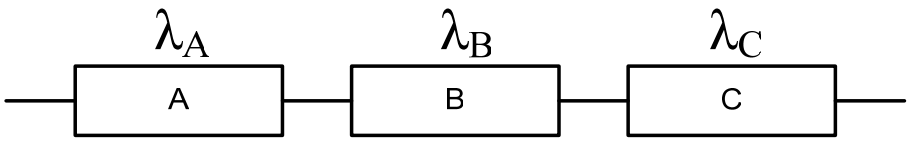
Methods for Verification and Validation of Safety

Lars Strandén, Andreas Söderberg,
Jan Jacobson, Josef Nilsson

SP Technical Research Institute of Sweden

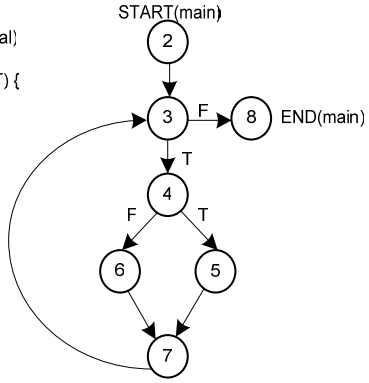


Example program

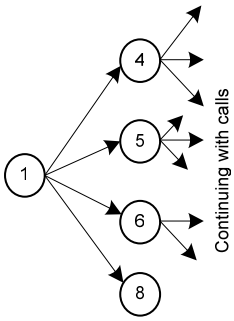
```
1 main() {
2 int a = Mean_ADC0() // Analog value (initial)

3   while(a > MIN_LIMIT && a < MAX_LIMIT) {
4     if(PortA() & POS)
5         Run_ApplON();
      else
6         Run_ ApplOFF();
7     a = Mean_ADC0();
    } //END while
8 Report(ALARM);
  } //END main()
```

Corresponding flow graph



Corresponding call graph



Electronics

SP Report 2007:14

Methods for Verification and Validation of Safety

Lars Strandén, Andreas Söderberg,
Jan Jacobson, Josef Nilsson

Abstract

Methods for Verification and Validation of Safety

Functional safety for automotive embedded systems is a topic of increasing importance. The report describes methods and techniques for verification and validation of safety. Application examples are given.

References are given to standards for functional safety developed by the International Electrotechnical Commission (IEC) and the International Standardisation Organisation (ISO).

The report is based on the work of the AutoVal project of the IVSS (Intelligent Vehcle Safety Systems) research programme.

Key words: automotive electronics, dependable systems, safety, reliability, validation

SP Sveriges Tekniska Forskningsinstitut
SP Technical Research Institute of Sweden

SP Report 2007:14
ISBN 978-91-85533-82-4
ISSN 0284-5172
Borås 2007

Contents

Preface

New safety functions and the increased complexity of vehicle electronics enhance the need to demonstrate dependability. Vehicle manufacturers and suppliers must be able to present a safety argument for the dependability of the product, correct safety requirements and suitable development methodology.

The objective of the AutoVal project is to develop a methodology for safety validation of a safety-related function (or safety-related subsystem) of a vehicle. The validation shall produce results which can be used either as a basis for a whole vehicle type approval driven by legislation, or for supporting dependability claims according to the guidelines of the manufacturer.

The AutoVal project is a part of the IVSS (Intelligent Vehicle Safety Systems) research programme. IVSS is systems and smart technologies to reduce fatalities and severe injuries, this can be done by crash avoidance, injury prevention, mitigation and upgrading of handling, stability and crash-worthiness of cars and commercial vehicles enabled by modern IT. Both infrastructure dependent and vehicle autonomous systems are included as are systems for improved safety for unprotected road – users. The core technologies of IVSS are:

- Driver support & human – machine interface (HMI) systems
- Communication platforms – external / internal to the vehicles
- Sensor – rich embedded systems
- Intelligent road infrastructure & telematics
- Crashworthiness, bio-mechanics and design of vehicles for crash-avoidance and injury prevention.
- Dependable systems
- Vehicle dynamic safety systems

Partners of the AutoVal project are Haldex, QRtech, Saab Automobile, SP Technical Research Institute of Sweden and Volvo AB. Following researchers and engineers have participated in AutoVal:

Mr Henrik Aidnell, Saab Automobile
Mrs Sabine Alexandersson, Haldex Brake Products
Mr Joacim Bergman, QRtech
Mr Per-Olof Brandt, Volvo
Mr Robert Hammarström, SP
Mr Jan Jacobson, SP (project manager)
Dr Lars-Åke Johansson, QRtech
Dr Henrik Lönn, Volvo
Mr Carl Nalin, Volvo
Mr Anders Nilsson, Haldex Brake Products
Dr Magnus Gäfvert, Haldex Brake Products
Mr Josef Nilsson, SP
Mr Lars Strandén, SP
Mr Jan-Inge Svensson, Volvo
Mr Andreas Söderberg, SP

www.ivss.se www.vinnova.se www.vv.se

Summary

Many of the functions of a modern car or truck are realised using programmable electronic systems. The systems are embedded in all parts of the vehicle and can perform safety-related functions as engine control and stability control. Verification and validation are needed all through the development life cycle to demonstrate that safety in reached.

The methods and techniques listed in the report are grouped as

- review
- models
- analysis
- dynamic methods
- methods regarding formality
- development methods

The aim of every method is explained, a comprehensive description is given and references are stated. Examples are given how the methods can be applied.

There is no single method that can provide all answers to the safety questions raised. However, there are lots of different techniques, methods and tools to support verification and validation. The validation methods have to be combined in order to achieve sufficient quality.

1 Verification and validation

Many of the functions of a modern car or truck are realised using programmable electronic systems. The systems are embedded in all parts of the vehicle and can perform safety-related functions as engine control and stability control. Functional safety is part of the overall safety that depends on a system or equipment operating correctly in response to its inputs.

Verification and validation is needed all through the development life cycle to demonstrate that safety in reached. Verification is defined as confirmation by examination and provision of objective evidence that the requirements have been fulfilled [IEC 61508-4, clause 3.8.1] But validation is understood as the confirmation and provision of objective evidence that the particular requirements for a specific intended use are fulfilled [IEC 61508-4, clause 3.8.2]

It is not trivial to show that a complex system actually is suitable for its intended use. Careful risk analysis must be made already from the beginning of the development. Every step of the realisation shall be confirmed by verification. At the end of the development, there shall be an overall safety validation to demonstrate functional safety.

1.1 The validation plan

A set of methods and techniques (a "tool box") is needed. There is no single method that can provide all answers to the safety questions raised. However, there are lots of different techniques, methods and tools to support verification and validation. A limited set of tools has to be recommended as "the contents of the tool box" used for validation.

The methods and techniques listed in the report are grouped as

- review
- models
- analysis
- dynamic methods
- methods regarding formality
- development methods

The validation methods have to be combined together in a validation plan. The plan shall list requirements and validation methods. A validation procedure is useless without solid safety requirements. Both the safety functions and a measure of their performance (or integrity) must be specified. Activities performed earlier during the verification in earlier phases of the development work may also serve as evidence for adequate safety.

1.2 Selection of validation methods

Different validation methods are applicable at different stages of the development life cycle. The overall safety validation is, according to the overall safety life cycle, conducted after the realisation and installation of the safety-related system. There are also other phases of the life-cycle that are important for the validation. Techniques and measures applied in other phases than "the overall safety validation phase" will contribute to the safety validation.

The IEC 61508 standard recommends certain methods to be included in the overall safety validation. The higher safety integrity levels tend to require a more stringent validation. The same principle is used in the draft standard ISO 26262 where methods and techniques are recommended in a similar way.

A large number of techniques and measures exist for verification and validation. It will be necessary to recommend a limited number for use in automotive applications. But it cannot be expected to find a set of techniques and measures applicable for all automotive systems. The different realisation techniques and the different types of systems will require several available methods.

Current practise of safety validation was compared among the AutoVal partners. The most used methods were selected, and some additional validation methods were included. (Tables 1.a and table 1b). However, the specification of validation methods at different safety integrity levels (SILs, ASILs) need further work and is not yet included in this report.

