-world operating conditions.

Beyond testing there are many more analytical methods, such as static analyses, symbolic interpretation, model checking, and formal proof. They can complement the de-

ITEA CONFIDENTIAL

ployment of the test and that can be deployed where the dynamic test cannot be used yet: in the early development phases. Since the test is a very important practical analysis technique for the DESS partners it will be discussed in this guideline in detail (chapter 4). Other V&V activities will also be addressed briefly (chapter 3).

In general, V&V activities should start early within the development process. Already in the requirements definition phase, the requirements for the validation and verification of the system should be taken into account. This implies an early involvement of a V&V team in the development process. The task of V&V in the requirements definition phase is to specify the requirements for verification and validation of the system during system development, production and maintenance.

One important reason for the early deployment of V&V is the experience that the expenses of eliminating errors which are detected at an early stage are generally lower than the expenses of eliminating errors which are detected at a later development phase. A change of the specification, for instance, is more economical than a change of the end product. Further advantages of an early start of V&V activities are that an early support of the improvement of the unambiguity and completeness of specifications can be achieved thus, and that the testing will be more thorough in general.

The use of an independent, qualified V&V team is often recommended as a means to ensure the development of high-quality (embedded) software (*Zelkowitz and Rus 01*). Independent means that the V&V team is not directly involved in the development process, but accompanies the development and (at least) reviews, verifies, and validates the products or artefacts of the software development. The classical form of independence includes technical, managerial, and financial independence (*IEEE 98*). In practice, forms of independence are often found which are less stringent and which compromise. For instance, in the form of technical and financial independence, but including a common management which controls both, software development and V&V activities.

The expenses for V&V activities will depend on the quality requirements and the criticality of the embedded systems, i.e. on the degree of the effect a faulty functionality could provoke. For instance, errors in the avionic software of an aeroplane may cause more damage (and endanger human lives) than a faulty software in a washing machine or in a vacuum cleaner.

Generally speaking, testing is concerned with discovering whether certain properties (qualities) exist. This is achieved by determining and evaluating the actually existing properties. Testing in the case of software-based systems means, for example, to test the user-friendliness, functional correctness, safety, reusability, temporal and/or memory behaviour of the system. In this context, we use the following definition of the test aim:

The aim of testing is to gain confidence that a test object possesses the required properties.

There are more ways than one to reach this goal. For instance, the test can attempt to

ITEA CONFIDENTIAL

prove the consistency (correctness) of the test object and its requirements. Since testing is of an experimental nature this can only happen in certain cases. On the other hand, the test objective can also be error detection. Theoretical deliberation and practical experience have shown that the indirect testing approach, i.e. error detection, is a very effective approach to quality assurance.

A very common procedure for testing comprises the following main technical activities (cf. Figure 1):

- test case determination,
- test data selection,
- expected results prediction,
- test execution,
- monitoring, and
- test evaluation.

This structure facilitates a systematic procedure and the definition of intermittent results.

In the course of **test case determination**, the test cases, with which the test object should be tested, are defined. A test case defines a certain input situation to be tested¹. It comprises a set of input data from the test object's input domain. Each element of the set should, for example, lead to the execution of the same program functionality, the same program branch or the same program state, depending on the test criteria applied. The test case determination is the most important activity for a thorough test, since it determines the kind and scope of the examination and thus the quality of the test. A test case abstracts from a concrete test datum and defines it, only in so far as it is required for the intended test. During **test data selection** the tester has to choose a concrete element from each test case with which the test should be executed.

Determining the anticipated results and program behaviour for every selected test datum constitutes **expected results prediction**. If it is not possible to specify unequivocal output values or the expected behaviour, acceptance criteria or other reference data could be used for the prediction of expected results. Subsequently, **test execution** is performed. The test object is run with the selected test data. The output values and the program behaviour are thus determined.

¹ Note that this is a restricted definition of a test case since it is focused on the input domain of the test object. Other definitions also include, beyond the input, the expected output into the notion of the test case (cp. *IEEE 90, DO-178B, BS 7925*)

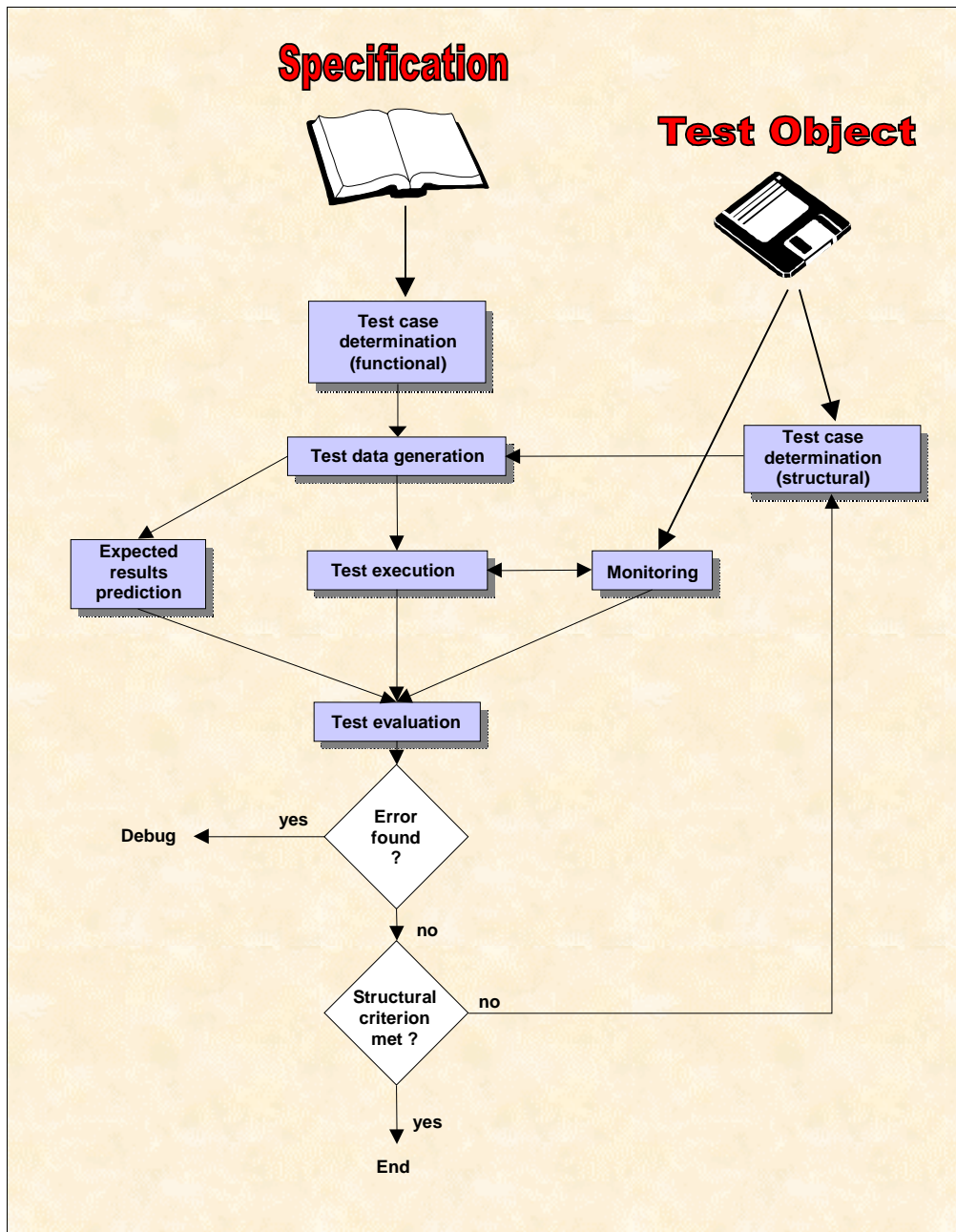


Figure 1: Test procedure

The behaviour of the test object can be observed and recorded during test execution by means of **monitoring**. A common method is to instrument the program code according to a white-box test criterion. For that purpose, the source code is extended by inserting statements at control-flow or data-flow-relevant points of the program, which count the number of executions of the corresponding program parts. Another kind of monitoring is performed by several capture-and-replay tools. They record the outputs produced by the test object on the screen and save them for regression testing.

ITEA CONFIDENTIAL

In the course of **test evaluation**, actual and expected values as well as actual and expected program behaviour are compared, and thus the test results are produced. Finally, the test is evaluated by comparing the test results achieved with the test objectives aspired to.

Testing should always be oriented towards customer satisfaction. The customer demands that the system functions correctly according to his requirements, in this way decisively influencing what is meant by quality in this context. The requirements defined by the customer thus determine the testing procedure.

A very effective, indirect approach to test case design which combines both testing for functional correctness and error detection is the following: Test cases are designed on the basis of the requirements with the aim to detect errors. An error occurs if the behaviour of the test object deviates from the requirements. If the procedure detects an error we have the basis for quality improvement. If no errors are detected the correctness of the specific test case is proven. This testing approach is called *falsification of requirements*.

Complex software systems usually require complex tests, i.e. testing is concerned with a huge number of tests. The most important task when testing large systems is to cope with the complexity of the test.

Testing large systems usually requires a different procedure than testing smaller systems. One possibility to cope with the complexity of the test is to split it into several manageable smaller tests. It is also important to determine on a strategy which ensures that the sum of all single tests will result in a thorough test of the overall system.

A general testing approach of complex component-based systems oriented towards their internal architecture starts with testing the smallest system components. These are first of all tested in isolation. After completing the component tests, the components are integrated into a higher level for further testing. In this higher level only those test aspects which have so far not been taken into consideration are applied, e.g. interfaces between components. The stepwise composition of the tests where the system environment (e.g. hardware) is more and more included leads to larger and larger tested system parts, until the system has been completely integrated and tested.

For real-time systems (or systems with temporal requirements), correct system functionality depends on logical as well as on temporal correctness. Static analysis alone is not sufficient to verify the temporal behaviour of real-time systems. In the following an approach for testing systems with temporal requirements will be described. This approach is based on evolutionary algorithms.

In the course of system design, schedulability analyses are employed to construct systems in a way that these meet the temporal requirements. Techniques of static analysis are used to assess the execution times of tasks as pre-condition for the schedulability analysis. To date, no commercial tools have been developed for the static analysis of a

ITEA CONFIDENTIAL

program's temporal behaviour. Therefore, in practice execution times often have to be estimated by a complex and error-sensitive manual inspection of the code (*Puschner and Vrchoťický 97*). Even if professional tools for the static computation of execution times were to be available in the future, errors can not be ruled out of time estimations, e.g. caused by:

- errors in the implementation of static analysis tools,
- incomplete information on the dynamic behaviour of the target processor, or
- false information given by the developers on the program's dynamic features, e.g. maximum number of iterations of program loops.

Therefore, as long as no formal verification methods exist, systematic testing is an inevitable part of the verification and validation process for real-time systems. A very promising approach for testing temporal behaviour is evolutionary testing. In various experiments evolutionary testing has been compared to static analysis (*Mueller and Wegener 98*), random testing (*Wegener and Grochtmann 98*), and systematic testing (*Wegener et al. 99*) and achieved good results.

Evolutionary operators used for evolutionary tests have been constantly improved. The use of extended evolutionary algorithms overcomes the problems reported in using genetic algorithms. This establishes the foundation of the first industrial application of the evolutionary test for the dynamic test of a motor control system.

One major objective of testing is error detection. The temporal behaviour of real-time systems is defective when input situations exist in such a manner that their computation violates the specified timing constraints. In most cases the task of the tester therefore is to find those input situations with the shortest or longest execution times to check whether they produce a temporal error. Within evolutionary testing the search for the shortest and longest execution times is regarded as an optimization problem to which evolutionary algorithms are applied. With the exception of very simple real-time systems, temporal behaviour always forms a very complex multi-dimensional search space with many plateaus and discontinuities. Due to the complexity of the temporal behaviour, it is not astonishing that some researchers in control theory have drawn the conclusion that computer systems are inherently probabilistic in terms of their timing behaviour (*Törn-gren 98*). Correspondingly difficult is the testing of temporal behaviour with conventional black-box and white-box test methods. The evolutionary test therefore grounds on a stochastic procedure: evolutionary algorithms.

Evolutionary algorithms represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's theory of evolution. They are characterized by an iterative procedure and work in parallel with a number of potential solutions, the population of individuals. In every individual, permissible solution values for the variables of the optimization problem are coded. The evolutionary search and optimization process is based on the three fundamental principles selection, recombination

and mutation.

When using evolutionary algorithms for determining the shortest and longest execution times of test objects, each individual of the population represents a test datum with which the system under test is executed. For every test datum, the execution time is measured to determine the fitness value of each individual.

The evolutionary process repeats itself until a given stopping condition is reached, e.g. a certain number of generations, or when an execution time is found which is outside the specified timing bounds of the system under test. In this case, a temporal error is detected. If, however, all the times found meet the timing constraints, confidence in the temporal correctness of the system is substantiated.

The approach described here applies the extended evolutionary testing (*Wegener and Grochtmann 98*) which allows the combination of multiple strategies, e.g. global and local searches and the automatic distribution of resources in accordance with the success of the strategies. For a detailed discussion of evolutionary algorithms, see *Mitchell 96*. Figure 2 shows the structure of the evolutionary process.

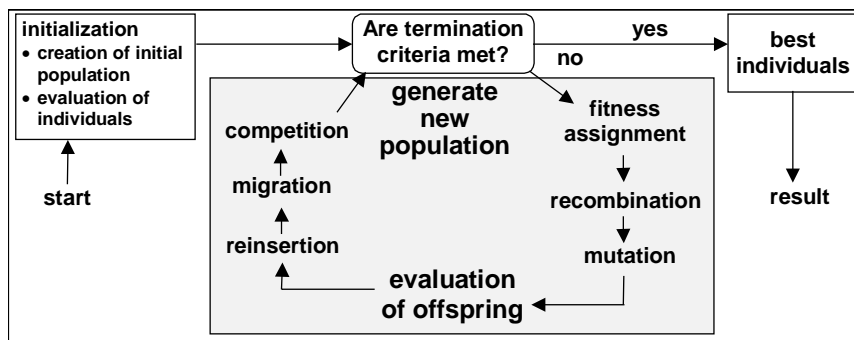


Figure 2: Structure of the extended evolutionary algorithm

An application field for the evolutionary testing method is a new motor control system for six- and eight-cylinder blocks that is currently under development. It contains several tasks that have to fulfil timing constraints. The execution times for these tasks are determined by the use of hardware timers in the target environment. They have a resolution of 400 ns. A hardware timer is questioned directly before and after the execution of the tested task to determine its execution time.

Evolutionary testing (ET) is used to verify the results from the developers' tests (DT) which are based on the functional specification of the system as well as on the internal structures of the tasks. The longest execution times determined by the evolutionary testing are compared to the maximum execution times found by the developers' tests. The tests are performed on the target processor later used in the vehicles. The test results are shown in Table 1.

It can be clearly seen that evolutionary testing has found longer execution times for all

ITEA CONFIDENTIAL

the given tasks. This is surprising for evolutionary testing treats the software as black boxes, whereas the developers are familiar with the function and structure of their system. An explanation might be the use of system calls, compilation and optimization, as well as dependencies on other system components, of which the temporal effects can only be rated with difficulty by the developers. With regard to the motor control system execution times determined by evolutionary testing do not exceed the temporal constraints of a task in any of the cases. The irregularities of the test results therefore have not been disquieting. The intensive testing of the temporal behaviour with systematic and evolutionary tests has strengthened the developers' confidence in a correct temporal behaviour of the system.

Task name	Max. execution time in μs		Lines of code	Input parameter
	ET	DT		
Task zr2	69,6 μs	67,2 μs	41	18
Task t1	120,8 μs	108,4 μs	119	18
Task mc1	112,0 μs	108,4 μs	98	17
Task mr1	68,8 μs	64,0 μs	81	32
Task k1	59,6 μs	57,6 μs	39	14
Task zk1	58,4 μs	54,0 μs	56	9

Table 1: Maximum execution times of motor control tasks

For all the tasks of the motor control system, evolutionary testing achieved better results in comparison with the developers' tests. This illustrates how difficult it is to thoroughly test the temporal behaviour of systems by using systematic testing. However, since no search strategy can guarantee that extreme execution times will be found, the use of evolutionary testing alone is not sufficient for a thorough and comprehensive examination of temporal behaviour. A test strategy for real-time systems should at least include systematic testing and evolutionary testing. For more information about testing embedded software systems with temporal requirements please look at www.systematic-testing.com.

3 Validation & Verification

Validation and verification (V&V) activities can and should be executed in parallel to all occurring system and software development phases. As always, it is understood that the earlier an error is detected the lower the cost of its elimination will be. In DESS, the V-Model (*V-Model 97*) is taken as basis for the development of a software process model. The DESS process model provides development phases like Requirements Definition, Design, Coding, and Integration (Figure 3, *DESS 01-2*). V&V activities can be carried out correspondingly in parallel to all these phases, e.g. Requirements-Review, Model-Checking, Component-, Integration-, System- and Acceptance Testing (Figure 4) (see also *DO-178B*).

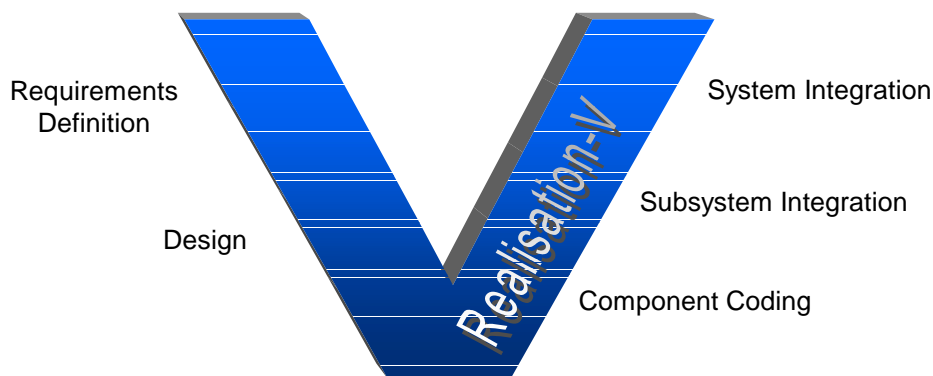


Figure 3: Development activities (*Realisation-V*)



Figure 4: V&V activities (*Validation-V*)

A typical, minimal V&V-model intends to check at least the artifacts that result from the different development phases (Figure 5). The artifacts which emerge first during the development constitute the requirements for those that arise at a later stage during the development. Thus, the Component Design as well as the implemented subsystems, for example, need to meet the requirements of the Architectural Design.

In addition to the activities mentioned above, V&V can also take on other tasks, like, for

example, a risk analysis on the basis of requirements or process quality evaluations. The effort invested in generating V&V is, as previously mentioned, dependent on the project quality goals and the software criticality.

Within the scope of DESS, the proceeding models for the software construction, and validation/verification as presented in the figures 3 and 4 are distinguished in *Realisation-V*, and *Validation-V*. Figure 5 is a detailed version of Figure 4. At a close look, the arrows presented in Figure 5 are V&V processes. As inputs, they have the object to be checked, and the requirements on the object. As outputs, they produce analysis results (cp. Figure 6). As previously said, the requirements on the object under analysis are artifacts which have evolved at an earlier development stage. A detailed illustration of the V&V process is shown in Figure 7. The process consists of the following phases:

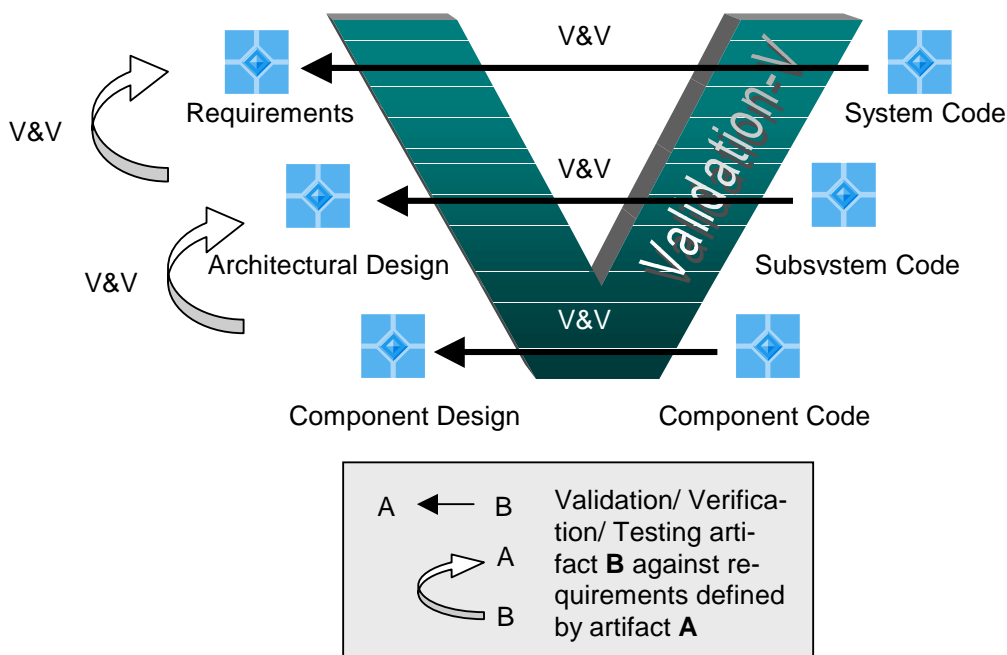


Figure 5: Artifact-oriented V&V (*Validation-V*)

- **Planning & Control:** Managing, planning, and tracking the V&V process (cp. chapter 4.1). The main results of this activity are plans and standards describing strategies and procedures to perform and control the V&V process.
- **Preparation:** Description of a detailed analysis strategy. Among other things, it will be defined which aims are to be achieved by this the analysis, and which V&V techniques should be used.
- **Specification:** Exact specification of the analysis, for instance by describing the environment and the analysis cases.
- **Implementation:** Technical realisation of the analysis by providing an executable analysis environment and executable inputs for the object to be analysed.

- **Execution:** Analysis execution, possibly including a coverage analysis for the object to be analysed (cp. chapter 4.3.1, section *coverage analysis*).
- **Wrap-up:** Evaluation of the analysis results, if applicable, release advice for the analysed object.

In principle, this procedure may be carried out for all sorts of V&V, i.e. for early analysis activities such as reviews up to tests. Activities such as implementation and execution thereby have varying meanings.

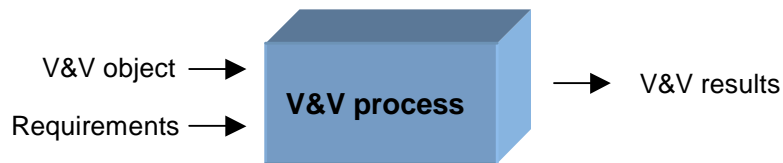


Figure 6: V&V process

V&V activities in the earlier development phases are of a rather static character whereas in the later development phases they are more dynamic. Reviews are, for example, static procedures, and testing is the most important dynamic V&V-procedure. The V&V of requirements is of a rather static character. The V&V of the design can, among other things, be supported by model checking. If code is available dynamic component, integration, system, and acceptance tests can be carried out.

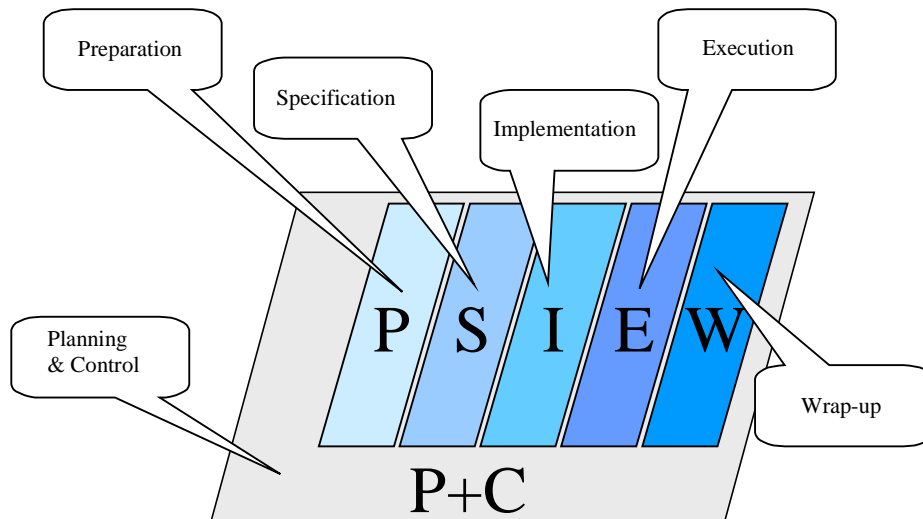


Figure 7: V&V process in detail

ITEA CONFIDENTIAL

Validation and verification are distinguished as follows: (cp. *IEEE 90*):

Validation: “Do we build the right product?”. The developers (and the customer) examine if the development results in the product the customer desires. The primary focus of validation is customer satisfaction.

Validation checks a problem solving against the requirements as stated by the customer. The aim is to prove that the solution is adequate regarding the customer's requirements. Validation requires to consider real application conditions of the object under analysis. Thus, validation includes that the contractors, the developers, check whether they are developing exactly the product which was ordered by the customer. Validation is mostly carried out by testing. The customer should participate at validation, at least at acceptance testing.

Verification: “Do we build the product right?”. The developers check whether they are working properly during the development.

Verification is intended to check one development result against a previous one. For instance, an implemented, embedded subsystem against an architecture description, which was developed during design. Another example is checking a model, having been developed during design, against requirements of an earlier development phase. In case a development step is being automated and in case this transformation has been proven accurate, verification may be simplified or even be omitted.

Customer participation is mandatory for validation. Concerning verification, his participation will be certainly a useful support. Basically: the more the customer and his requirements are involved in the analysis, the sooner the aimed quality can be achieved (catchword: *customer satisfaction based V&V*).

A lot of techniques are available for validation and verification, e.g.

- static analysis,
- simulation,
- model checking,
- formal correctness proofs,
- symbolic execution,
- review (inspection, walkthrough),
- testing,

and many more. For a general survey about techniques see for example *Kaner et al. 93* and *Binder 99*.

Model checking is a technique for analysing models. Models are abstractions (prototypes) of the system to be created. They evolve at an early development stage, fre-

quently during design. Software models are usually graph-shaped, and they may be interpreted, simulated, and be executed symbolically. Examples are *finite state charts*, *message sequence diagrams*, and *Simulink block diagrams* (for *Simulink* look at www.mathworks.com).²

Model checking tests models against previously defined requirements or properties. Of particular interest for the DESS context is to test whether temporal requirements are be met by the model. Other forms of tests are possible, e.g. whether deadlocks or unreachable model parts are existing. For analysing purposes, the model is being traversed, and it is being checked whether it meets the required properties or not. An approach for the special topic *model testing* is described in *Conrad et al. 99*.

Besides, models may also be used in order to test the final implemented software product. In this case, the model serves as test basis, i.e. test cases are being derived from the model to test the final product.

Static analysis means checking without running the V&V object (the object under analysis). These checks are text analyses, with the syntactic correctness of the texts taken for granted.

An effective form of static analysis with a strong reference to the functionality of the V&V object is provided by the **review**. The review is carried out in a team, which should include both, people having participated in the development of the V&V object and people having not. We can distinguish two kinds of reviews: walkthrough and inspection:

- **Walkthrough:** The review team simulates the execution of a V&V object by means of its source text. Test data are being created with which the team can work through the program step by step. This allows to see how the V&V object works internally. The walkthrough is suited, among other things, to reveal awkward algorithmic solutions and errors in algorithms.
- **Inspection:** The team checks the V&V object by means of its source text, e.g. line by line against a check list. A particular form of inspection is the **technical review**. Here, the team members first prepare themselves separately and then introduce a number of discussion points in the subsequent inspection. Apart from the aim to prove conformity with functional requirements, an inspection can also aim at, for instance, proving the conformity of the V&V object with regard to standards.

Static techniques are able to uncover missing, deficient, redundant, or unrequired functionality in the source text of the object under analysis. A big advantage of static analyses is their use in early development phases. The object to be analysed does not neces-

² The definition of model checking given in this paper is a definition in the broader sense. Different kinds of models are included. In many publications, however, model checking is used in a narrower sense for checking *finite state charts*

sarily have to be executable. Thus, static analyses can also be applied to requirements and design.

Testing means to check dynamically whether requirements are fulfilled. Testing includes the execution of a test object with test data and the subsequent evaluation of test run results. Both, validation and verification can be carried out by testing. As previously said validation predominantly occurs by testing. Compared to other V&V techniques, testing has the big advantage of analyzing the system in *realistic application conditions*. This allows, for instance, to realistically examine real-time behaviour and the interplay with the hardware.

Practical experiences have shown that in particular the combined use of different V&V methods results in an effective error uncovering. Dynamic and static analyses complement each other, and thus form a comprehensive and efficient analysis of a test object. It depends on the kind of the system and its criticality in which scale, which order and which combination V&V techniques should be used. In case of embedded real-time systems, the test is of utmost importance.