Table 1a: Selection of methods for the concept phase

Activity	**Phase in the overall safety lifecycle (IEC 61508)**	**Phase in the overall safety lifecycle (ISO 26262)**	**Method**
System modelling/definition	(2), 3	Item definition	Passport diagram Modelling FMEA (design on system level) FMEA (production)
Hazard Identification and classification	(2), 3	Hazard analysis and risk assessment	Warranty Data Assessment What if? Analysis Preliminary Hazard Analysis Functional FMEA HAZOP Risk Graph Controllability Check lists Brainstorming Fault Tree Analysis Legal demands System engineering tool
Risk analysis	3	Hazard analysis and risk assessment	Risk Class Matrix Markov modelling Fault Tree Analysis Security D-FMEA
Specification of safety requirements	4, 5	Functional safety concept	What Causes? Analysis Safety requirements allocation

Table 1b: Selection of methods for the product development phase

Activity	**Phase in the overall safety lifecycle (IEC 61508)**	**Phase in the overall safety lifecycle (ISO 26262)**	**Method**
Hardware development	9	Development of system	Company-internal methods for hardware analysis Hardware component FMEA, FMECA Reliability Analysis Design review Environmental stress test EMC test Endurance test for complete system Statistical testing
Software development	9	Software development	MISRA Guidelines V-model Design review SW-module testing Simulation (software-in-the-loop) Static/Dynamic Code Analysis Modelling Formal methods
Software+ECU validation	13	Development of system	Software+ECU validation for one node according to specification from external supplier. Software+ECU validation for all nodes according to specification from external/internal suppliers.
Software/Hardware validation	13	Development of system	System test in complete vehicle Field tests Test rig FMEA FTA Simulation (MIL, SIL) HIL simulations for one node Black-box testing
Overall safety validation	(2), 3	Product release	Safety case

2 Reviews

2.1 General

In this report *review* is considered the most general term for manually analyzing available documents. Since interpretation of review and related terms differs in the literature a further description is needed in this document. According to standard IEEE 610.12-1990 there are the following terms:

- *Review* - A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include code reviews, design reviews, formal qualification reviews, and requirements reviews, test readiness reviews.
- *Audit* - An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements or other criteria.
- *Inspection* - A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems. Types include code inspections; design inspections.
- *Walkthrough* - A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code and the participants ask questions and make comments about possible errors, violation of development standards and other problems.

Problems shall not be solved during a review only pointed out. The examination should (if possible) be carried out by persons that were not involved in the creation of the reviewed document. The required degree of independence could e.g. be determined by safety integrity level demands of the system as defined in standard IEC 61508. Usually test cases (scenarios), checklists and guidelines are used together with *reading,* for the individual. Reading is defined by IEEE 610.12-1990 as:

- a technique that is used individually in order to analyze a product or a set of products. Some concrete instructions are given to the reader on how to read or what to look for in a document.

Audit is a review with a specific purpose, to let management understand the project status, and does not imply any specific techniques.

There are some general aspects related to the use of reviews:

- Results shall not be used for person appraisal.
- Feedback from early reviews will improve later ones.
- Training of personnel improves review quality.
- Rotation of review roles improves quality.
- Active support and acceptance is needed from management and those involved in reviews since extra resources are needed early but could be removed in later phases of development.

In this report the focus is on techniques and not on when and where to apply them. The two main techniques considered here are *Inspection* and *Walkthrough*. The distinction between them is that the author/designer has the initiative (is active) during *Walkthrough* while the inspector is active during *Inspection*. Thus the scope and focus will be different; the author looking from his perspective (creator or server) and the inspector from his (user or client). Generally *Walkthrough* is less formal than *Inspection*. *Walkthrough* and *Inspection* are general techniques in which various qualities of any kind of document are assessed. We also have *Informal review* as an alternative with a more ”brain-storm” like scope with no or very limited requirements on formality. Here, it is not decided who has the initiative. Generally, the more formal a technique is the easier it is to repeat it and to maintain the same quality.

We next have to define what we mean by *formal*. Generally this means that there is a well defined (repeatable) procedure and organisation. According to www.processimpact.com/articles/two_eyes.html there are the following characteristics for formal handling:

- Defined objectives
- Participation by a trained team
- Leadership by a trained moderator
- Specific participant roles and responsibilities
- A documented procedure for conducting the review
- Reporting of results to management
- Explicit entry and exit criteria for each work product
- Tracking defects to closure
- Recording process and quality data to improve both the review process and the software development process.

From these one can think of less formal handling by removing one or more of the characteristics.

The performance of reviews could be improved in different ways. One example is to use *Sample-driven inspection* as described in http://www.cas.mcmaster.ca/wise/wise01/ThelinPeterssonWohlin.pdf. Another example is to use tool support for handling reviews see e.g. http://www.goldpractices.com/practices/fi/index.php.

We also have to classify the techniques with respect to the number of persons involved:

- Two persons – the author and the reviewer
- A (small) group – the author, reviewer and possibly some other roles

As a summary we get the following principle technique cases:

- Inspection – two persons
- Inspection – group
- Walkthrough – two persons
- Walkthrough – group
- Informal review – two persons
- Informal review – group

When using the techniques the borders between Inspection, Walkthrough and Informal review may not always be clear. Notice that the classification does not depend on the artefact being reviewed; not the contents, scope, level of detail, maturity or extent. The use of inspection, walkthrough and informal review can be applied for different artefacts and under different conditions. This includes artefacts of development process, development (according to process), operation and maintenance. Some example artefacts are:

- Specifications (function, design, architecture tec.)
- Descriptions (product, process etc.)
- Implementation (software, hardware etc.)
- Operation documentation (operation manual, service information etc.)
- Project related documents (plans, reports, configuration, delivery summary etc.)

2.2 Means

2.2.1 General

The means described below are *checklist*, *reading* and *scenarios*. Checklist can be used for review, audit, inspection and walkthrough. Checklist could also be used for reading and scenarios. In any case reading is necessary for studying artefacts. Scenarios could be used for reading, review, audit, inspection and walkthrough.

2.2.2 Checklist

Aim: To draw attention to, and manage critical appraisal of, all important aspects of the system.

Description: A checklist is a set of questions to be answered by the person performing the check. Checklists can be used anywhere, however, it is important to keep the checklist short and focussed and to formulate the questions so they cannot be answered routinely without reflection. One example could be to change "Has property xx been evaluated?" to "What are the proofs of the evaluation of property xx?". A principal aspect is how general/specific a checklist shall be i.e. what is the scope addressed by the checklist? For example, checklists may contain questions which are applicable to many types of system. As a result, there may well be questions in the checklist being used which are not relevant to the system being dealt with and which should be ignored. Correspondingly there may be a need, for a particular system, to supplement the standard checklist with questions specifically addressing the system being dealt with. In any case it should be clear that the use of checklists depends critically on the expertise and judgement of the persons creating and applying the checklist. As a result, the decisions taken with regard to the checklists should be fully documented and justified. The objective is to ensure that the application of the checklists can be reviewed and that the same results will be achieved unless different criteria are used. To a certain extent checklists are dynamic and should be evaluated regularly. Thus there could also be a need for creating a meta-checklist i.e. a checklist for how to handle checklists.

References:

- IEC 61508-7 B.2.5

2.2.3 Reading

Aim: To perform a focussed study of documents.

Description: Before reading starts the reader is informed what is important for him/her to focus on. This concerns both specific technical aspects and the role he/she is assigned. Some commonly used principles are given below.

- Ad-hoc - no systematic guidance is provided
- Checklist based reading – the important aspects are listed in checklists
- Scenario-based reading - a scenario, encapsulated in a set of questions or a detailed description is used. This is applicable for Defect-based reading, Function Point-based reading, and Perspective-based reading (see reference).
- Reading by hierarchies - the reader develops a description of the individual low-level items. This is repeated using higher levels until the reader considers the top level of the entire artefact (bottom up). The opposite could also be used; starting from top level going to lower levels successively.

References:

- http://www.goldpractices.com/practices/fi/index.php

2.2.4 Scenarios

Aim: To perform a focussed study of documents by considering a concrete behaviour.

Description: A scenario describes a functional behaviour of a system and normally includes a group of more specific scenarios. A scenario uses inputs, performs actions possibly in a sequence and generates outputs. Also pre-conditions and post-conditions could be stated. The advantage of scenarios is that there is a focus on the use of the system and thus everyone can understand and appreciate the scenarios immediately. The use of scenarios could be accompanied by checklists for checking the different steps of the scenario sequence. One example use is to have checklists for checking *properties* of the system.

Example: A principal scenario template is used (many other exists).

Name: Money withdrawal from ATM (Automatic Teller Machine)
Pre-conditions: Money on account
Sequence: Enter credit card
Enter personal code – correct value
Specify withdrawal
Specify amount – money exists on account
Take money
Pre-conditions: Less money on account
Money in wallet

Alternatives and erroneous situations could also be included e.g. if money does not exist or if incorrect code has been entered three times etc.

References:

- http://www.uml.org/ (scenarios are included in UML)

2.3 Walkthrough

In the literature there are many different interpretations what walkthrough really is. Generally a walkthrough is considered less formal than an inspection and this is probably the normal case. However, in this report we do not include level of formality in the definition of walkthrough but instead only require that the author is the active part. Thus the walkthrough mainly serves the needs of the author. The effect is that there is a bias towards what the author considers important and the overall quality of the results might be lower than for an inspection. On the other hand the detailed considerations and motivations could be made visible in a better way using walkthroughs. Further, there could be a more focussed and cost-effective review if the author concentrates on problematic issues. A walkthrough will also reflect the functional behaviour of the application better than an inspection and could be a better alternative e.g. when considering HMI aspects together with a user.

In this report, walkthrough could be used for the same cases as inspection as long as the initiative is changed from inspector to author. In practise, since doubling review efforts will not be economically feasible, walkthroughs will normally be performed before related inspections with less people involved and with less formality. This implies that the walkthrough could be made closer in time to the development of the artefact since less preparation is needed.

2.4 Inspection

2.4.1 Two-person inspection

Aim: To perform a cost-effective check.

Description: The idea is to have a minimal inspection with the author and an independent person going through the artefact produced by the author. Even though not made with optimal quality there are a number of advantages:

- Could be made with minimum preparations i.e. with no real organisation efforts.
- Could be made with minimum delay i.e. when the author still remembers all decisions and motivations.
- Could be made with minimum resources.

This method is suitable for not critical documents.

References:

- http://www.tol.oulu.fi/~tervo/Experiences.pdf

2.4.2 Fagan inspection

Aim: To reveal mistakes and faults by using a well defined procedure.

Description: Even though stated back in 1976 the applicability of Fagan inspection remains today. The original paper focussed on software development but in principle any kind of artefact could be considered (with some obvious modifications of roles and tasks). The description below follows software development aspects but without considering the corresponding process as such. For inspection a group of persons is defined with the following roles (one person assumed for each role):

- Moderator – the coach of the group and the most important role (requires special training)
- Designer – the creator of the software design
- Coder/Implementor – the creator of the implementation
- Tester – the creator/executer of tests

Thus four persons are proposed. If a person has more than one role he/she shall take the first suitable role in the list. Other experienced persons then have to take the vacant roles. It is suggested that two hour meeting is maximum but two such meeting is allowed per day. The list below shows the associated steps of the group:

- Planning – arranging meetings and participants and checking maturity of material
- Overview – the designer presents the scope
- Preparation – each participant studies the available material in advance with focus on common error types
- Inspection – the coder (normally) describes the implementation of the design. The focus is then to find errors. How to correct them is not considered.
- Rework – errors are corrected after inspection meeting(s)
- Follow-up – the moderator checks that error s have been correctly handled. The moderator also decides if a reinspection is necessary.

Errors are classified according to severity by the moderator. Since training of involved persons is beneficial one way to do this is by making a preliminary inspection of some other representative design and source code. The result could generate a specification (or guideline) for how to perform inspections and it could be improved as inspections continue. Personal training is needed for management, moderators and participants.

References:

- IEC 61508-7 C.5.15

- Michael E. Fagan: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal 15(3): 182-211 (1976)
- Michael E. Fagan: Advances in Software Inspections, IEEE Transactions On Software Engineering, July 1986 http://www.mfagan.com/resource_frame.html

2.4.3 Yourdon's structured walkthrough

Aim: To reveal mistakes and faults by using a well defined procedure.

Description: This method is named walkthrough but in this report is considered a kind of inspection since the author of the artefact does not have the initiative. The method is based on Fagan inspection. The following roles are defined:

- Coordinator - the moderator (administrator) of the meeting.
- Scribe (or secretary) - keeps notes on everyone's comments.
- Presenter - (generally the author) describes the artefact.
- Reviewer – searches for defects of the artefact
- Maintenance Oracle – (or QA inspector) checks that the artefact is readable and maintainable and up to company standards.
- Standards Bearer – checks that standards the team agreed on beforehand are maintained.
- User Representative – checks the artefact from the user perspective

Process steps are:

- Organization
- Preparation
- Walkthrough
- Rework

with the same meaning as in Fagan inspection. The method is less formal and rigorous than formal inspections.

References:

- http://www.hep.wisc.edu/~jnb/structured_walkthroughs.html
- Macdonald, F. and Miller, J., 1995. Modelling Software Inspection Methods for the Application of Tool Support. Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.

2.4.4 Other Fagan based variants

Aim: To reveal mistakes and faults by using a well defined procedure.

Description: For several Fagan based variants different names are used for the same steps. Further, the work may be organized somewhat differently e.g. regarding focus and meeting effectiveness. Below, only method specific aspects are commented on.

The *NASA standard 2203-93* is close to Fagan inspection but includes some additional aspects. The roles are different with the following main differences:

- Inspector - every active member of the inspection team is considered an inspector in addition to any other assigned role.
- Reader - the reader (often the author) is responsible for guiding the inspection team through the artefact during the inspection meeting
- Recorder - the recorder is responsible for recording information during the meeting
- Author - the author is the originator of the artefact
- Gallery – passive group listening to the discussion but does not take active part

The group consists of four to seven persons with a minimum of three. Regarding the steps, and comparing with Fagan inspection, Overview and Follow-up steps are not explicit.

In *Humphrey's Inspection Process* reviewers are asked to find and document defects before the inspection meeting (at Preparation). These documents are merged and analyzed before the meeting and at the meeting they are gone through.

In *Gilb Inspection* an additional task is a brainstorming session in which bug fixes are suggested. The detection of defects is close to the handling in *Humphrey's Inspection Process* however analysis of defects does not take place before the inspection meeting. The focus is to log as many defects as possible. Explicit steps in the process are checking against entry and exit criteria.

References:

- http://audsplace.com/Classes/CSE360/Notes+Examples/Week03/FormalReviews.html
- http://www.goldpractices.com/practices/fi/index.php
- Macdonald, F. and Miller, J., 1995. Modelling Software Inspection Methods for the Application of Tool Support. Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.
- Tanveer Hussain Awan, 2003. Sources of variations in software inspections: An empirical and explorative study. TDT4735 Software Engineering, Specialization, Norwegian University of Science and Technology

2.4.5 Checklist based inspection

Aim: To reveal mistakes and faults by using experts.

Description: In *Phased Inspection* there are a number of phases regarding specific aspects. For each there are corresponding checklists. There are tow types of phases *single-inspector* where only one person is involved and *multiple-inspector* with more than one person involved. Checklist are answered and for the *multiple-inspector* case compared between different inspectors at a *reconciliation* meeting. The idea of this method is to make direct use of expertise.

References:

- Macdonald, F. and Miller, J., 1995. Modelling Software Inspection Methods for the Application of Tool Support. Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.
- Tanveer Hussain Awan, 2003. Sources of variations in software inspections: An empirical and explorative study. TDT4735 Software Engineering, Specialization, Norwegian University of Science and Technology

2.4.6 Active Design Review

Aim: To reveal mistakes and faults by using a well defined procedure.

Description: In *Active Design Review (ADR)* instead of one main inspection review there are several smaller ones each with its specific focus concerning the design. There could be different persons for each review and the following roles are defined:

- Designer – the creator of the artefact
- Reviewer – checks for defects according to the focus of the review

Process steps are:

- Overview – the designer presents the artefact and administration issues decided
- Review - each individual reviewer completes questionnaire, some reviewers have local scope and some have overall scope
- Discussion – meetings are held with each reviewer individually and the designer(s). When all agree the review is complete.