Figure 8 illustrates the procedure model being developed in DESS for the development of component-based, embedded real time systems. It is a modified V-model being attuned to the DESS requirements (the Realisation-V). The DESS process model is explained in detail in *DESS 01-2*.

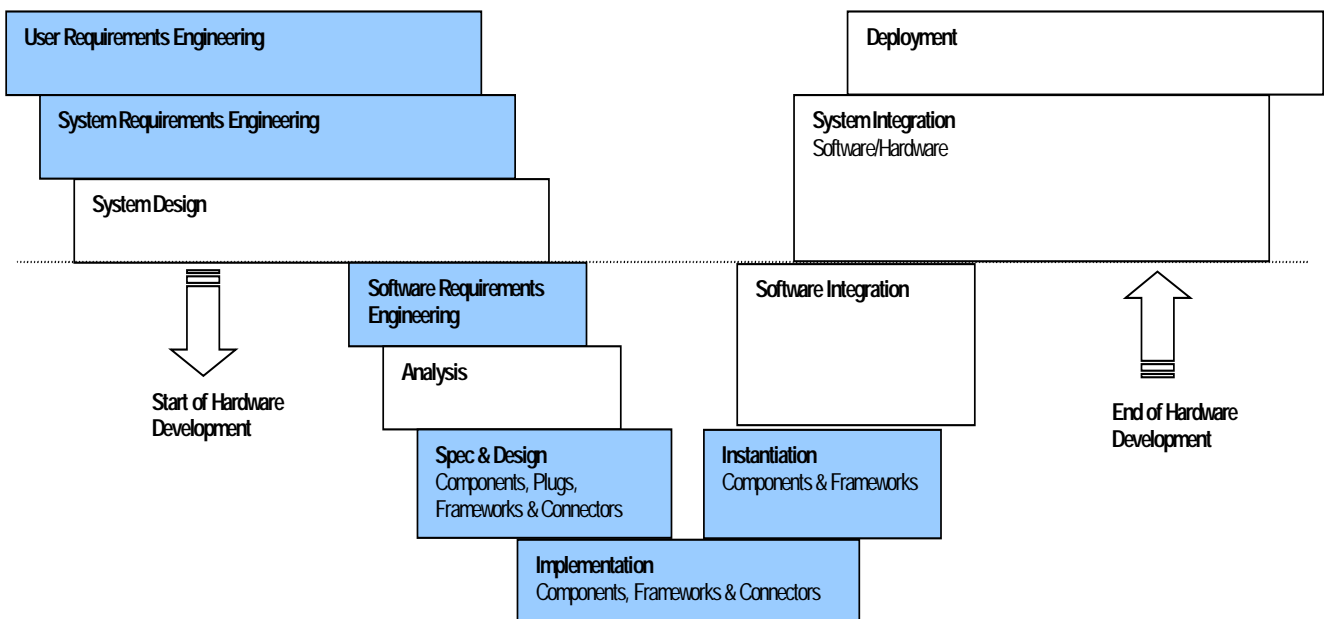


Figure 8: DESS Realisation-V

Figure 9 shows the V&V activities which can be applied within the DESS process model, in accordance with the DESS procedure model. Reviews could be well applied during the early development stages (and also in the later stages), model checking during design

workflows, and testing as soon as code is produced.

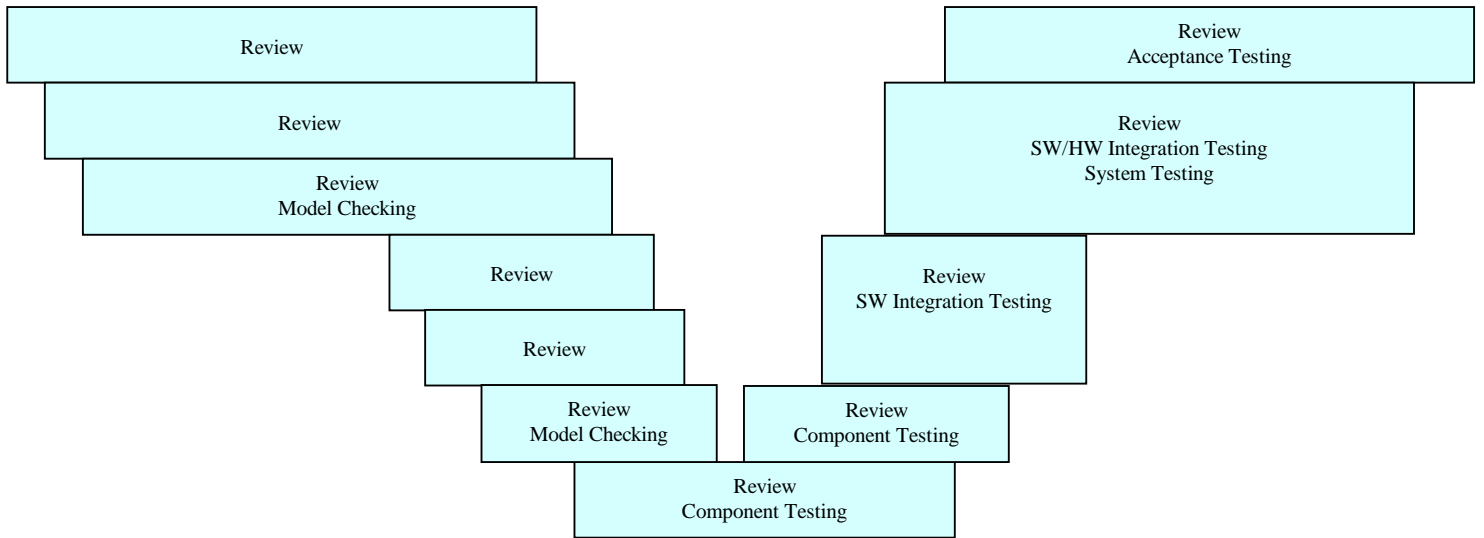


Figure 9: DESS Validation-V

4 Testing

The model for testing embedded real-time systems comprises four parts (Figure 10):

- **Test management:** includes management, planning, and control of the test process.
- **Test workflows:** the single workflows during testing: component test, integration test, system test, acceptance test, and regression test.
- **Test activities:** activities occurring within the workflows.
- **Test process improvement:** deals with planning and continuous improvement of the test process.

Each of the four parts of the test model will be considered in detail in the following sub-chapters.

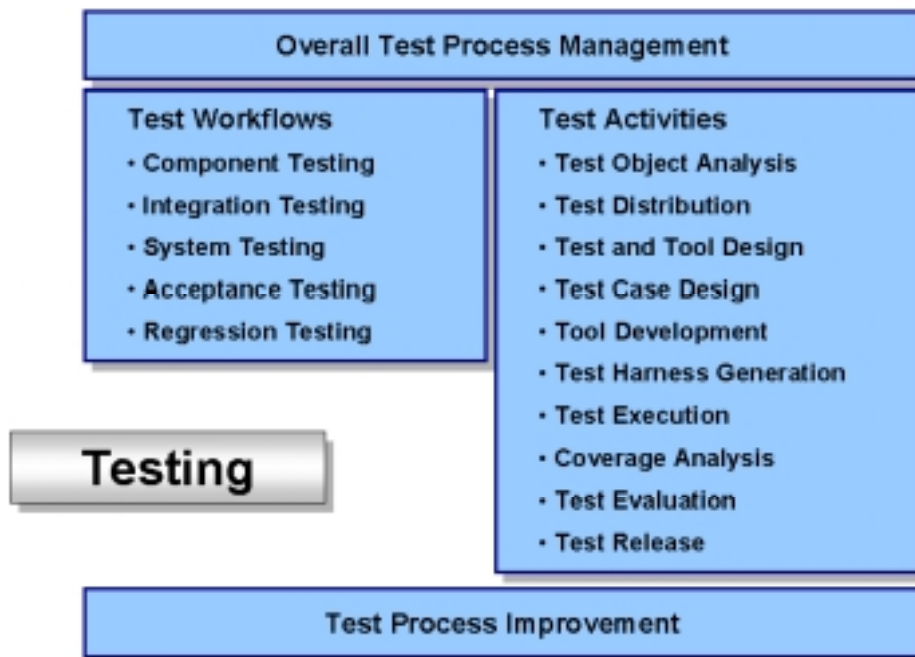


Figure 10: Test process model

4.1 Test Management, Planning, and Control

At the outset of the complete testing process and, if necessary, also during the testing process (e.g. before the start of the single test workflows (chapter 4.2)) each individual testing process is planned. Results of the planning are concepts (documents like the test master plan and test workflow plans) and, where applicable, standards to regulate the testing process. The planning needs to be coordinated with the customer of the system

ITEA CONFIDENTIAL

to be generated. If the system is one that requires the approval of a licensing authority (this refers to e.g. embedded avionic software) then the licensing authority is to be involved in the planning.

Planning includes the following points:

- Testing goals (incl. test process improvement goals (see chapter 0)), quality goals which should be reached.
- Manpower needed, estimated budget.
- Responsibilities, roles.
- The degree of independence of the testers.
- Technical infrastructure, required resources.
- Test end criteria, test release procedure (e.g. coverage criterion is to be met or all defined tests are to be passed without error).
- Test strategy.
- Test work breakdown, milestones.
- Configuration management plan (e.g. for the management of test objects, test specifications, test data, test results).
- Traceability.

During execution of test activities, test management activities are the following:

- Maintain the project plan.
- Track progress plus progress reporting of testteam (planning, budget).
- Maintain the interface between development- and testteam.
- Track progress on implementing improvement goals.

A test strategy determines how the testing is going to be executed. It determines on which quality aspects and on which test objects (e.g. sub systems) the test concentrates in order to use the resources and the available time in the most efficient way.

The goal is to organise the testing activities in such a way that:

- the most important problems are found.
- the problems are found as early as possible.
- the problems that are most difficult to fix, are found first (e.g. memory leakage).
- at the end a quality advice can be given.

The strategy should be based on a risk evaluation upon both the different test objects and the quality criteria. The higher the possible damage potential of system parts the greater the planning effort for testing these system parts should be. Examples for critical-

ITEA CONFIDENTIAL

ity classes can be found in many standards; the DO-178B names the following five: *Catastrophic, Hazardous, Major, Minor, and No Effect* (cp. *DO-178B*). On the basis of this risk evaluation an adequate test strategy can be determined. If one component fulfils a very critical task this component is to be tested most thoroughly (hardest/most critical first).

The following steps can be defined for determining a test strategy:

1. Determine the required qualities for the release/system (see Appendix A: Release procedure).
2. Determine the quality criteria which have to be tested (see Appendix B: Embedded Software Quality Criteria).
3. Determine the relative importance of all criteria.
4. Assign the criteria to workflows.

Table 2 shows an example of the result of these steps.

V&V Workflows/ Qualities	Early V&V	Component Testing	Integration Testing	System Testing	Acceptance Testing	Relative importance
Continuity			+	+	+	10
Economy			++	+		5
Functionality		+	++	++	++	15
Performance			+	+	+	10
User friendliness				+	++	10
Usability				++	++	15
Connectivity						-
Flexibility	+					5
Maintainability	+					5
Manageability	+					5
Portability	+					5
Re-usability	+					5
Testability	+					10
						100%

Table 2: Relative importance of quality criteria per workflow (example)

Based on the overall test strategy (described in the test master plan), detailed test strategies have to be defined for each test workflow that is going to be performed. For each workflow, determine the

ITEA CONFIDENTIAL

- quality criteria for the workflow and their relative importance. This should be driven by the test master plan.
- test objects which have to be tested (components, subsystems, interfaces, ...) and their relative importance, e.g. by using a questionnaire.

The results of these steps could be summarised in a test priority matrix. An example is given in Table 3. Based on this matrix the subsequent test activities could be planned.

Test objects/ Quality criteria	Subsystem 1	Subsystem 2	Subsystem 3	Subsystem 4	Total System	Relative importance
Continuity		++			+	15
Economy	+	++	+		+	32
Functionality	+	++	+	+	++	36
Performance		++			+	17
	10	40	20	10	20	100%

Table 3: Example of a test priority matrix for a particular workflow

The next step of the test planning is to determine the test techniques to be used for testing the selected quality attributes in each workflow. Goal is to cover with the smallest possible set of test techniques all the quality attributes which have to be tested. Various test techniques are suited differently well to test certain quality properties. A survey showing which techniques can be deployed to test which qualities is very helpful in this case (see Table 8). The result of this planning step is the assignment of test techniques to test objects. Table 4 shows an example for this assignment.

Test object	Test technique	Tested quality attribute
Sub system 1	Test technique 1	Continuity, Performance
	Test technique 2	Performance
Sub system 2	Test technique 2	Performance
...		

Table 4: Assignments of test techniques to test objects

During a test process different people have different responsibilities or roles, respectively. A simple example for this are the roles of the test designer and the tester. The test designer plans and supervises the test, whereas the tester executes it. With larger projects the number of responsibilities increase. One person often holds more than one role at the same time. Responsibilities of the development teams and test teams, over-all project management and the customer could be as described by the example matrix in

ITEA CONFIDENTIAL

Table 5.

Some useful roles referring to Table 5 are:

- RA: Management role.
- EW: Executing role.
- AE: Advisory role.
- RE: Binding advice.

ITEA CONFIDENTIAL

Workflows	Artifacts	Customer	Project Manager	Development team	Test-team
Test planning	Master plan	AE	RA	AE	AEW
Product specifications	Customer requirements	AEW	RA	AE	AE
	Req. documents	-	RA	AEW	AE
	Design docs	-	RA	AEW	AE
	Code	-	RA	AEW	AE
White-box component testing	Test specification	-	-	RAEW	AE
	Test execution	-	-	RAEW	AE
	Test report	-	RA	AEW	AE
Black-box component testing	Test specification	-	-	AE	RAEW
	Test execution	-	-	AE	RAEW
	Test report	-	RA	AE	AEW
Integration testing	Test specification	-	-	AE	RAEW
	Test execution	-	-	AE	RAEW
	Test report	-	RA	AE	AEW
System testing	Test specification	-	-	AE	AEW
	Test execution	-	-	AE	RAEW
	Test report	-	RA	AE	AEW
Acceptance testing	Test specification	AE	-	AE	RAEW
	Test execution	-	-	AE	RAEW
	Test report	RA	RA	AE	RAEW

Whereas:

- R: Responsible for the execution of the activity according to the defined standards.
- A: Authority who can make decisions about the activity.
- E: Expertise (knowledge and experience) to define or to follow the instructions for the execution of the activity.
- W: Work to execute the activity.