References:

- Macdonald, F. and Miller, J., 1995. Modelling Software Inspection Methods for the Application of Tool Support. Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.
- Tanveer Hussain Awan, 2003. Sources of variations in software inspections: An empirical and explorative study. TDT4735 Software Engineering, Specialization, Norwegian University of Science and Technology

2.4.7 N-fold inspection

Aim: To reveal mistakes and faults by using a well defined procedure.

Description: In *N-Fold inspections* there are n independent teams conducting inspections of the same artefact and supervised by one moderator (coordinator) for the teams together. The teams do not have to use the same inspection method, instead it is an advantage if they do use different methods. The following steps are needed for the overall process:

- Planning – administration of teams
- Overview – common for all teams
- Inspection - performed in parallel for all teams
- Collation - aspects are merged

References:

- Macdonald, F. and Miller, J., 1995. Modelling Software Inspection Methods for the Application of Tool Support. Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.
- Tanveer Hussain Awan, 2003. Sources of variations in software inspections: An empirical and explorative study. TDT4735 Software Engineering, Specialization, Norwegian University of Science and Technology

2.4.8 Meeting-less inspection

Aim: To reveal mistakes and faults by using a well defined procedure.

Description: An example method is *Formal Technical Asynchronous review method (FTArm)* where there is no inspection meeting; instead people comment the artefact directly online. The following roles are defined:

- Moderator - the administrator of the meeting.
- Producer – creator of artefact
- Reviewer - searches for defects of the artefact

Process steps are:

- Setup – preparation of inspection
- Orientation – the same as in Fagan inspection
- Private review – the inspector checks and gives comments in private
- Public review – all private comments become public and each inspector can study the results of the other and vote concerning defect status
- Consolidation – the moderator analyses the results and decides if a group meeting is anyhow needed
- Group review meeting - for resolving remaining problems
- Conclusion – the moderator produces a final report

References:

- Macdonald, F. and Miller, J., 1995. Modelling Software Inspection Methods for the Application of Tool Support. Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.
- Tanveer Hussain Awan, 2003. Sources of variations in software inspections: An empirical and explorative study. TDT4735 Software Engineering, Specialization, Norwegian University of Science and Technology

2.5 Informal review

Aim: To solve immediate problems with no preparations.

Description: In this case a meeting is held in order to discuss and possibly handle a problem. The meeting can be seen as a kind of brain-storm and does not require any formalities. Thus ideas can be tested but the quality of the review cannot be judged and cannot be repeated in the same way later on. The advantage of the informal review for surveys is that it can take place immediately and can result in rapid actions.

Reference:

- http://www.cs.utexas.edu/users/almstrum/classes/cs373/fall97/review-guidelines.html

3 Models

3.1 Introduction

The definition of *model* used in this report is defined by the following:

1. A model represents a real or imagined system.
2. A model considers the significant aspects, from a specific perspective, and ignores the others.
3. A model is developed *before* analysis (or use) takes place and not as a result of it.

For example, FTA is not considered a model due to condition 3, while Markov modelling fulfils all three. A model can be used for investigation of properties of the system and prediction of future outcomes. A model could be related to *behaviour* or to *structure* and both types are exemplified below. The motivation for gathering models in a separate chapter is that they could be used for different verification and validation techniques. Thus the creation of models could to a certain, but limited, extent be considered independent of the techniques being used. For example, models defined by UML could be used for reachability analysis (state models) and functionality analysis (sequence diagrams).

For object-oriented development UML is the dominant notation. Since UML defines several models they are separated according to their purposes instead of belonging to a single UML chapter.

For an overview of models see http://www-ihs.theoinf.tu-ilmenau.de/lehre/ihs/IHS1-behavioral-models-and-languages.pdf and http://engr.smu.edu/cse/7312/spring04/7312-unit6-broihier.ppt.

3.2 Reliability block modelling

Aim: To model, in a diagrammatic form, the successful operation of a system or a task.

Description: Reliability block modelling (RBD Reliability Block diagram) takes the view of a *functioning* system i.e. making it possible to analyse the probability of the system operating correctly. States are not included. The idea is to describe the system from a reliability perspective using blocks and connections (arcs) between them. There could be more than one incoming arc to a block and more than one outgoing arc from a block. A block can be seen as a switch:

- closed – for an operating block
- open – for a non-operating block

Blocks can be connected in series or in parallel and an operating system exists when there is a closed connection from start to end. Blocks are assumed to fail independent of each other and thus special considerations have to be taken when defining blocks. Thus, if there is a common cause failure of two blocks a new block has to define containing the two blocks. A block could contain functionality implemented in software, hardware, mechanics etc and could reflect any size of functionality behind. Normally a block represents a physical or logical component but it is also possible to let a block represent a failure mode or event thus extending the expressiveness of RBD. If Pi is the probability of a block i operating then

$$\prod_{1}^{n} Pi$$

is the probability of n blocks connected in series are operating (all blocks have to be operating) and

$$1 - \prod_{1}^{n} (1 - Pi)$$

is the probability of n blocks connected in parallel are operating (at least one block is operating). Normally there will be a mixture of blocks connected in series and in parallel but also k-out-of-n blocks and majority voting can be modelled. A hierarchical structure is supported; a block could be expanded into sub blocks. More complicated cases could also exist e.g. a connection from one block within a group of parallel blocks connected to another block within another group of parallel blocks.

Such cases should be avoided in order to make use of RBD less complex and possible principle solutions are:

- to redefine fault model e.g. what should be considered a common fault
- to redesign the system
- to create new, and possibly bigger, blocks
- to duplicate block
- to manually create equations for the complex cases (if not too many)

An alternative is to let the complex cases remain and use tools for analysis.

Example: The picture below shows a system with two blocks in series where each consists of two redundant blocks. Reliability is here denoted R which, in an actual case, could be a distribution, probability (i.e. distribution at a specific point of time) etc.

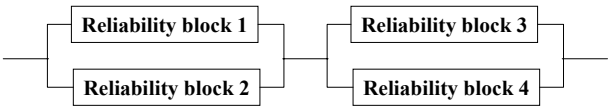


Figure 1: RBD example

For the whole system we get $R_{1,2} \bullet R_{3,4} = (1-(1-R_1) \bullet (1-R_2)) \bullet (1-(1-R_3) \bullet (1-R_4))$. If all R_i are the same we get the total reliability as $(2R-R^2)^2$.

References:

- IEC 61508-7 C.6.5
- http://www.mfn.unipmn.it/~bobbio/IFOA/ifoa-rbd.ppt
- http://www.weibull.com/basics/fault-tree/ (some complex RBDs are given in examples)

3.3 Finite state

Aim: To model the system as states with transitions.

Description: Even though Markov modelling considers states it will be handled in a separate chapter below. This follows the general classification in the literature. Finite state representation has many related names e.g.

- Statechart which is a language that support hierarchies of states (developed by David Harel). Statechart has a strong industrial support (many applications).
- Automaton
- State representation in UML (object-oriented version of Statechart and with some differences in the details)
- Finite state machine
- State machine
- Timed state machine
- Non-deterministic finite state machine
- Finite state automaton
- Hierarchical state machine
- State transition diagram
- Finite state automaton
- Mealy automaton
- Moore automaton
- ROOM (Real-time Object-Oriented Modeling) includes parallel processes, communication and state representation of behaviour. ROOM is included in later versions of UML.

These are variants (or denote the same idea). The principle is to define states and transitions between them. For each state only a subset of conditions (or inputs) has to be considered which simplifies

evaluation. Different representations are used and by using tools simulation are often possible. Since the representation is graphical it enables a common understanding of the behaviour. In any realistic system a hierarchical representation is necessary i.e. the support of expanding a state into sub-states. An important aspect is how the representation handles communication and concurrency (i.e. real-time aspects) e.g. are all states handled as having their own processors i.e. true parallelism, is communication synchronous or asynchronous etc. Thus there is a mapping issue from states to run-time constructs.