Table 5: Example for distributed responsibilities

4.2 Test Workflows

In the following, the workflows component, integration, system, acceptance and regression testing will be explained for the test of component-based, embedded real-time systems. The concept of component, as stated within the framework of the DESS project, serves as important basis for those activities. This concept will be briefly addressed in the following section *Component Testing*.

In short, the workflows have the following contents:

- **Component testing:** testing a single, elementary software component.
- **Integration testing:** testing the interplay (the interfaces) between components and between components and their respective environment (also hardware).
- **System testing:** testing the overall system, containing software and hardware, from the developers' point of view.
- **Acceptance testing:** testing the overall system from the customers' point of view.

The first of the workflows mentioned are usually rather white-box-oriented, i.e. information on internal realisation are used when testing. The latter workflows are rather black-box-oriented, i.e. testing is much more based on the specifications of the respective test objects.

Regression testing means testing after modification of the test object.

4.2.1 Component Testing

This section will describe the testing of elementary units of a component-based system. The concept of component, as developed within the DESS project, will serve as a basis.

The definition of a component within DESS is the following:

*A **component** is a logically highly cohesive, lowly coupled, documented software module that can be used as a unit of development, reuse, composition and adaptation. It therefore is an exchangeable architectural element of a software system that acts as a part within a larger whole. It provides dedicated functionality that can be used in a specific application environment in order to accomplish higher-level goals.*

We can distinguish between a *component blueprint* and a *component instance*. A blueprint is a description of a component, whereas instance refers to the executable component. A blueprint may have several instances, and the behaviour of an instance is described by its blueprint. In contrast to a blueprint, an instance may have a state.

A component is part of a component-based system and has an interface in order to interact with other components. The interface consists of two parts: a *provided interface*

and a *required interface*. The component offers its services for other components via the *provided interface*, whereas the component uses the *required interface* to access the services of other components in order to fulfil its own functionality.

Among a variety of information, the blueprint of a component also provides test information for a component. This is information which can be used for the test of a component as well as results of already executed tests.

As defined within the DESS project, the notation of components (for specification and documentation) is oriented on the *Unified Modelling Language* (UML) (see www.omg.org). For more information regarding component definition and description see *DESS 01-1* and *DESS 01-2*.

Component testing aims at checking if the developed and isolated components behave as required, and if deviations from the target behaviour exist, respectively. For this purpose, they have to undergo isolated tests. The target behaviour of components can be taken from the design and the requirements.

Typical errors revealed by component testing include

- incorrect logical decisions,
- incorrect loop operation,
- error in implemented algorithms,
- inadequate performance,
- incorrect computation sequence,
- incorrect response to valid input data combinations, and
- incorrect response to invalid input data.

Component testing will be accomplished by a software engineer. It is very effective if the tester is an independent person, i.e. an engineer who has not developed the component under test. Software with a high criticality (e.g. safety or mission critical software) should always be tested by an independent tester, e.g. a test engineer or a member of the quality management. Component testing can be carried out either in the form of a black-box or a white-box approach. Test case determination techniques for both, black-box and white-box testing are described in chapter 4.3.2. In order to be able to test a component in isolation we need a test bed (or test harness) (see chapter 4.3.1). A test bed represents the minimal essential environment for the execution of the component.

4.2.2 Integration Testing

Integration testing will be carried out if several software components are integrated into one subsystem/system, and if components are integrated on the target hardware, respectively. The components' interrelation (the interfaces) will be tested as well as the components' interrelation with the target hardware. It is presupposed that the composed

ITEA CONFIDENTIAL

system parts (software as well as hardware) have already been tested in isolation. Integration testing can also be referred to as interface testing.

Interfaces between components are not merely the interfaces which are directly visible, like those via parameters, for example, that are explicitly handed over when calling components. There are also less obvious interfaces, like, for example, the integration via global data areas. Especially these hidden interfaces are very error-prone and therefore integration testing should be particularly thorough in these cases.

Errors which are primarily to be revealed by integration testing are errors in interrelation or communication between components and between components and their target hardware. In addition, it often happens, according to experience, that by using the method of integration testing, errors inside of components are revealed which were not revealed during component testing. The detection of component errors, however, rather remains the objective of component testing.

The reason for errors at interfaces to appear, although the system units involved in the interface have been tested separately, shall be clarified with the following example: Two methods, M1 and M2, realised by different people are to be integrated and subjected to integration testing. M1 calls M2 and provides M2 with input with the aid of corresponding input parameters of M2. During the single tests each of the programmers assumes that the other provides the testing of the validity of the input and each of them concludes his test successfully. During integration testing it turns out that the method M2 partly works with inadmissible input data since no testing of the input occurs (these and other interface problems can be reduced considerably by using formal interface specifications).

Typical errors revealed by integration testing include

- incorrect interrupt handling,
- failure to meet timing requirements,
- invalid software handling of hardware transients or failures,
- resource contention problems,
- errors in hardware/software interfaces,
- incorrect behaviour of feedback loops,
- stack overflow, and
- violation of the software partitioning.

Integration testing is carried out by the software developers. Also here it should be considered: the more independent the tester the more effective the testing will be. The architecture description generated during the design and higher system specifications (system design, requirements) serve as a basis for the test, as far as they contain statements on the architecture and on the interfaces between components. The integration to be tested must comply with the requirements contained within these descriptions.

ITEA CONFIDENTIAL

When components are reused the test of these within their new environment starts with integration testing. Dealing with reused components we proceed on the assumption that they have been tested before. During integration testing it is tested if the reused component proves itself in the new environment.

Integration strategies deal with procedures for a composition of system components. Well-known procedures are, e.g. top-down, bottom-up, and big-bang integration. Particularly those integration strategies are of interest that support the thorough testing of a system.

The top-down and the bottom-up strategy belong to the incremental integration strategies, and the big-bang approach belongs to the non-incremental integration strategies. When the incremental approach is used only one component is included into a new composition at the time, whereas several or all system components are composed at the same time with the non-incremental strategy.

The incremental integration possesses several advantages over the non-incremental integration. Two points, in particular, should be pointed out as relevant for testing: First of all, the incremental procedure offers a better basis for a systematic approach when testing and therefore secures a higher probability of revealing errors. Secondly, the incremental strategy offers better support for localising errors and their causes.

For object-oriented systems the top-down integration looks like this: Classes are integrated from the upper classes on down to the lower classes. When the bottom-up strategy is applied the procedure works in the opposite direction³ (cp. Figure 11). Where the top-down procedure is concerned proxies (in this case *drivers*) have to be generated which simulate the behaviour of the not yet integrated lower classes. For the bottom-up approach, on the other hand, proxies (*stubs*) are to be generated which represent the not yet existent upper classes. The effort that has to be exerted to generate proxies for upper classes is generally much higher than that needed for the generation of lower class representatives. For this reason the top-down procedure is the one to be favoured for the process of test execution where object-oriented systems are concerned.

In practice, the integration of large-scale systems is usually not just top-down or bottom-up. Rather, mixed forms are used, i.e. different strategies are applied to different parts of the system. For instance, in practice there is the so-called sandwich-integration, a combination of both approaches, top-down and bottom-up. Another, also for testing purposes, useful strategy is to assemble the core functionalities first, and then to integrate the secondary functions. This procedure allows to test important system functions very early and, if possible, together with the customer.

³ Note that top-down and bottom-up in OO-systems have other directions compared with top-down and bottom-up in traditional systems

Another possible approach for integration and testing could be as follows. First focus on system parts that contain completely new components. Secondly, parts that contain substantially changed components should be investigated. Finally, the re-used system parts are taken into account. Already used and thus well-tried components tend to prove more reliable than new, still untried components. The new components are thus critical points in the system. At these points, more precisely at the interfaces between those new components and their respective environments, integration testing needs to be particularly thorough.

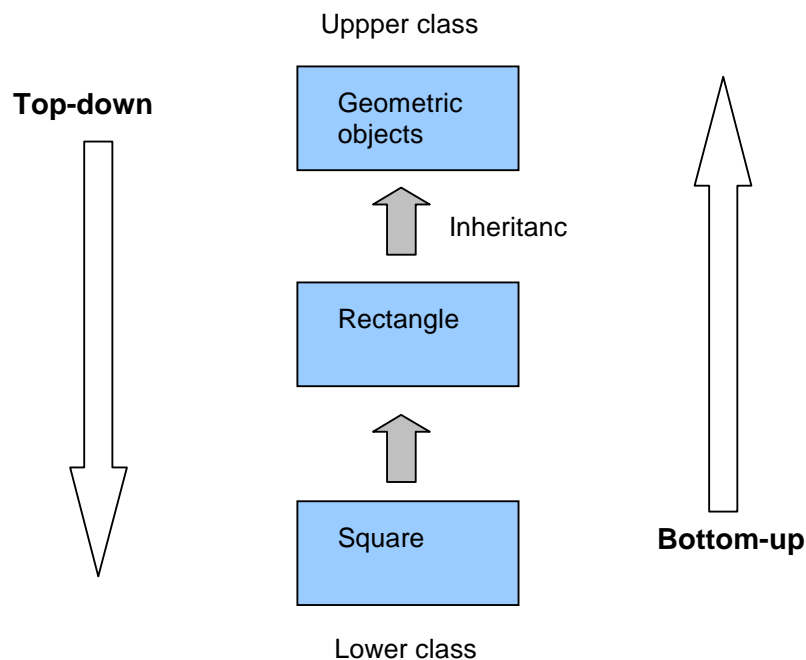


Figure 11: Top-down- and bottom-up integration in OO-systems

Executing integration testing with test data implies the generation of a test harness which allows the execution of the system/subsystem to be tested concentrating on the interfaces. The test harness consists of the minimal environment needed for the execution, and contains drivers and stubs simulating the behaviour of the real systems parts to be integrated at a later stage.

4.2.3 System Testing

System testing means testing the whole system (consisting of hardware and software), including GUI-testing. While integration testing focuses on the interfaces between software components and between components and target hardware, system testing focuses on testing the relevant quality attributes of the system as a whole.

System testing checks whether the total system meets the specified system require-

ments. In this case, the test object consists of the software system, being completely composed of components, and its hardware environment. The tests have to be performed in an environment that matches the operational target environment as much as possible. With this test, the primary view is that of the customer and of the subsequent system users, respectively. Developers and/or an independent testing team will carry out the system testing. High-level system descriptions, particularly the system requirements, the system design and manuals serve as basis for system testing.

The intention is to test whether the system meets the specified functions and whether those system states which are visible for the user and described in the specification can be achieved as required. In addition to that, the following characteristics need testing:

- Run time and memory behaviour of the system.
- Behaviour when processing large amounts of data.
- Interplay of the system with peripheral equipment such as printers and hard disks.
- Consistency of manual and system behaviour.

Performance testing checks the behaviour of memory space and time in both, usual and extreme situations. Among other things, the focus will be on the behaviour in extreme situations, for instance when processing large amounts of data, and on its response time behaviour in such a situation. **Robustness testing** exceeds the limit of acceptable charge, and tests the subsequent system behaviour. Software tools, being able to provide the system with large charges are useful for both, performance and robustness testing. **Beta testing** is testing the system by chosen pilot customers. It is one of the last tests being executed before delivering the system to the customer.

4.2.4 Acceptance Testing

Acceptance testing aims at receiving the customer's acceptance for the system. The test is carried out from the customer's point of view, involving the customer. It is intended to test whether the product meets the requirements of the customer, and whether it can be deployed. The software quality management should also be strongly involved in this test.

Acceptance testing is a high level requirements validation on the target. The test is carried out on the target computer, and from the customer's perspective. Technically viewed, acceptance testing is similar to system testing, but during acceptance testing the customer is strongly involved and the test is executed under real application conditions. The outcome of these tests enables the customer to determine whether or not to accept the system.

The main goals for acceptance testing are to determine whether a system meets its initial requirements, i.e. the requirements as defined in the requirements specification, as set-up by the customer. Therefore, it depends on the customer what specification will be the input for the acceptance testing. In general, system descriptions from the customer's

perspective, in particular user requirements and manuals (and maybe contracts), serve as basis for acceptance testing.

4.2.5 Regression Testing

Regression testing means to repeatedly test an already tested (sub)system after a change. Aim of the test is to ascertain whether the modified system fulfils its requirements (possibly modified meanwhile) as always or not.

Strictly speaking, it is tested whether

- unmodified system parts show unchanged behaviour,
- modified system parts operate as required, and whether
- the system as a whole fulfils its requirements.

Test information (in particular test data and test results) of previous tests will be reused thereby.

Repeated testing of an already tested system occurs, for instance, after

- errors were eliminated,
- the system was embedded in a new environment,
- a component was replaced by another, or after
- the system was extended by new functions.

Regression testing constitutes a large part of maintenance.

If a reused or a new component is going to be used in an already tested system, a subsequent regression testing will also have to be executed. With regression testing, we assume that the new component has already been tested in isolation, and may have been used in other systems. In this case regression testing concentrates on the interplay between the new component and its new environment (cp. chapter 4.2.2).

A major precondition for regression testing is analysing the impact of changes. Here, we have to determine which system parts are being affected by a change. Of interest are, apart from the purely static (textually determinable) changes, in particular the dynamic changes, i.e. changes which arise during system execution. The effect which results, for instance, from replacing one component by another is, statistically viewed, likely to be localisable. However, the dynamic system changes are usually much more far-reaching. For instance, the new component could (other than its predecessor) access to global data areas also used by other components, and could thus change the behaviour of the other components. These change effect need to be analysed in order to state an appropriate test strategy, and to determine effective test cases for regression testing.

4.3 Test Activities

The test activities described in the following are those that occur within the workflows (chapter 4.2). A model consisting of these activities is depicted in Figure 12. It consists of

- systematically designed, procedure-bound tests and
- intuitive, experience-based, procedure-free tests.

The procedure-bound method starts with the **test object analysis**, the aim of which is to determine testing units, relations between them, and testing requirements on the test objects and their relations. The available system descriptions form the bases of this analysis. After having identified what is to be tested by this means it will be determined in the following step of the **test distribution**, where, respectively in which environments and with which methods the test objects are to be tested or proved: the test objects, including the test requirements, are allocated to different test environments. In the phase called **test and tool design**, which follows this distribution, the concrete test set-up is specified. Test methods, test cases, expected values, and test supporting facilities to be employed are, among other things, all part of this. Thus, this phase determines how the testing should be carried out.

The aim of the **tool development** is to provide the necessary test supporting facilities in the form of hardware or software, respectively. If no (noteworthy) supporting facilities are necessary this phase can be omitted. Within the **test implementation** the test is developed, executed, and evaluated according to the test procedures described before (during test & tool design). The described process model is a cyclic model, i.e. re-entry from a later to an earlier phase is possible. After the performance of previously determined test end criteria the test objects are released from the point of view of the tester (**release advice**).

The procedure-free branch of the testing accounts for the experience that, despite all the systematics employed in the test object analysis and the test design, a certain amount of errors (even possible design errors) can only be detected by intuitive testing. With the help of procedure-free tests errors shall be detected which may be discovered by experience, intuition, and coincidence, rather than with the help of a systematic procedure. The awareness that only a sensible mixture of systematic and intuitive procedures leads to successful testing serves as a background to this double strategy.

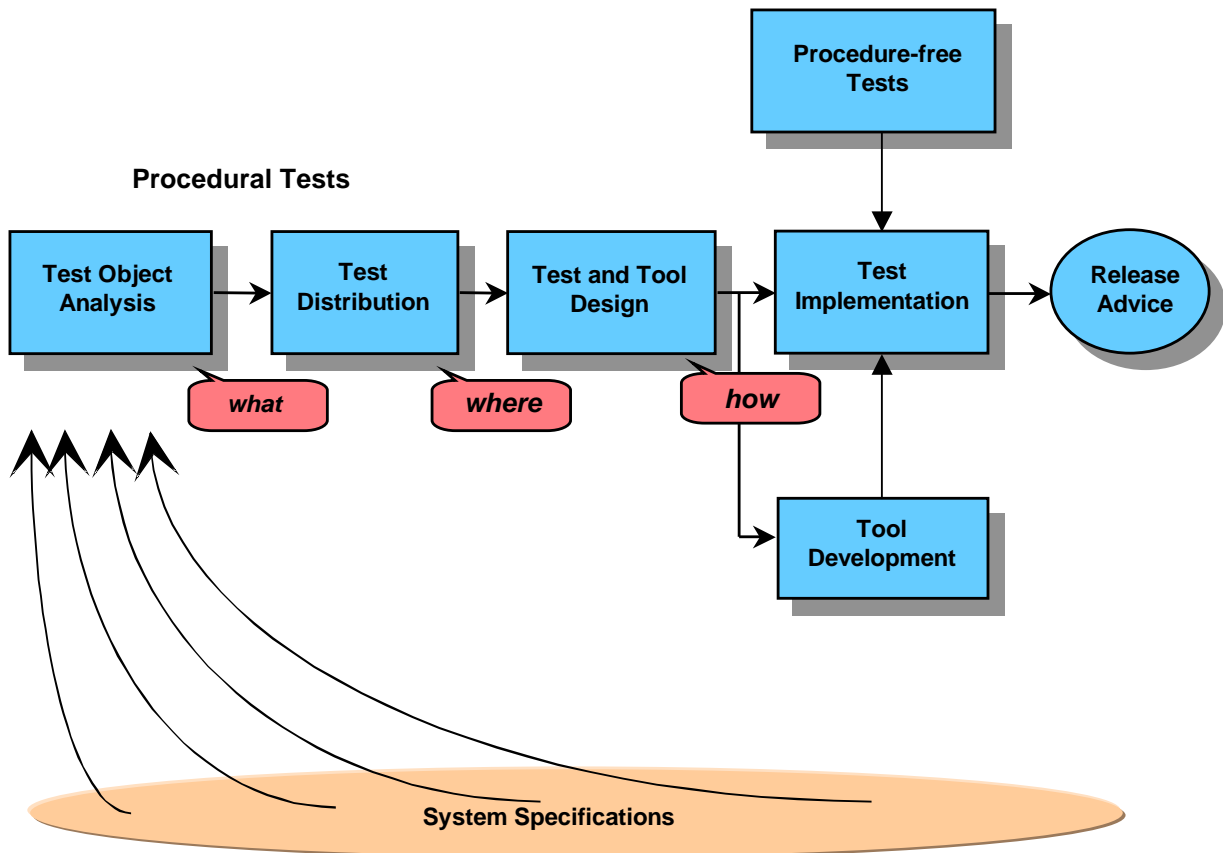


Figure 12: Test activity model

4.3.1 From Test Object Analysis to Test Evaluation

The goal of **test object analysis** is to identify

- *test objects* in the form of functions, components, interfaces, subsystems or others (depending on the test workflow),
- *relationships* between these test objects, and
- *test requirements* for the test objects.

System specifications are the foundations of the test object analysis. Depending on the test workflow, respective specifications are used as a basis of this analysis, e.g. the system architecture specification as the most important basis of the test object analysis during integration testing, and the component specifications as most important bases of component testing.

The following applies to testing in general: the more precise or formal the system specifications, which forms the bases of the test, the easier tests can be derived from them. Vague statements such as “*the system ought to be user-friendly*” and “*the system should respond within an acceptable period of time*” are much more difficult to test than

ITEA CONFIDENTIAL

more precise statements such as “*the system is supposed to keep open at the most five windows at the same time*” and “*the time of response should not exceed a maximum of 0.1 seconds*”.

In the course of the **test distribution** it will be determined which tests are to be executed within which test environments. The aim of the test distribution is to distribute the tests to be executed in such a way among the different test environments that, preferably, most meaningful tests can be realised at lowest possible costs. Examples for test environments are, among others, simulation environments, host computers, special devices, target computers, and the target system (e.g. a plane or a car). The result of a test distribution is a design which can easily be illustrated in the form of a matrix. (cf. Table 6).

	Simulator	Host	Subsystem Rig	Sytem Rig	Target Computer	Target System	...
Test object 1	+	+					
Test object 2			+		+		
Test object 3			+	+	+		
Test object 4	+			+	+		
Test object 5					+	+	
...							

Table 6: Test distribution matrix

In the **test design and tool design** phase the concrete test set-up is specified. Parts of it are

- testing techniques to be used,
- test cases,
- expected values or evaluation criteria,
- test evaluation procedures and
- test-supporting facilities (hardware and software tools).

A good basis of the test and tool design is a description of capabilities of testing techniques (cf. Table 7).

ITEA CONFIDENTIAL

Testing Techniques Capabilities	
Kind of test	Description of the technique: e.g. review, static test, dynamic test, white-box component test, black-box component test, integration-, system- and/or acceptance test
Test workflow(s)	List of test workflows where the test technique can be applied
Quality attributes	List of quality attributes that can be tested with this test technique

Table 7: Testing techniques capabilities

Each test technique is suited to test one or more quality attributes. Table 8 gives an overview of the dynamic quality criteria and test techniques.

	API Test	In-System Test	Fagan	...
Continuity		+	+	
Economy			+	
Functionality	+	+	+	
Performance		+	+	
User-friendliness			+	
Usability			+	
Security			+	

Table 8: Dynamic quality attributes versus test technique

Examples for testing techniques are described in the following. Fagan (Table 9) is a formal review technique (*Fagan 76, 86*).

Fagan	
Kind of test	Review, Static Analysis
Test workflow(s)	All
Quality attributes	All

Table 9: Fagan

API-tested software (Table 10) is software where all existing functions are called at least once in a test which is executed and passed, all enum values should have been used in a successful test and each possible event should occur in at least one successful test.

Requirements for the API test are:

- For each API call, at least one test has to be defined.

ITEA CONFIDENTIAL

- For each enum input value, at least one test has to be defined.
- For each event, at least one test has to be defined.

API Test	
Kind of test	Dynamic testing, Black-box Testing
Test workflow(s)	Black-box Component Testing
Quality attributes	Functionality

Table 10: API Test

The purpose of an in-system test (Table 11) is to test the product within a reference system which is very similar compared with the system the customer uses. Example: the product consists of hardware (chips) and its associated software which is built into a reference system. This reference system resembles the system that customers use.

In-system Test	
Kind of test	Dynamic Testing, Black-box Testing
Test workflow(s)	Integration Testing, System Testing, Acceptance Testing
Quality attributes	Functionality, Performance, Continuity

Table 11: In-system Test

Test cases are chosen input situations or stimuli for test objects¹. They are largely determined on the basis of the test object analysis' results, the system descriptions underlying the test, and the intended quality targets. For each requirement at least one test case should be generated which examines if the developed embedded software system meets the requirement (or not). In doing so, it is important to determine and test the quality criteria linked with the requirements.

Test case determination is an essential activity with the help of which the quality and scope, and thereby also the effectiveness and the costs of a test can be assessed. Additionally, it is a very creative phase in which the tester's imagination, intuition, and experience play a vital role. Test case determination often aims at detecting most possible errors and realising the highest possible test coverage. Chapter 4.3.2 will detail the various possibilities of test case determination.

The task of the **tool development** is to provide the necessary test supporting tools. Examples are data bases for managing the data used by testing (e.g. test objects, test cases, test data (inputs), test harnesses, test results) and tools for displaying test results graphically.

ITEA CONFIDENTIAL

The task of the **test harness generation** is to provide a framework for a test object in order for it to be executed. A test harness includes driver and stubs. A driver is used to call the test object. It typically provides test data, controls and monitors execution, and reports results of the execution. Stubs are used to simulate the behaviour of system parts called by the test object. A stub is a skeletal or special-purpose implementation of a system part.

Test execution means running the test object with test data. Each defined test case has to be executed and each execution of a test case has to be repeatable.

During the test execution the coverage of the test object, which was achieved by the test, can be measured on the basis of its structure. This process is called **coverage analysis**. This analysis wants to determine the scope and proportion which the executed tests have brought to bear upon the execution of the test object. The general aim of this activity is to get a feeling for the quality of the executed test. Examples of coverage criteria are given in chapter 4.3.2.4. Prerequisite for the analysis is an instrumentation of the test object code by means of inserting additional statements (counters) into the code.

The degree of test coverage achieved with a test run is defined as the ratio of the number of the actually passed structure elements to the number of elements which are possible in principle (a coverage of 60 statements of a component which contains 120 statements in total produces a statement coverage of 50 percent for this component).

It is advisable to realise the coverage analysis as a separate test execution process, i.e. to implement two test runs: one test run without and one with instrumentation. Both test runs are executed with the same test data. The reason for a separate test run for the coverage measurement is that the instrumentation changes the test object, e.g. its temporal behaviour. In the worst case the instrumentation could even lead to errors within the test object. On the basis of the idea of a separate test run for the coverage analysis the following testing procedure can be defined:

- Step 1:* Determine functional (black-box) test cases, generate test data, predict expected values, execute and evaluate test (target/actual value comparison).
- Step 2:* Instrument test object, execute test with previously generated test data and determine coverage.
- Step 3:* Considering the obtained structure coverage, possibly determine further test cases and execute tests in the manner described above (steps 1 and 2) until the envisaged coverage target or the general test end criteria has been achieved.

A high coverage does not necessarily mean a detection of all errors. An example for this case is the coverage of the statement

$$A = B/C .^4$$

In most cases the execution of this statement will cause a problem if C equals zero. The test will only detect the problem, even with a complete path coverage, if zero is used as a value for C. Thus, the coverage criterion *path coverage* is not sufficient to detect the problem. Consequently, the detection of errors is very much dependent on the used concept of structure (in this example: the path) and on the selection of test data passing through this structure. The finer the chosen structures and the higher the coverage of these structures, the higher the probability of error detection. With a granulation consisting of statement, branch, and path coverage, the statement coverage would be a rather coarse structure, whereas the path coverage would be a rather fine structure.

Test evaluation means analysing the results of the executed tests. This includes the comparison of actual with expected results and evaluation of coverage results provided by coverage analysis.

After having completed the test process, the process as well as the results should also be analysed, and improvement possibilities of the test process should be identified (TPI). For this, developers, quality assurance personnel, and the project management board should get together. A very minimal criterion for stopping the test process is that all user requirements as stated at the beginning of the development are tested.

4.3.2 Test Case Design

Test cases are chosen input situations or stimuli for test objects. Some important criteria for test case determination are introduced in the following. It concerns test case determinations on the basis of

- partitioning cases in normal-range, boundary, and robustness cases,
- equivalence partitioning,
- the usage of the Classification-tree method,
- the test object structure, and
- the usage of procedure-free test cases.

4.3.2.1 Normal-range, boundary, and robustness testing

In general, test cases or test situations can be divided into three classes:

- **Normal-range testing:** The system works with admissible standard values.
- **Boundary testing:** The system is in a boundary or extreme situation.

⁴ A becomes B divided by C

ITEA CONFIDENTIAL

- **Robustness testing:** The system is provided with inadmissible values.

Test cases from each of the three operating modes are to be used for the test.

Normal-range test cases demonstrate the ability of the software to respond correctly to normal inputs and conditions.

Examples of normal-range test cases could be the following:

- Real and integer input variables with values within the valid range of the types.
- For time-related functions, multiple iterations of the code should be performed to ensure correct implementation of time-based requirements.
- For state transitions, test cases should be developed to exercise the normal operation of the state machine.
- For software requirements expressed by algorithms, the normal range test cases should verify the use of variables and boolean operators within the algorithms.
- Enumerate input variables with value within the valid range of the types.

According to experience, **boundary testing** has a high probability of error detection. Examples for boundary testing are:

- The challenging of a processor with a dense sequence of interrupts.
- The behaviour of a software loop in its final run.
- The transition of one system state to another.

It is sensible to use not only the boundary itself but also the situations and values around the boundary value when applying the method of boundary testing. Thus, it can be tested, among other things, if the system handles the transitions between normal-range, boundary and robustness testing correctly.

Robustness testing serves to test the robustness of the system. It is tested, how stable the system behaves when it is handled incorrectly, and if the designated exception handling functions correctly. Examples for robustness testing are:

Test cases for robustness testing could include:

- Operator errors.
- Downloading of data, the amount of which exceeds the specification limit.
- Real and integer variables should be exercised with invalid values, where possible.
- System initialisation should be exercised during abnormal conditions such as component failure.
- The possible failure modes of the incoming data should be determined and exercised, e.g. invalid data input from the system.

ITEA CONFIDENTIAL

- For loops where the loop count is a run-time computed value, test cases should be developed to attempt to compute out-of-range loop count values, thus demonstrating the robustness of the loop-related code.
- A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly.
- For time-related functions, test cases should be developed for arithmetic overflow protection mechanisms.
- For state transitions, test cases should be developed to provoke transitions that are not covered by the software requirements.
- Enumerate variables with invalid values, where possible.

These types of test cases demonstrate the ability of the software to respond to abnormal inputs and conditions.