Notice that the formal verification method *model checking* uses state representation for proofs.

Example: This example is partly from http://www.ics.uci.edu/~rgupta/ics212/w2002/models.pdf and describes the handling of alarm for vehicle seat belt. The picture shows the corresponding states and transitions. If transition condition is not fulfilled one remains in the current state.

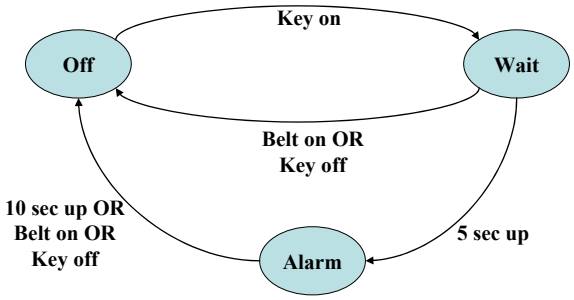


Figure 2: State machine example

The behaviour is described by:

- When key is turned on there is an acceptable delay of 5 seconds before the alarm is activated if belt is not on during the period.
- If key is turned off there is no alarm.
- If belt is on there is nor alarm.
- The alarm can be active at a maximum of 10 seconds.

Notice that opening the belt will not cause activation of alarm if key is already on since a prerequisite for activating the alarm is the *event* Key on and not the *state* Key on.

References:

- IEC 61508-7 B.2.3.2
- http://citeseer.ist.psu.edu/cache/papers/cs/26581/http:zSzzSzwww.wisdom.weizmann.ac.ilzSz~harelzSzStatecharts87.pdf/harel87statecharts.pdf
- Harel, David, and Michal Politi, Modeling Reactive Systems with Statecharts, The STATEMATE Approach. McGraw-Hill, 1998, ISBN 0-07-026205-5
- ROOM Selic, Bran, Garth Gullekson, and Paul. T. Ward, Real-Time Object Oriented Modeling, John Wiley & Sons, 1994, ISBN0-471-59917-4

3.4 Queuing model

Aim: To model the system or parts of it as queues.

Description: For certain applications the behaviour can be described by items arriving independently to places where the items can be served. In such cases a queuing model could be used. Queuing model has a strong mathematical underpinning. The characteristics of queuing models are:

- Customers and servers
- Server and system capacities
- Dynamic aspects (if customers and servers can be created and deleted, if conditions change etc.)
- Number of customers and servers (1, n, infinite)
- Arrival process for customers (Markovian, Poisson, arbitrary, deterministic etc.)
- Queuing behaviour (leave, stay, move)
- Queue ordering (FIFO, LIFO, priority etc.)
- Service time distribution (Markovian, Poisson, arbitrary, deterministic etc.)

Complex systems can be modelled by creating structures of queues connected as a queuing network. Mathematical analysis is possible for small systems but simulation is needed for larger ones. Some examples of use are processes to be scheduled, communication through network, faulty units to be handled i.e. any case where there are limited resources and varying number of clients needing them.

Example: For queuing models there is a standard notation according to Kendall, however, there are some different versions also (with extra parameters added) but the basic notation is X/Y/m where X defines interarrival process, Y defines service distribution and m is number of servers. The most commonly used queuing model is M/M/1, i.e. Poisson arrival process / exponential service time / one server, which is shown in the figure below.

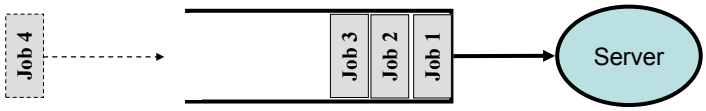


Figure 3: Queuing example

Thus jobs arrive according to Poisson distribution, are buffered and handled by the single Server according to an exponential time distribution.

References:

- http://www.sce.carleton.ca/courses/94501/s02/queuingModels.pdf
- http://www.cs.utexas.edu/users/browne/cs380ns2003/Papers/SimpleQueuingModelspdf.pdf

3.5 Concurrent processes

Aim: To model the system as a number of communicating concurrent processes.

Description: Process algebra concerns these aspects but are treated as formal methods instead and described below. Here only Petri net will be considered. Petri net has also an extensive formal support but the graphical structure motivates its place here as a model. A Petri net is a network of *places* and *transitions* alternately, i.e. a place is followed by a transition, with connection between them. A place can be *marked* or *unmarked* and an initial *marking* is defined i.e. the places having a token (or more). A transition is enabled when all the input places to it are marked. When enabled, it is permitted to fire. If it fires, the input places to the transition become unmarked, and each output place from the transition is marked instead. The Petri net model can describe sequences, non-determinism and concurrency and can be extended to allow for timing aspects.

Example: The example below shows a producer-consumer process with buffer where a grey denotes a marked place. The left hand side is the producer and the right hand side is the consumer.

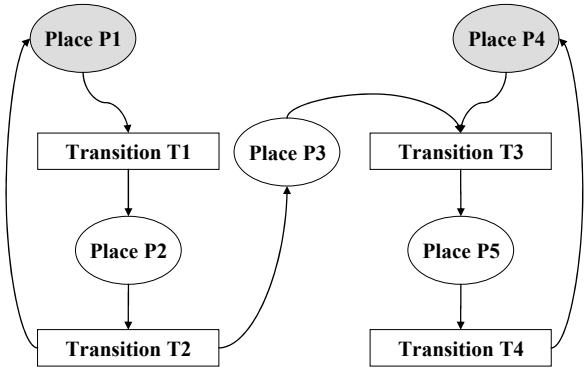


Figure 4: Petri net example

The following things happen:

1. T1 fires since P1 contains a token (T3 cannot fire since there is no token in P3). P1 looses its token.
2. P2 gets a token.
3. T2 fires and a token is placed in P1 and P3. P2 looses its token.
4. T3 fires since P3 and P4 contain tokens.
5. P5 gets a token. P3 and P4 loose their tokens.
6. T4 fires and a token is placed in P4. P5 looses its token.
7. etc

Notice that some action are made in parallel (i.e. not deterministic) and thus some lines in the sequence can be interchanged.

References:

- IEC 61508-7 B.2.3.3
- http://www.cs.uwaterloo.ca/~nday/Papers/tr03-01.pdf
- http://www.ida.liu.se/~TDTS30/lectures/lect5.frm.pdf

3.6 Markov modelling

Aim: To evaluate the reliability, safety or availability of a system.

Description: Markov modelling could be applied using state representation and preferably when states are repeatedly returned to. The Markov property characterises the model: given a state the future development does not depend on the history i.e. there is no memory involved. Often transitions are specified using exponential distributions with constant failure rates (one example of a stationary Markov process). A Markov model is described by:

- A number of states.
- A number of transitions with transition probabilities.

In simple cases, the formulae which describe the probabilities of the system are readily available in the literature or can be calculated manually. In more complex cases, some methods of simplification (i.e. reducing the number of states) exist. For very complex cases, results can be calculated by computer simulation of the graph. Tools exist to simulate and compute Markov graphs. There is also the Hidden Markov model where states are not directly observable. Instead indirect means are used where each state produces an output with a specific probability. Other variants of Markov models exits e.g. time dependent versions. The applicability of Markov models is very general but a typical use of Markov models is when repair and recovery states are possible but. The example below shows the idea.

A graph of the system is constructed defining the failure states. These are represented by the nodes of the graph. The edges between nodes, which represent the failure events or repair events, are weighted with the corresponding failure rates or repair rates. The failure events, states and rates can be detailed in such a way that a precise description of the system is obtained. The Markov technique is suitable for modelling multiple systems in which the level of redundancy varies with time due to component failure and repair.

Example: The figure below shows the Markov representation of a system with two identical components. The Common cause factor (β) gives the proportion of common cause faults (0-1). Reliability is defined as

$$R(T) = e^{-\lambda t}$$

which gives a probability of fault for small values of t (Δt) approximately as λΔt. The exponential failure and repair rates (λ and μ respectively) are assumed to be constant.

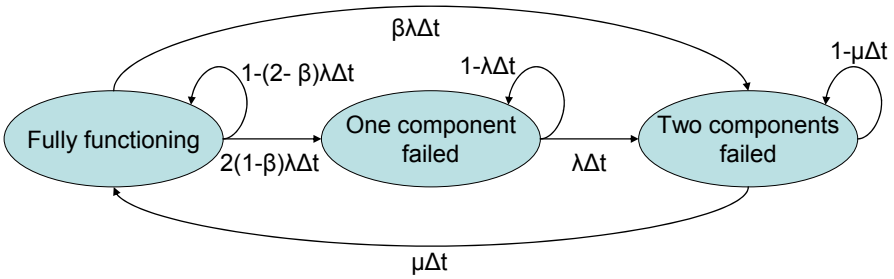


Figure 5: Markov model example

References:

- IEC 61508-7 C.6.4
- IEC 61165

3.7 HW simulated model

Aim: To create a software model of an electrical/electronic circuit.

Description: The function of the safety-related system circuit can be simulated on a computer via a software behavioural model. Individual components of the circuit each have their own simulated behaviour, and the response of the circuit in which they are connected is examined by looking at the marginal data of each component. Some examples of hardware modelling languages are:

- VHDL
- Verilog

Example: A principal VHDL example is shown below for a logical circuit.

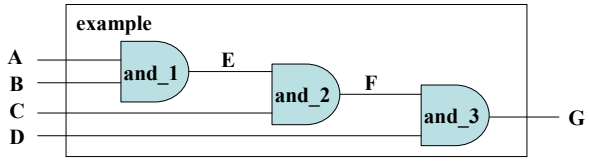


Figure 6: HW model example

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL:
USE WORK.ALL

ENTITY example IS
   PORT(a, b, c, d : IN STD_LOGIC; g : OUT STD_LOGIC);
END example;

ARCHITECTURE example_1 OF example IS

– – Using COMPONENT makes it possible to create structures
   COMPONENT and_function
      PORT(a, b : IN STD_LOGIC; c : OUT STD_LOGIC);
   END COMPONENT;

– – The component realisation is defined elsewhere
   FOR and_function : and_1 USE ENTITY WORK…..
   FOR and_function : and_2 USE ENTITY WORK…..
   FOR and_function : and_3 USE ENTITY WORK…..

– – Internal signals
   SIGNAL e, f : STD_LOGIC;