4.3.2.2 Equivalence Partitioning

Another possible test case determination is provided by equivalence partitioning. This procedure involves the partitioning of test situations (input domain of the test object) into classes, each class testing specific aspects of the system. Different classes are testing different aspects, and all elements and values, respectively, of one class are testing the same aspect. Thus, for the concrete test, it will be sufficient to use just one or a few values of one class at a time. These values will cover the entire class. An example for such a partitioning would be the segmentation of the input domain of a function in order to have class 1 test system requirement 1, class 2 test system requirement 2 and so forth. Equivalence partitioning requires a more thorough analysis than the previously mentioned classification of normal-range, boundary, and robustness testing. In each case, it should be examined whether the effort is justified.

In the following, we will describe the classification-tree method, which is a concrete form of equivalence partitioning.

4.3.2.3 Classification-tree method

The classification-tree method (*Grochtmann and Grimm 93, Grochtmann et al. 95*) is a special approach to (black-box) partition testing partly using and improving ideas from the category-partition method defined by Ostrand and Balcer (*88*).

By means of the classification-tree method, the input domain of a test object is regarded under various aspects assessed as relevant for the test. For each aspect, disjoint and complete classifications are formed. Classes resulting from these classifications may be further classified - even recursively. The stepwise partition of the input domain by means of classifications is represented graphically in the form of a tree. Subsequently, test cases are formed by combining classes of different classifications. This is done by using

the tree as the head of a combination table in which the test cases are marked. When using the classification-tree method, the most important source of information for the tester is the functional specification of the given test object. A major advantage of the classification-tree method is that it turns test case design into a process comprising several structured and systematized parts - making it easy to handle, understandable and also documentable.

The use of the classification-tree method will be explained using a simple example. The test object is a Computer Vision System which should determine the size of different objects (Figure 13). The possible inputs are various building blocks. Appropriate aspects in this particular case would be, for example, the size, the color and the shape of a block (Figure 14).

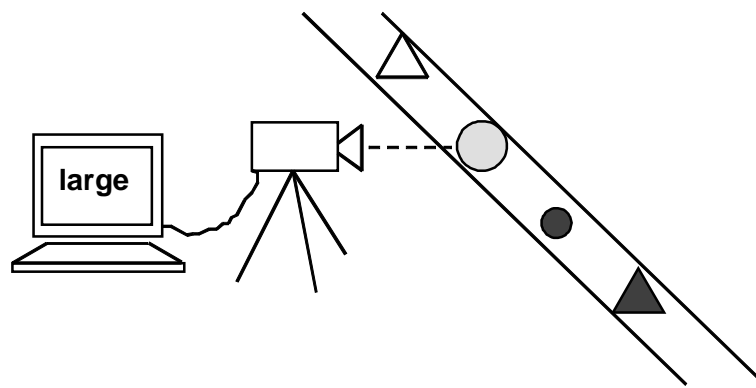


Figure 13: Computer Vision System

The classification based on the aspect 'colour' leads, for example, to a partition of the input domain into red, green and blue blocks, the classification based on the shape produces a partition into circular, triangular and square blocks. An additional aspect is introduced for the triangle class: the shape of triangle. The various classifications and classes are noted as classification tree (Figure 15). Some possible test cases are marked as examples in the combination table associated with the tree. Test case three, for instance, describes the test with a small blue isosceles triangle.

The classification-tree method is especially suited for automation since (a) it decomposes the test case design process into several steps which can be automated individually allowing the tool to appropriately guide the user and (b) it offers a graphical notation well suited for visualization in a modern graphical user interface.

The classification-tree editor CTE is based on the classification-tree method and supports systematic and efficient test case determination for black-box testing (*Grochtmann et al. 93*). The two main phases of the classification-tree method - design of a classification tree and definition of test cases in the table - are both supported by the tool. For each phase a suitable working area is provided.

The classification-tree editor CTE uses a separate window on the screen (Figure 16). In the upper part of the window there is a drawing area in which the user can build up a classification tree interactively (*Draw Pad*). The lower part of the window depicts a corresponding table in which test cases can be marked interactively (*Table Pad*). Each test case row is numbered (*Testcase Pad*). The menu bar offers access to several pull-down menus which provide various commands, e.g. for saving, editing and printing.

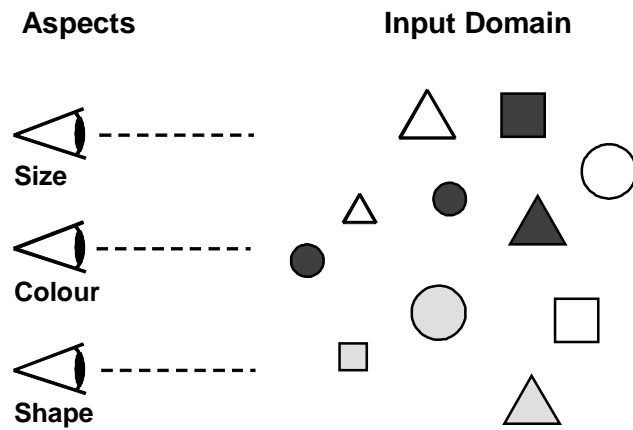


Figure 14: Aspects for classification

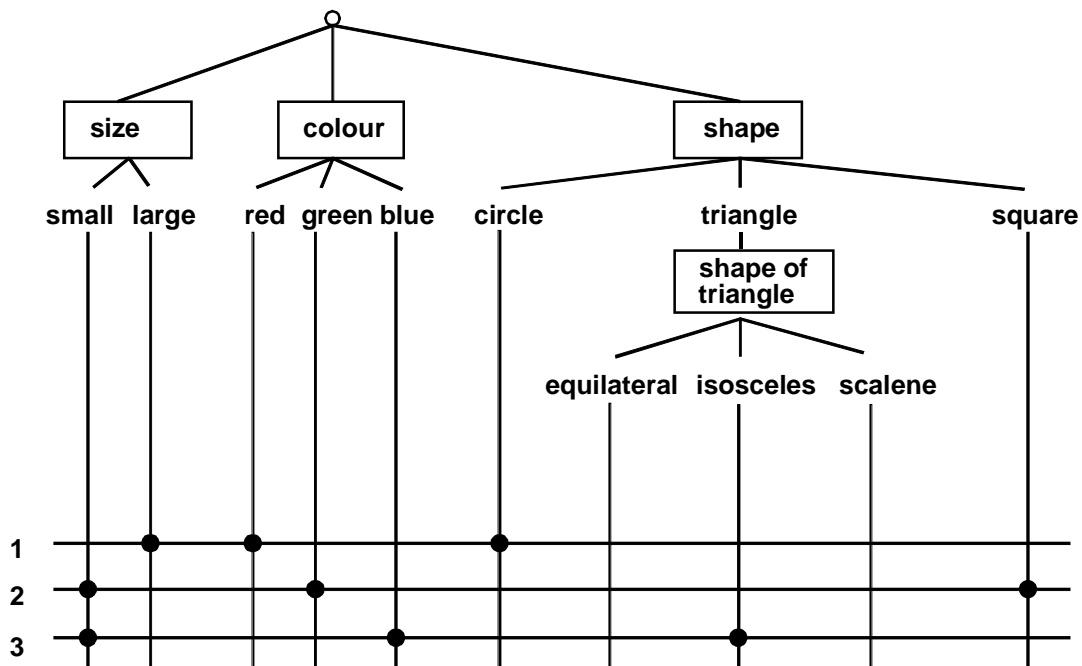


Figure 15: Classification tree

To give the user optimal support, editing is done in a syntax-directed and object-oriented way. Several functions are performed automatically. These include drawing of connec-

tions between tree elements, updating the combination table after changes in the tree and checking the syntactical consistency of table entries.

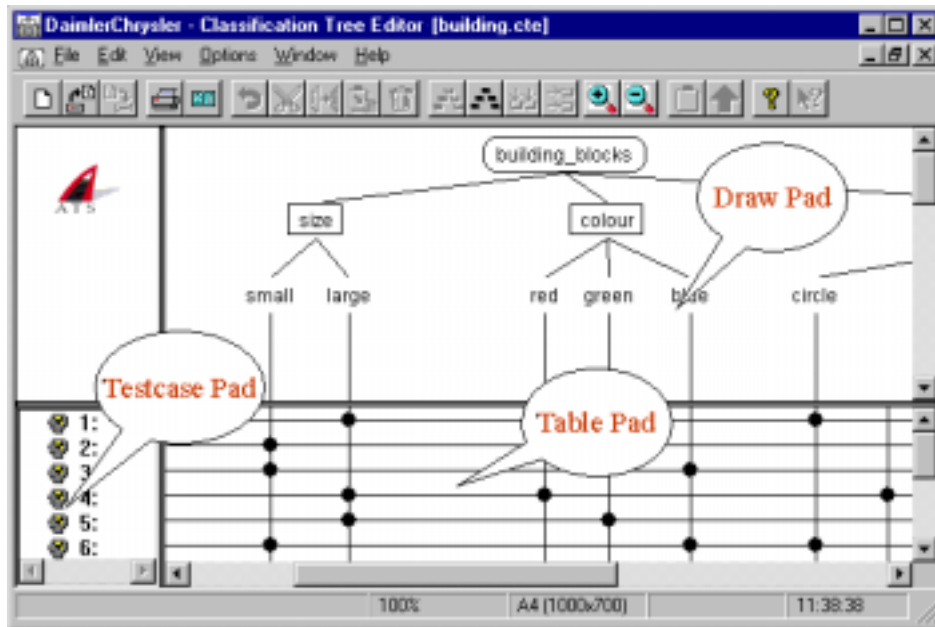


Figure 16: CTE

The CTE offers features which allow large-scale classification trees to be structured in order to support the test case design for large testing problems efficiently. As test documentation plays an important role in systematic testing, the CTE offers suitable support for this activity. For example, the test case design can be documented easily by printing out the trees and tables. Furthermore, the tool can automatically generate text versions of the test cases, based on the test case definition in the table. The CTE is an integral part of the overall computer-aided test system TESSY (Wegener and Pitschinetz 95). For more information concerning the CTE visit www.ats-software.de.

4.3.2.4 Structure-oriented testing

Structure-oriented testing (also referred to as coverage testing) keeps to the structure of the test object, i.e. it follows the form rather than the semantic content of the functionality of the test object. Basically, these tests always intend to put into execution specific structural elements, e.g. requirements, components, functions, paths, statements. A highest possible coverage is usually aimed at. The higher the quality requirements for the software to be tested, the higher the coverage should turn out.

Typical examples for this kind of testing are statement, branch, and path coverage on the basis of control flow graphs, data flow coverage on the basis of annotated control flow charts, and coverage of call graphs (see, for example, Frankl and Weyuker 88, Jin

ITEA CONFIDENTIAL

and Offutt 98, Ntafos 88, Rapps and Weyuker 85. More than a hundred different kinds of coverage are listed in Kaner 96).

Basically, structure-oriented testing may be used

- to determine test cases, and
- to determine the test object coverage as achieved by executed tests, and thus to obtain support when assessing the test quality (as already explained in ch. 4.3.1).

If the aim is to determine test cases for coverage, test cases will be derived from the internal structure. The intention is to put into execution specific structural elements, and to check whether the test object confirms the expected behaviour, or whether there are derivations from it. After having defined the test criterion (e.g. statement coverage), test data have to be generated leading to a run of the system parts to be covered.

Following the derivation of test cases from the test object structure, and previous to their application to the test object, it is imperative to test whether the determined structure tests are semantically sensible, and whether they should be put into execution. It may thus be the case that a specific coverage criterion is already included in an other criterion. In this case, it would be sufficient to use the more comprehensive criterion. Examples will be mentioned beneath.

Structure-oriented testing does not imply that the relation between structure element and the functionality connected to it is directly apparent. Rather, the relation has yet to be determined. At first, the test case only signifies that a structure element (e.g. a specific internal statement) has to be put into execution. In order to achieve this, we have to determine an input leading to it. Here, it may be useful to determine the functionality which is connected to the structure element. Not later than the determination of the outputs expected, this functionality has to be determined.

A major strength of structure-oriented testing is its support when uncovering unreachable system parts (e.g. deactivated code in a component). The reason is that structure-oriented testing usually implies to aim at a high coverage of the internal structure, thus quickly making clear which system parts cannot be executed. A weakness of structure-oriented testing however is that, in its pure form, it lacks the view on the functional requirements. For instance, missing paths (or missing code) are not likely to be detected without knowledge of the functional requirements.

Different from structure-oriented testing, functional testing concentrates more on the functional expected behaviour of the system, as required by the test object and as usually specified in the requirements. The test object is being regarded as black box, and test cases are derived from the functional specification.

The two testing approaches do not compete. Rather, both approaches complement each other, thus presenting a highly efficient technique for testing. For the purpose of a customer-oriented or user-oriented testing, structure-oriented testing should be used rather

ITEA CONFIDENTIAL

as a complementing, and thus not as a primary test (case determination) criterion. The primary test criterion should aim at testing the compliance with functional requirements of customers or system users by the test object. Thus, primary testing basically concentrates on the semantic, functional requirements as purely structure-oriented tests. However, structure-oriented testing certainly is a useful supplement to functional testing.

A most effective testing strategy will combine both approaches as follows (compare Figure 1 and Figure 17): first, functional testing will be carried out. Test cases will be determined on the basis of functional requirements, test data be generated, expected results be determined, the test object will be executed with the test data, the actual results be determined, and finally the test will be evaluated.

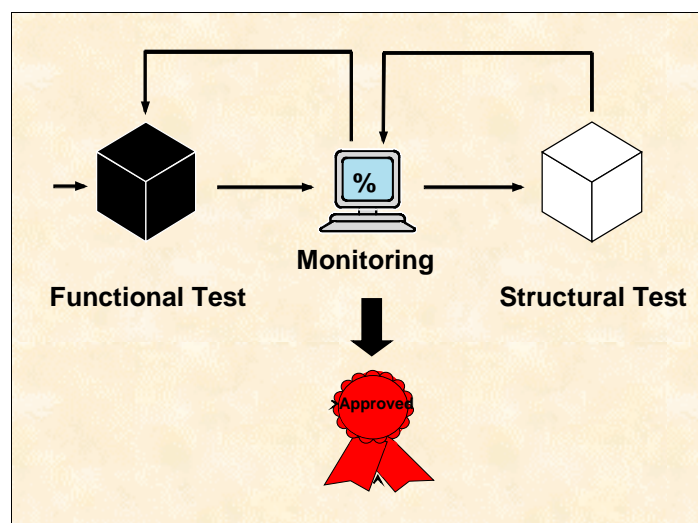


Figure 17: Effective test strategy

After having completed functional testing, structural testing according to a previously stated coverage criterion will be executed, but only to that extent yet uncovered by functional testing. If, for instance, the structural coverage criterion requires to achieve 90% of all internal branches, and 70% have already been covered by functional testing, only those test cases for covering the remaining 20% of the branches will have to be determined by structural testing. This strategy thus has its focal point on functional testing. Structural testing will only be used if necessary to complement the previously determined internal coverage.