BEGIN
   and_function :  and_1 PORT MAP(a, b, e);
   and_function :  and_2 PORT MAP(e, c, f);
   and_function :  and_3 PORT MAP(f, d, g);
END example_1;
```

References:

- IEC 61508-7 B.3.6
- http://www.verilog.com/
- http://www.vhdl.org/

3.8 Mechanic part simulated model

Aim: To create a software model of a mechanical unit.

Description: The function of the mechanical parts of the safety-related system can be simulated on a computer via a software behavioural model. Included aspects are e.g. logical behaviour, response time, mechanical redundancy etc. Implementation aspects such as mechanical strength, selection of material etc are not included here since they are not within scope of this report. Since only the behaviour is considered and there is no specific notation for simulated mechanical parts, in principle, any model described above could be used but. However, a simple model should be chosen since the behaviour of a mechanical part normally is very limited.

Example: See model examples above (in a specific case not all are applicable).

References:

- -

4 Analysis

4.1 Introduction

Aim: To avoid systematic faults that can lead to breakdowns in the system under test, either early or after many years of operation.

Description: This systematic and possibly computer-aided approach inspects specific static characteristics of the prototype system to ensure completeness, consistency, lack of ambiguity regarding the requirement in question (for example construction guidelines, system specifications, and an appliance data sheet). A static analysis is reproducible. It is applied to a prototype which has reached a well-defined stage of completion. Some examples of static analysis, for hardware and software, are
– consistency analysis of the data flow (such as testing if a data object is interpreted everywhere as the same value);
– control flow analysis (such as path determination, determination of non-accessible code);
– interface analysis (such as investigation of variable transfer between various software modules);
– dataflow analysis to detect suspicious sequences of creating, referencing and deleting variables;
– testing adherence to specific guidelines (for example creepage distances and clearances, assembly distance, physical unit arrangement, mechanically sensitive physical units, exclusive use of the physical units which were introduced).

References: IEC 61508-7

4.2 Field experience

Aim: To use field experience from different applications as one of the measures to avoid faults either during E/E/PES integration and/or during E/E/PES safety validation.

Description: Use of components or subsystems, which have been shown by experience to have no, or only unimportant, faults when used, essentially unchanged, over a sufficient period of time in numerous different applications. Particularly for complex components with a multitude of possible functions (for example operating system, integrated circuits), the developer shall pay attention to which functions have actually been tested by the field experience. For example, consider self-test routines for fault detection: if no break-down of the hardware occurs within the operating period, the routines cannot be said to have been tested, since they have never performed their fault detection function. For field experience to apply, the following requirements must have been fulfilled:
– unchanged specification;
– 10 systems in different applications;
– 10^5 operating hours and at least one year of service history.
The field experience is demonstrated through documentation of the vendor and/or operating company. This documentation must contain at least
– the exact designation of the system and its component, including version control for hardware;
– the users and time of application;
– the operating hours;
– the procedures for the selection of the systems and applications procured to the proof;
– the procedures for fault detection and fault registration as well as fault removal.

References: IEC 61508-7

4.3 Independence analysis

Aim: To examine if the probability of an unintended dependency or a dependent failure between parts of the design, documentation or persons claimed to be independent is sufficiently low.

Description: The development organization and/or available technical information, PCB, PCBA, and firmware source code is studied/inspected due to the method used for achieving independence and its corresponding justification. The method used for independence may be justified by qualitative or quantitative analysis results or both. Independence may be claimed on different system levels and/or between different design teams or documents whereby it is not always practicable or even possible to quantify the probability of dependent failures.

Reference: None

Example: -Two subsystems claimed to be independent by the method of physical separation may be justified by a well motivated, correctly derived and sufficiently low β-factor.
-A non-safety related and a safety related part of a software claimed independent by the method of separation of data-flows may be justified by the correct application of data-flow analysis and sufficient functional tests
-Two design teams claimed independent by the method of physical separation may be justified by third-part inspection of the design, documentation and development organization

4.4 Proven in use

Aim: To analyse if a subsystem or component may be regarded as proven in use.

Description: The subsystem subjected to be proven in use is analysed for following properties:
-Have the subsystem/component a clearly restricted functionality ?
-The documentary evidence on previous use of the subsystem/component ?
-Measures taken for preventing systematic failures
-The extent of previous use in terms of operating hours and the claimed failure rate (single side lower confidence limit of at least 70%).

Reference: IEC 61508-2, clause 7.4
IEC61508-7, annex D

Example: N/a

4.5 Queuing analysis

Aim: To analyze a queuing model for verifying system properties.

Description: Queuing analysis is used for analyzing performance in a system (or subsystems) described by queues, customers and servers. The evaluation is based on mathematics and preferable made by tools performing calculations or simulations. In order to evaluate the model it is important to analyze e.g. how well mathematical distributions match with reality i.e. to analyze the prerequisites of the queuing model. Once this is made many aspects can be evaluated e.g.

- Upper and lower bounds on throughput and response time.
- Waiting time and total time; average, jitter etc.
- Queue length.
- Loss level (applicable when customers are dropped if queue is longer than a specific value).

- Probability of waiting.
- Server and buffer utilization.

Evaluation can be made for different purposes such as:

- Evaluating the current design.
- Evaluating transient and equilibrium states.
- Finding an optimal design.
- Sensitivity analysis i.e. which parameters affect most?

The aspects above could also be considered for more complex cases of connected queues (queuing networks), e.g. queues connected in series, parallel and in closed loops.

References:

- http://www.cs.washington.edu/homes/lazowska/qsp/
- http://informatics.iic.ac.in/issue2/article.php?aid=1

4.6 Requirement analysis

Aim: To analyse the conformity between the requirements specification and the users functional expectations (the user may as well be another design team or design project internal customer)

Description: The requirement analysis is mainly performed through mutual walkthroughs between the design team producing the requirements specification and the user. Several factors have to be considered during the requirement analysis, such as:
-The user does not completely understand the function they require from the design team
-The development team and the user has different vocabularies causing misunderstandings
-Persons with adequate experience, technical expertise may not be available to lead the requirements engineering activities
Usually is the functionality expressed as use cases and modelled in order to support the discussion between the design team and the user.

Example: N/a

Reference: None

4.7 Performance requirements

Aim: To establish demonstrable performance requirements of a system.

Description: An analysis is performed of both the system and the system requirements specifications to specify all general and specific, explicit and implicit performance requirements.
Each performance requirement is examined in turn to determine
– the success criteria to be obtained;
– whether a measure against the success criteria can be obtained;
– the potential accuracy of such measurements;
– the project stages at which the measurements can be estimated; and
– the project stages at which the measurements can be made.
The practicability of each performance requirement is then analysed in order to obtain a list of performance requirements, success criteria and potential measurements. The main objectives are:
– each performance requirement is associated with at least one measurement;
– where possible, accurate and efficient measurements are selected which can be used as early in the development as possible;
– essential and optional performance requirements and success criteria are specified; and
– where possible, advantage should be taken of the possibility of using a single measurement for more than one performance requirement.

References: IEC 61508-7, clause C.5.19

4.8 Performance analysis

Aim: To ensure that the working capacity of the system is sufficient to meet the specified requirements.

Description: The requirements specification includes throughput and response requirements for specific functions, perhaps combined with constraints on the use of total system resources. The proposed system design is compared against the stated requirements by
– determining the use of resources by each system process, for example, processor time, communications bandwidth, storage devices, etc;
– determining the distribution of demands placed upon the system under average and worstcase conditions;
– computing the mean and worst-case throughput and response times for the individual system functions.
For simple systems an analytic solution may be sufficient, while for more complex systems some form of simulation may be more appropriate to obtain accurate results.
Before detailed modelling, a simpler "resource budget" check can be used which sums the resources requirements of all the processes. If the requirements exceed designed system capacity, the design is infeasible. Even if the design passes this check, performance modelling may show that excessive delays and response times occur due to resource starvation. To avoid this situation, engineers often design systems to use some fraction (for example 50 %) of the total resources so that the probability of resource starvation is reduced.

References: IEC 61508-7, clause C.5.20

4.9 Failure analysis

Aim: To analyse a system design, by examining all possible sources of failure of a system’s components and determining the effects of these failures on the behaviour and safety of the system.

Description: The analysis usually takes place through a meeting of engineers. Each component of a system is analysed in turn to give a set of failure modes for the component, their causes and effects, detection procedures and recommendations. If the recommendations are acted upon, they are documented as remedial action taken.

References: IEC 61508-7, clause B.6.6

4.10 Failure Mode and Effects Analysis (FMEA)

Aim: To analyze potential failure modes and their resulting consequences using a formal and systematic procedure. The FMEA may also be extended to an FMECA – Failure Mode and affects and Criticality Analysis that adds weights to the addressed failure modes and their consequences. FMECA is commonly used in the automotive industry. The field of application for FMEA may be scoped into following major areas:

- Design and development FMEA
- Manufacturing process FMEA

Design and development FMEA aims to increase the quality and/or reliability of a product or to validate the product. The manufacturing process FMEA aims to assure that the product is manufactured as efficiently as possible and will not be addressed further in this section.

Design and development FMEA
Design FMEA is a qualitative and static analysis method that may be used at any level in a system. The method as such is not limited to a particular system and is therefore applicable on for instance:
-Administrative systems
-Logistic systems
-Chemical process systems
-Technical systems.
This section regards only FMEA practiced on technical electronic systems. However these systems are not limited to electronic controls but may as well contain devices such as hydraulic, pneumatic, mechanical or electro mechanical subsystems.

Description: The FMEA is used for proving fulfilment of requirements stated for a specific functionality of a system. In this section it is assumed that all types of requirements are related to a previously performed risk-analysis.

On a very detailed level of analysis the requirements are often divided in two parts:
-The system component fault tolerance
-The system component behaviour at fault

On a more general level of analysis the requirements are more commonly expressed as a scoring system. The including subsystem components in this case shall already have been verified against requirements such as above mentioned.

Examples of potential requirement formulations:
Detailed level of analysis (component level):
-The safety function shall remain operating according to the system specification even in the presence of a single failure.

General level of analysis (system or subsystem level):
-The braking system RPN shall not exceed 980 (see section about FMECA below)

Field of application
The FMEA procedure may be used both as a very low-level and detailed analysis and validation technique but also as a high level and functional analysis technique. The field of application of the FMEA is in this report divided into following three system levels:

-System level
The FMEA focuses on the system behaviour due to failure modes applied on the main system functions. The failure modes considered are functional and may not be expressed in terms of specific