In order to be able to implement this strategy, monitoring the test object will be necessary. Thereby, we have to detect in which parts the tests put into execution the internal structure of the test object. In a restricted sense, this monitoring or coverage analysis gives evidence on the quality reached by testing, if quality is understood as a form of highest possible coverage of the test object structure.

ITEA CONFIDENTIAL

In the following, some structure-oriented testing criteria, being of interest within the scope of the DESS project, will be discussed.

Requirements-based test coverage is one of the criteria already mentioned: for each requirement, at least one test has to be executed.

On the basis of a control flow graph, different coverage measures could be defined, e.g. node, edge, and path coverage. A control flow graph is a graph with nodes and directed edges representing statements, and transitions between statements. In this context, we may assume that the graph has exactly one start node and one end node. In case a graph should have various of these special nodes, the following ideas may be transferred respectively.

The following are examples for coverage measures:

- **Statement coverage:** Each statement has to be executed at least once. The requirement that each programmed statement has to be executed at least once when testing certainly is a clear minimal criterion. However, this is a very weak criterion and proves insufficient for thorough testing.
- **Branch coverage** (also referred to as **decision coverage**): Each branch (each edge in the control flow graph) is to be executed at least once. Empty branches, i.e. branches without statements have to be considered, and to be executed, as well. Branch coverage includes statement coverage, and puts greater emphasis on the control structure of the test objects than statement coverage.
- The following types of **condition coverage** may be distinguished:
 - **Simple condition coverage:** Assumes that a decision is composed of atomic single conditions. Atomic single conditions are elementary conditions, which themselves do not anymore consist of simpler conditions. Each atomic single condition has to be set at least once to *true*, and at least once to *false* by the execution.

Example: Decision $((A=0) \text{ AND } ((B<0) \text{ OR } (C>0)))$ has three atomic conditions: $(A=0)$, $(B<0)$, and $(C>0)$. Each of these has to be set at least once to *true*, and at least once to *false*.
 - **Minimal multiple condition coverage:** Each single condition, atomic or not, has to be set at least once to *true*, and at least once to *false* by the execution. In contrast to the simple condition coverage, this approach accounts for the structuring of decisions (by means of brackets).

Example: In addition to the coverage described in the example above the compound conditions $((B<0) \text{ OR } (C>0))$ and $((A=0) \text{ AND } ((B<0) \text{ OR } (C>0)))$ has to be set at least once to *true*, and at least once to *false*.
 - **Multiple condition coverage:** Any possible combinations of truth values for all

ITEA CONFIDENTIAL

single conditions of a decision have to be put into execution.

- **Path coverage:** Execute at least once each possible path based on the control flow graph from start to finish. With the occurrence of loops, there will always be an indefinite number of paths to be executed. Thus, a certain number of loop iterations which are interesting from the point of view of the test have to be defined. Path coverage subsumes branch coverage (and thus, statement coverage as well).

Detailed measures are to be found in *Beizer 83*, for example. The more stringent the coverage criterion, the more thoroughgoing the test will be. The weakest criterion is statement coverage, the strongest is path coverage. From an economic point of view, branch coverage presents a feasible criterion. Many testing tools to be available at the market support this criterion. This is also true for the tool TESSY (*Wegener and Pitschinetz 95*) which was developed by the DaimlerChrysler research department.

Another kind for coverage criteria are coverings of call graphs. The call relations, existing between the functions of the system to be tested, may be illustrated by means of a call graph. The nodes appearing in this graph represent functions, whereas the edges represent the available call relations. An edge will point from A to B, if function A is calling function B. For the test, different coverage measures may be identified on the basis of this graph. The control flow coverage measures statement, branch and path coverage, respectively, as already described above and known from the functional level, will reoccur here on a higher level.

Linnenkugel and Müllerburg (*90*) mention different measures for the coverage of a function call graph, which will be discussed in the following:

- **all-modules (functions):** each function has to be called at least once.
- **all-relations:** each call between two functions has to be executed at least once.
- **all-multiple-relations:** any call available between two functions each has to be executed at least once. Multiple executions are applied if, and only if, the calling function has various call sites for the function called.
- **all-call-sequences:** each, on the basis of the call graph, possible call sequence of functions is to be executed at least once.

If the call graph contains loops, there will always be an indefinite number of paths. Accordingly, an indefinite number of call sequences would have to be executed. In order to handle the last-mentioned coverage criterion, the authors have defined two additional measures, each stating a maximal number of loop iterations.

In case, the nodes occurring in the call graph are not functions but components, each graph node represents a system unit which usually exports not only one, but several functions (cp. chapter 4.2.1). For this case, similar to the measures above, the following coverage criteria could be defined:

ITEA CONFIDENTIAL

- **all-exports:** each exported function is to be executed at least once.
- **all-imports:** each export is to be executed by each importer at least once.
- **all-multiple-imports:** all call points of imported functions are to be executed at least once.
- **all-import-call-sequences:** all imports are viewed as unique imports, i.e. for each import, only one call point in the component to be imported is taken into account. All call points in addition to that may be neglected. Based on this consideration, all possible sequences of all unique imports in the system are to be executed at least once.
- **all-multiple-import-call-sequences:** considers all call points of imports available in the system. All sequences possible on the basis of these call are to be executed at least once.

As already mentioned, for further examples of coverage criteria see *Kaner 96*.

4.3.2.5 Procedure-free tests

By means of procedure-free tests, those errors are to be detected which may be discovered by experience, intuition, and coincidence, rather than with the help of a systematic procedure. In connection with procedure-free testing, two main approaches will be stated: **error guessing** and **random testing**.

Error guessing, also referred to as intuitive testing, is a very common and highly effective approach for error detection, based on the experience and the intuition of the tester. By experience, the tester intuitively defines test cases which frequently lead to success, and detect errors. For example, the tester may suspect a test case when entering a special value such as zero. Error guessing includes this case in the test on the basis of the tester's assumption. It will be of advantage if the tester has a high degree of practical experience.

Random testing generates test data on the basis of chance. Often, the assumptions previously made on the probability of the error detection force of possible inputs to the test object are being taken into account. Random testing is usually extensive when determining expected results, since the functional context, in which the generated test data belong, still has to be identified.

4.4 Test Process Improvement

The topic of process improvement is a very important one in the course of specifying and handling a process. This also applies to the test process described in this paper. Testing is often criticised as being ineffective and expensive. The aim of the test process improvement is to change the existent process in such a way that the test will become more effective and the connected efforts (e.g. money, manpower, time, technical infrastructure) will be reduced. A test is effective if it yields the greatest possible insight into

ITEA CONFIDENTIAL

the test object's quality.

It is to be determined carefully for each individual case what improvement means concretely. An example for a rather insufficient test process is one

- in which testing starts very late in the development process (i.e. shortly before the product will be used),
- which is planned insufficiently (i.e. consecutive steps are not synchronised properly, responsibilities are not clear),
- in which the test personnel is inadequately qualified,
- for which hardly any methods and tools are provided,
- which can only be controlled with difficulty.

This example obviously offers a number of possibilities for improvements. As the costs for the correction of an error are lower the earlier the error is detected, it is sensible to start the test as early as possible in the development process. Proper planning, good controllability, well-instructed test personnel, and efficient methods and tools will equally contribute to a further improvement of this test process.

Any process improvement cannot be regarded without taking into account its context. A test process is part of a development process, which in turn is part of an organisation. The test process thus has to blend with this environment, and it has to be able to quickly adapt to this environment. In case the test process is being improved, and the underlying development process remains unchanged then a new testing may show an increased effect only to a certain degree. It proves useful to examine the effects of a modified test process on its environment (development process, organisation), and to alter the processes of the environment if necessary. Conversely, the effects of the environment on the test process should also be considered.

If the improvement consists in, for instance, a new test case determination technique being based on a specific notation of requirements the new technique can only be employed if this notation is used for the specification of requirements (i.e. within the requirement process) as well.

A guideline for improving the test process is the following:

- **Determine optimisation aims.** This step clarifies, for instance, the scope of improvement, which test processes may be affected by the improvement, and which concrete optimisation aims are followed up. For example, the following aims are possible: testing is to become faster, to be less cost intensive, and testing is to detect more errors than it does at present.
- **Determine actual state.** The test process in its present form is determined and described.
- **Define desired state.** Considering the optimisation aims, the desired test process is

ITEA CONFIDENTIAL

being specified.

- **Implement changes.** Steps are taken which supports the achievement of the desired state.

The psychological aspect of change processes has to be considered, i.e. the effect on the people affected. Effective changes definitely imply the acceptance of all participants! This has to be taken into consideration both, when defining the desired value and when implementing changes. In case changes are being asserted against the consent of all participants they are likely to produce a counter effect, i.e. the state deteriorates.

Some general, application-independent measures for test process optimisation include the following:

- Start testing in the development process as early as possible, if possible already during the requirements phase.
- Employ efficient, re-usable methods and tools. Overall, strive for a high automation of the test.
- Test planning and test execution by a qualified and independent team. By independent, we mean that people in the test team will not be directly involved in the development.
- In conjunction with the quality management, the test process should possess a preventive effect on the development process in order to avoid the occurrence of errors at all. In addition, the testability of the system to be produced is equally to be guaranteed.
- Continuously measure and improve the quality of the test process.

In software development, there is a variety of models in aiming at process improvement, for instance the well-known *Capability Maturity Model (SEI 95)*. These models only very abstractly address the test process, thus there are only very rough guidelines for the concrete deployment in practice of test improvement. Beyond these models, there are some improvement models which are more specifically concerned with the test process.

- *Testability Maturity Model (TMM)* by David Gelperin (*Gelperin 96*).
- *Test Improvement Model (TIM)* by Ericson, Subotic, and Ursing (*Ericson et al. 96*).
- *Testing Maturity Model* (ebenfalls TMM) by the Illinois Institute of Technology (*Burnstein et al. 96*)
- *Test Process Improvement Model (TPI)* by Tim Koomen and Martin Pol (*Koomen and Pol 99*, www.iquip.com).

When compared to the other test optimisation models, TPI aims much more at the deployment in practice and contains more detail (e.g. more optimisation steps and instructions). Concrete guidelines and instructions are given in order to obtain process im-

provement.

5 Conclusion

We presented a comprehensive concept for the validation and verification of embedded real-time software systems. As basis serves the proceeding model developed in DESS for the realisation of embedded real-time systems, which is V-model-oriented and which has been adjusted and extended according to the particular requirements in DESS (embedding, real time, object orientation, and components).

The DESS realisation model consists of workflows such as requirements engineering, design, coding and integration. This *Realisation-V* is being accompanied by a *Validation-V*, which describes the analysing activities applicable parallel to software construction. The Validation-V has workflows like review, model-checking, component testing, integration testing, system, and acceptance testing, respectively.

Since the test is of great importance as practical analysing technique, we considered it in more detail than the other forms of V&V. The test model contains four submodels: test management, test workflows, test activities, and a model for test process improvement.

A V&V concept for software systems will be the more useful, the more particular software properties are being considered. With the systems considered in DESS, the following properties are in the centre: embedding, real time, components, and object orientation. All these properties are addressed in this guideline.

This guideline is the result of a co-operation of DESS partners who worked together on this subject. This work is primarily based on the fact that all DESS partners provided their knowledge and their experience of testing software-based systems, which were documented in a state-of-the-art report. On this basis, due to the co-operation at various DESS meetings and workshops, and last not least by communicating via working papers and emails, this guide could be created. The co-operation went very well. Realising a workshop, being dedicated to the creation of the V&V guide, proved truly valuable. A possibility to improve co-operation in future projects should include to carry out workshops as early as possible.

The suitability of the V&V concept described in this paper is shown in particular in the practical use of the methods and techniques described in it. I.e. only the intensive practical use of this guideline will demonstrate its value. It is certainly very useful to take into account any feedback of the V&V guide for the future use and modifications of the guideline.

ITEA CONFIDENTIAL

Glossary

Acceptance test	Testing the complete (fully integrated) system under the customer's point of view. Test conducted to determine whether or not a system satisfies its acceptance criteria as defined in the customer/user requirements specification
Activity	A tangible unit of work in a workflow
API	Application Programmers Interface
API test	A test technique in which the API of a component is tested. API tested software is software where all existing functions are called at least once in a test which is executed and passed, all enum values should have been used in a successful test and each possible event should occur in at least one successful test
Artifact	Intermediate or final development product (e.g. project plan, requirements, design, architecture, model, (component, sub-system, system) code, V&V plan, test cases, glossary, manual)
Beta test	A test for a computer product prior to commercial release. Beta testing is the last stage of testing, and normally involves sending the product to beta test sites outside the company for real-world exposure
Black-box test	Tests that are defined without knowledge of the internals of a software system. It is a test based on the external interface and the specification of a system
CMM	Capability Maturity Model
Component test	Test of a single component
Coverage analysis	Analysis with the aim to gather information about the coverage of the test object obtained by test execution
Embedded (software) system	A (software) system designed to perform a dedicated function within a larger system
Instance	An individual case of a set of things. E.g. an object is an instance of a class.
In-system test	Test in which a component is tested in a complete system. It concentrates on performance, the combination with other components and stress tests
Integration test	Testing interfaces between system parts (i.e. between soft-

ITEA CONFIDENTIAL

	ware components/subsystems or between software parts and it's hardware environment) during integration in which software elements, hardware elements or both are combined and tested until the entire system has been built
Real-time system	A system which is able to response to an external event within a given time, i.e. a system which is able to meet time requirements
Regression testing	Testing after modification of a previously tested system to verify that the modified system still meets its requirements
SQA	Software Quality Assurance
Static analysis	V&V activity which does not involve execution of the test object, but analysing texts describing the software system
System	A running system, consisting of hardware and software
System test	Testing the complete system under the developers' point of view. Test to verify and validate whether the system meets its requirements
Test activity	A piece of work occuring within a test workflow
Test and Tool Design	Phase in which the test and the tools supporting the test are specified in detail
Test bed/harness	Minimal environment necessary for executing the test
Test case	An abstract description of inputs ¹
Test distribution	Distribution of test objects onto different test environments
Test environment	The total set of techniques, tools (software and hardware) and activities that are used to validate and verify the software products
Test evaluation	Evaluation of the results obtained by test execution (incl. coverage results)
Test execution	Execution of the test object with test data, producing an outcome
Test management	Managing, planning, and tracking the whole test process
Test object	The object or item to be tested (e.g. a component, a subsystem, a system, an interface).
Test Object Analysis	Phase in which the test objects, relations between them and test requirements are determined
Test plan	Documentation that specifies the scope, approach, resources

ITEA CONFIDENTIAL

	and schedule of intended testing activities
Test release	Release of the test object from the tester's point of view
Test workflow	Workflow occurring during testing, e.g. component, integration, system, acceptance, or regression testing
Testing	Dynamic examination whether an item under test meets its requirements
TPI	Test Process Improvement
V&V activities	Validation and Verification activities
Validation	"Do we build the right product?". The developers (and the customer) examine if the development results in the product the customer desires. The primary focus of validation is customer satisfaction
Verification	"Do we build the product right?". The developers check whether they are working properly during the development. It is evaluated whether the products of the given development phase satisfy the conditions imposed at the start of that phase
White-box test	Test that is defined with knowledge of the internal structure of the test object. It mainly concentrates on the verification whether the implementation of the system is OK
Workflow	A set of related activities producing a collection of related artifacts

References

- Beizer 83** Beizer, B.: *Software Testing Techniques*. Van Nostrand Reinhold Company, New York, 1983
- Binder 99** Robert V. Binder: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999
- BS 7925** *Standard for Software Component Testing BS 7925-2*. British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST), www.testingstandards.co.uk
- Burnstein et al. 96** Burnstein, I., T. Suwannasart, and C.R. Carlson: *Developing a Testing Maturity Model: Part I and II*. Illinois Institute of Technology, 1996
- Conrad et al. 99** Conrad, M., Doerr, H., Fey, I., and Yap, A.: *Model-based Generation and Structured Representation of Test Scenarios*. Proceedings of the Workshop on Software-Embedded System Testing (WSEST '99) at the National Institute of Standards and Technology, Gaithersburg, Maryland, USA, 1999
- DESS 00** *Testing software-based Systems - State of the Art*. DESS document, September 2000. Available at www.dess-itea.org
- DESS 01-1** *Definition of Components and Notation for Components*. DESS document, April 2001. Available at www.dess-itea.org
- DESS 01-2** *The DESS Methodology*. DESS document, December 2001. Available at www.dess-itea.org
- DO-178B** *RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Requirements and Technical Concepts for Aviation (RTCA), Inc., December 1992. Internet: www.rtca.org
- Ericson et al. 96** Ericson, T., A. Subotic, and S. Ursing: *Towards a Test Improvement Model*. EuroSTAR 1996
- Fagan 76** M.E. Fagan, *Design and Code Inspections To Reduce Errors in Program Development*. IBM Systems Journal, Vol 15, No. 3, 1976, pp. 219 - 248
- Fagan 86** M.E. Fagan, *Advances in Software Inspections*. IEEE Transactions on Software Engineering, Vol. 12, No. 7, July 1986, pp. 744 - 751
- Frankl and Weyuker 88** Frankl, P.G.; Weyuker, E.J.: *An Applicable Family of Data Flow Testing Criteria*. IEEE Trans. on Software Engineering, Vol. 14, No. 10, 1988, pp. 1483-1498

ITEA CONFIDENTIAL

- Gelperin 96** Gelperin, D.: *A Testability Model*. STAR 1996
- Grochtmann and Grimm 93** Grochtmann, M., Grimm, K.: *Classification Trees for Partition Testing*. Software Testing, Verification & Reliability, Volume 3, Number 2, Juni 1993, Wiley, pp. 63-82
- Grochtmann et al. 93** Grochtmann, M., Grimm, K., Wegener, J.: *Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor*. Euro-STAR '93 - 1st European International Conference on Software Testing Analysis and Review, 25-28 Oktober 1993, Edwardian Hotel, London, pp. 169-176
- Grochtmann et al. 95** Grochtmann, M., Wegener, J., Grimm, K.: *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE*. Eighth International Software Quality Week (QW'95), 30. Mai-2. Juni 1995, San Francisco, California, USA, Paper 4-A-4, 11 pages
- IEEE 90** IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 610.12-1990
- IEEE 98** IEEE Standard for Software Verification and Validation, Std. 1012-1998
- Jin and Offutt 98** Jin, Z.; Offutt, J.: *Coupling-based criteria for Integration Testing*. Software Testing, Verification and Reliability, Vol. 8, 1998, pp. 133-154
- Kaner 96** Kaner, C.: *Software Negligance and Testing Coverage*.
www.kaner.com/coverage.htm
- Kaner et al. 93** Cem Kaner, Jack Falk and Hung Quoc Nguyen: *Testing Computer Software*. Van Nostrand Reinhold Company, New York, 1993 (2nd edition)
- Koomen and Pol 99** Koomen, T., Pol, M.: *Test Process Improvement: a Practical Step-by-Step Guide to Structured Testing*. Addison-Wesley. See also www.iquip.com.
- Linnenkugel and Müllerburg 90** Linnenkugel, U.; Müllerburg, M.: *Test Data Selection Criteria for (Software) Integration Testing*. Proceedings of the 1st International Conference on Systems Integration, IEEE, April 1990
- Mitchell 96** Mitchell, M.: *An Introduction to Genetic Algorithms*. Cambridge, Massachusetts: MIT Press, 1996
- Mueller and Wegener 98** Mueller, F., Wegener, J.: *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints*. Proceedings of the IEEE Real-Time Technology and Applications Symposium RTAS '98, pp. 144-154, 1998
- Ntafos 88** Ntafos, S.C.: *A Comparison of some Structural Testing Strategies*. IEEE

ITEA CONFIDENTIAL

Trans. on Software Engineering, Vol. 14, No. 6, 1988, pp. 868-874

- Ostrand and Balcer 88** Ostrand, T., Balcer, M.: *The Category-Partition Method for Specifying and Generating Functional Tests*. Communications of the ACM, Volume 31, Number 6, June 1988, pp. 676-686
- Puschner and Vrchaticky 97** Puschner, P., Vrchaticky, A.: *Problems in Static Worst-Case Execution Time Analysis*. Proceedings of the 9th ITG/GI-Conference on Measurement, Modeling and Evaluation of Computational and Communication Systems, pp. 18-25, 1997
- Rapps and Weyuker 85** Rapps, S.; Weyuker, E.J.: *Selecting Software Test Data Using Data Flow Information*. IEEE Trans. on Software Engineering, Vol. 11, No. 4, 1988, pp. 367-375
- SEI 95** Software Engineering Institute, Carnegie Mellon University: *The Capability Maturity Model*. Addison-Wesley, 1995
- Törngren 98** Törngren, M.: *Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems*. In Wikander, J., and Svensson, B. (Eds): Real-Time Systems in Mechatronic Applications, Kluwer Academic Publishers, Boston, USA, 1998
- V-Model 97** www.v-modell.iabg.de/vm97.htm#ENGL
- Wegener and Grochtmann 98** Wegener, J., and Grochtmann, M.: *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing*. Real-Time Systems, 15, pp. 275-298, 1998
- Wegener and Pitschinetz 95** Wegener, J., Pitschinetz, R.: *Tessy - An Overall Unit Testing Tool*. Eighth International Software Quality Week (QW'95), 30 May – 2 June 1995, San Francisco, California, USA, pp. 3-A-3/1-14
- Wegener et al. 99** Wegener J., Sthamer H., and Pohlheim H.: *Testing the Temporal Behavior of Real-Time Tasks using Extended Evolutionary Algorithms*. EUROSTAR'99, Barcelona, Spain 1999
- Zelkowitz and Rus 01** Zelkowitz, M.V.; Rus, I.: *Understanding IV&V in a Safety Critical and Complex Evolutionary Environment: The NASA Space Shuttle Program*. Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), 12-19 May 2001, Toronto, Ontario, Canada. IEEE Computer Society, Los Alamitos, California, pp 349-357. See also www.ivv.nasa.gov.

Appendix A: Release procedure

When negotiating release dates with customers, a compromise has to be made between available time, resources/funding and required quality. In this appendix, a brief description of the available options to define the quality of a release is specified for an example project.

Assigning a 'quality level' to reviews

Defining a 'quality level' for reviews (in early V&V workflows) is possible by defining a trade-off between performing an optional cross-check (one but lowest level) and performing a mandatory inspection (highest quality level). Table 12 shows possible inspection/cross-check combinations.

Required way of reviewing / cross checking	Corresponding quality level
-/-	0
-/o	1
-/c	2
-/m	3
o/c	4
o/m	5
m/-	6

Whereas:

- 'm' :means mandatory.
- 'c': means customised.
- 'o': means optional.
- '-': means not applicable.

Note: 'o/m' means optional inspection, if not conducted then a cross-check is mandatory

Table 12: Cross-check / inspection and its corresponding quality level

Assignment of a 'quality level' to component-, integration-, system- and acceptance workflows

A 'quality level' for the workflows, other than reviews is defined in Table 13.

% of fulfilled test goals for the test	Corresponding quality level of the test results
< 50	0
50 - 65%	1
65 - 80%	2
80 - 90%	3
90 - 95%	4
95 - 99%	5
100%	6

Table 13: Percentage of fulfilled requirements for a workflow, versus corresponding quality level

Quality level in relation to the project tailoring

Two attributes have an impact on the project's defined software process:

- project type.
- project size.

Project type can either be 'business' or 'economy'. Project size can be 'small' or 'large'. This tailoring procedure directly influences the quality of the deliverables.

With this, the following quality level in relation to project tailoring can be defined as in Figure 18.

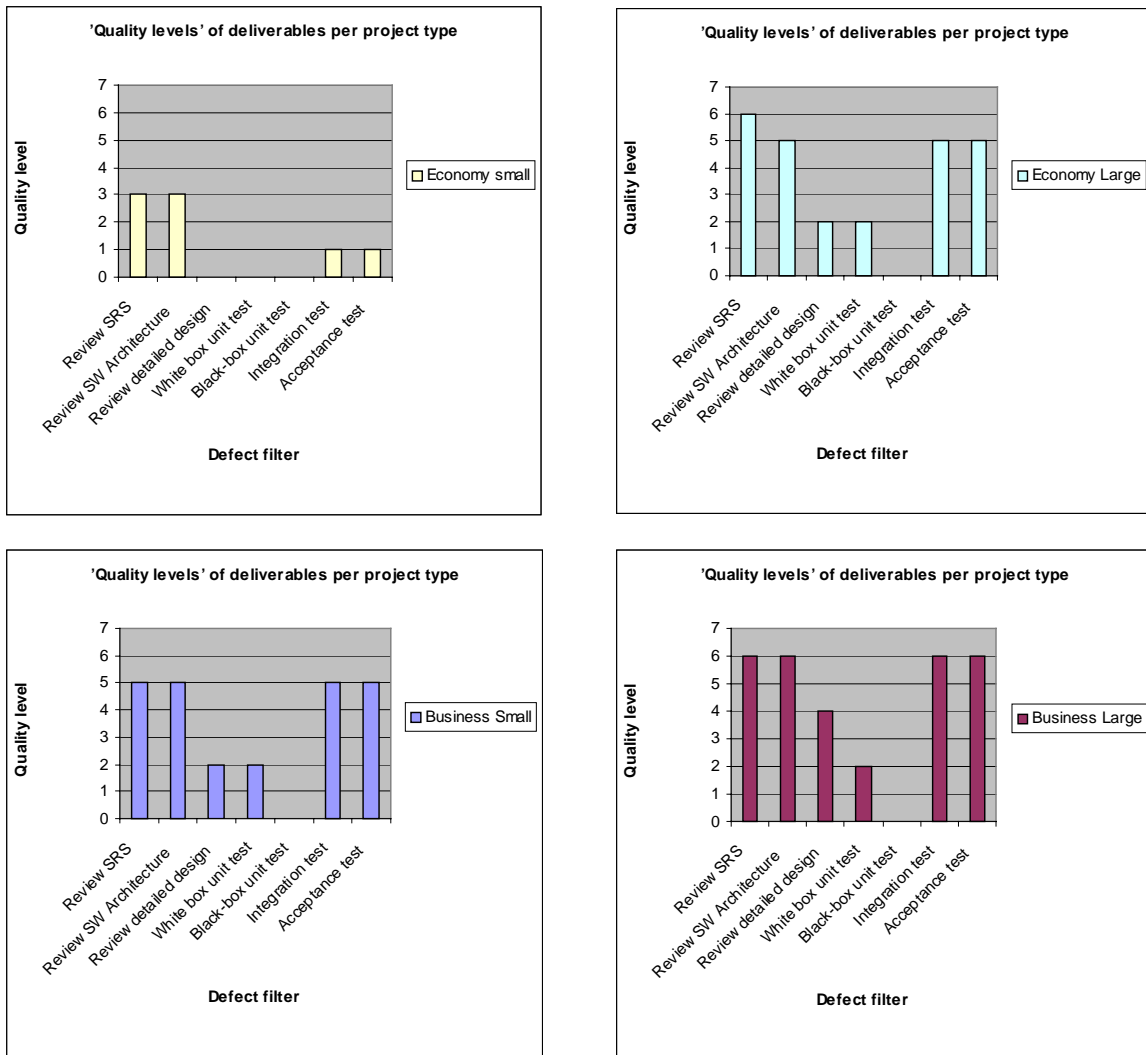


Figure 18: Quality level in relation to project tailoring

Appendix B: Embedded Software Quality Criteria

An extensive list of quality criteria is given in [ISO9126]. The TMap method defines a subset of these, especially suited for testing (more information about TMap at www.iquip.com). These quality criteria are divided into static and dynamic ones. The dynamic criteria are tested on the running software. The static criteria can be tested without running the software.

Dynamic criteria are:

- Continuity: the assurance that the system stays operational, is insensitive to disturbances and that it can easily be restarted after a serious problem (reliability, availability, robustness and error recovery).
- Economy: economic use of system resources (CPU load, memory, etc).
- Functionality: the correctness and completeness according to the system specification.
- Performance: the time it takes to perform certain actions.
- User-friendliness: the ease of learning to operate the system for an end-user (possibly split up in first-time and experienced users).
- Usability: does the system fulfil the user needs (validity) ?
- Security: protection against unauthorised use of data.

Static criteria are:

- Connectivity: the ability to connect the system with other systems.
- Flexibility: the ability for the user to extend or change the functionality of the system without changing the code.
- Maintainability: the ability to change the system to conform new requirements or to solve new problems.
- Manageability: the ease to install the system and to make it operational.
- Portability: the diversity of different hardware and software environments where the system can be used.
- Re-usability: the ability to reuse (parts of) the code.
- Testability: the ease with which the functionality and performance of the system can be tested.