failure causes without continue performing detailed low-level analysis. The result from the FMEA at this level may function as a Hazard and risk analysis as well as a system design analysis.

Examples of potential applicable failure modes:
-Reduced braking capability on the left wheel
-Side door permanently locked

FMEA performer: System integrator

-Subsystem level
The FMEA focuses on the system behaviour due to failure modes applied on the subsystems interfaces. The failure modes considered are still functional but related to individual subsystem interfaces. The result of the FMEA at this level is more related to the system design analysis than the system level FMEA. Several failure modes on subsystem level may compose one single failure mode in the system level FMEA

Examples of potential applicable failure modes:
-Stuck-at failure on a ECU standard digital output
-Wrong message received by certain node on the LIN-bus network

FMEA performer: System integrator or vendor/designer

-Component level
The FMEA focus on the system behaviour due to failure modes applied in individual components (e.g. an ECU). The failure mode is not functional but specifically related to the electronic design of the system component. The result of the system component FMEA is used in order to determine the quality/reliability of the component. The result may also be used for studying the component behaviour at fault due to diagnostic coverage and safe failure fraction and may contain specific reliability prediction data for individual electronic components.

Examples of potential applicable failure modes:
-Short circuit between the base and the emitter on a NPN BJT (transistor)
-Altered value of a resistor
-Erroneous address value set in the program counter (microprocessor)

FMEA performer: Vendor/designer

Selection of applicable failure Modes
Before the procedure of FMEA may be committed relevant failure modes to analyze has to be selected. Several parameters influence on the applicability of the failure modes. All failure modes should be motivated and documented prior to performing the analysis.

Persistence models
-Transient faults – These types of faults persists only for a very short time (e.g. one clock-cycle)
-Intermittent faults – These types of faults persists for a limited amount of time (e.g. D-flip-flop erroneously reseted during one operating cycle)
-Permanent faults – These types of faults persists during the remaining of the system lifetime (unless repaired).

Fault source models
-Disturbances (Electro magnetic interference and signal integrity)
Such faults are prevented by adopting principles for filtering and shielding and make use of appropriate measures when defining the PCB layout. However, if a minor deviation (systematic fault) is introduced in the preventive measures this may consequence in the occurrence of a system transient fault. Another example of such fault source may be spurious injection of heavy-ions.

Examples of such faults:
- Transient memory fault
- Transient analog/digital signal fault
- Transient program execution fault

A difficulty regarding these types of faults is that they may be operation-dependent i.e. only occurring due to a specific operating condition (e.g. at-speed or specific combination of inputs and load)

-Aging and wearing
Each electronic component have a limited lifetime. This lifetime depends on a huge amount of parameters and may be difficult to model. Important parameters to consider regards the operational (de-rating) and environmental conditions in which the component is to be used. When finally the component becomes sufficiently aged or worn-out the component properties alters from the specified ratings and hence becomes degenerated. This kind of operational condition is irreversible.

Example of such faults:
-Permanent open circuit on a transistor base (due to corrosion)
-Permanent altered value of a resistor (resistance shift due to several large transients).
-Permanent welding of relay contact

These types of faults are generally considered as stochastic single point faults and are commonly analysed using qualitative analysis. However, this analysis may be supplemented with quantitative analysis if sufficient statistic data is available.

Subsequent faults:
There are two kinds of subsequent faults, the first type of subsequent faults manifest themselves as single point faults but are in fact caused by a sequence of several subsequent faults. Generally such chains of faults are considered as stochastic single faults. The second type is denoted as common cause faults (CCF). The CCF has a single fault source that distributes as similar subsequent faults through all physically redundant channels.

Examples of subsequent faults:
-Resistor value alters which causes a transistor to burn open and hence the output permanently locked low.
-A transient distributes into the processor providing an erroneous value set in the program counter causing a software failure and hence a potential functional failure.

Example of CCF:
-Unexpected transient occurring in the power supply distributes trough all redundant channels causing a potential functional failure
-Two redundant but similar relays operates under the same operational and environmental conditions. After a certain amount of time the same contacts on both relays welds simultaneously.

From a reliability perspective subsequent faults may cause dependent failures (i.e. that several functional units fails in operation due to failure rates that is not independent).

Operational conditions:
When selecting applicable fault modes the anticipated operational and environmental conditions also should be taken under consideration in addition to the above mentioned fault model parameters. Following parameters is recommended to consider in IEC60812
-The use of the system
-The particular system elements involved
-the mode of operation
-The pertinent operational specifications

-The time constraints
-The environment
Field experience of components from previous use/analysis should also be regarded.

Lists of suitable failure modes for detailed analysis can be found in e.g. ISO 13849-2.

FMECA Failure mode, effects and criticality analysis

The FMECA is an extended form of the FMEA in which the criticality of failure modes and effects also are considered. Criticality combined with severity is a measure of risk. The FMECA usually results in a relative ranking of the contributions to the overall risks. The FMECA differs from commonly used methods for risk analysis/estimation by being less rigorous and hence is it not recommended to use FMECA as a stand-alone measure for judging if the risk is sufficiently reduced. Probabilistic risk analyses (PRA) is such a method and should be used in combination with the FMECA.

The fundamental parameters for FMECA (Risk, R, and Risk Priority Number (RPN)):

$R = S \times P$ and/or $RPN = S \times O \times D$

Where:
S (Severity) is an estimate on how strongly the effects of the failure will affect the system or the user
P (Probability) is an estimate of the probability of occurrence.
O (Occurrence) is an estimate of the frequency of occurrence of a failure mode.
D (Detection) is an estimate of the chance of detecting and eliminating the failure mode before affecting the user. The number is to be ranked in the reverse order due to the other parameters, i.e. the greater the probability for detection is the less resulting RPN or risk.

These parameters may be estimated by ranking instead of probabilistic data. By using the relationships above the main task is to minimize the RPN or the risk.

Ranking tables:
For each parameter used in the FMECA a ranking table is used that scores the parameter, usually the parameters are ranked in the range 1 to 10 as following examples:

S:	1-2	Approximately no effect, not affecting performance
	3-4	Noticeable effect, minor affect on the performance
	5-6	Noticeable effect, medium affect on the performance
	7-8	Effect requires user action, medium alternation of the system behaviour
	9-10	Effect requires instant user action, large alternation of the system behaviour
P:	The probability of the specific failure mode of occurring is: 1/1000 h	
O:	1-2	< 0.001 per 1000 items
	3-4	0.1 per 1000 items
	5-6	2 per 1000 items
	7-8	10 per 1000 items
	9-10	>50 per 1000 items
D:	1-2	The control will most probably detect the failure mode
	3-4	The control have a great chance of detecting the failure mode
	5-6	The control have a moderate chance of detecting the failure mode
	7-8	The control have a less chance of detecting the failure mode
	9-10	The control have a low or minor chance of detecting the failure mode
		Note : “control” may involve a user/operator

The FMECA is most suitable to perform on system level or subsystem level where a specific risk (RPN) is related to a complete function. When performing analysis at component level (FMEA) the same parameters are considered but do not use ranking numbers because single system components usually not directly relates to a risk. The requirement does instead regard the complete component design in order to assure that the component possess a sufficiently high integrity to be part in a subsystem that is related to the risk. Such requirements are usually qualitative (e.g. fault tolerance, component behaviour at fault)

An example od a FMECA worksheet (system level, subsystem level) is shown in table 2 and an example of a FMEA (component level) is shown in table 3.

Performing design FMEA on component level:
S: Failure effect dangerous ? Yes/No
O/P: Only use statistically motivated values on probabilities
D: Is the failure mode detected? Yes/No

Reference:
IEC 60812: Analysis techniques for system reliability – procedure for failure mode and effects analysis
IEC 61508-7, clause B.6.6.1

: **Table 2:** Example design FMECA worksheet (System level or subsystem level)

Space for administrative information														
Item ref.	**Item description**	**Failure modes**	**Possible failure cause**	**Local failure effect**	**End item effect**	**S**	**O**	**D**	**P**	**RPN**	**RISK**	**Method of detection**	**Notes/ Remarks**	**Design state**
1.1	ECU	HSD-Output stuck low	Moister (sneak circuit to ground)	Supply to starter engine disabled	Vehicle cannot restart	7	2	10	N/a	140	Low to Medium	The failure mode will most probably not be detected prior to occurrence	RPN too high	To be subjected for further design modifications
2.1	Lamp	Broken	Aging	Lamp cannot emit light	Left headlight out of operation	8	-	1	1.14E-4/h	-	Medium to high	The open circuit is most probably detected by the current feedback signal	1.24E-4/h is sufficient (previously 2.35E-4/h)	Finished

Table 3: Example design FMEA worksheet (Component level)

Space for administrative information									
Item Ref.	**Item description**	**Failure modes**	**Local failure Effect**	**End item failure effect**	**S**	**D**	**P** (E-5)	**Method of detection**	**Notes/ remarks**
A1.1	R11, SMT resistor, 10k (pull-up for Q12)	Open	Floating gate on Power pFET, may increase output sensitivity for interferences	ECU LSD output may behave unpredictably	Not dangerous	Yes	2.3	Internal CPU comparison due to redundant feed-back signals (see firmware file: Handle_failures.c)	Robustness too low – use resistor with more suitable MTTF
A1.2		Short	Power pFET gate always high. The transistor cannot turn to ON-state	ECU LSD output permanently disabled	Not dangerous	Yes	2.3	See A1.1	See A1.1
A1.3		Altered value	Assume 50% and 150%, function not primarily affected	Not affected	Not Dangerous	No	2.3	The failure remains undetected	No

Notes: The exemplified failure modes, effects or judgements are not fetched from a real case-study but serve only as hypothetic illustrations!
Abbreviations: ECU – Electronic Control Unit, MTTF – Mean Time To Failure, HSD – High Side Digital, LSD – Low Side Digital
The figure "nE-m" denotes $n*10^{-n}$

4.11 Common cause failure analysis

Aim: To determine potential failures in multiple systems or multiple subsystems which would undermine the benefits of redundancy, because of the appearance of the same failures in the multiple parts at the same time.

Description: Safety related systems often use hardware and software redundancy in order to avoid random hardware failures in components or subsystems. In redundant systems there is always a chance that the same failure will occur in all channels simultaneously. The common cause failure analysis results in a quantitative prediction on how large partition of the total rate of failure for one channel also occurring in several channels simultaneously. For an example, in a two channelled redundant system channel A has the total rate of dangerous failures λ_D. The partition of λ_D causing a common cause failure in both channel A and B is expressed as $\beta \lambda_D$ where β is denoted the common cause factor.

In contradiction with the procedure of retrieving component failure rates is the β-factor commonly qualitatively estimated by the means of a scoring table. Such a table usually take into account aspects such as:
-Separation/segregation
-Diversity/Redundancy
-Complexity/design/application/maturity/experience
-Assessment/Analysis and feedback of data

As described by the above examples the scoring table for the β-factor is not restricted to solely hardware or software but considers aspects from the whole development process.

The β-factor is usually estimated to a value in the range 0 - 20 % of the total failure rate of one channel.

References: IEC61508-6, annex D
IEC 61508-7, clause C.6.3

4.12 Event tree analysis

Aim: To model, in a diagrammatic form, the sequence of events that can develop in a system after an initiating event, and thereby indicate how serious consequences can occur.

Description: On the top of the diagram is written the sequence conditions that are relevant in the progression of events that follow the initiating event. Starting under the initiating event, which is the target of the analysis, a line is drawn to the first condition in the sequence. There the diagram branches off into "yes" and "no" branches, describing how future events depend on the condition. For each of these branches, one continues to the next condition in a similar way. Not all conditions are, however, relevant for all branches. One continues to the end of the sequence, and each branch of the tree constructed in this way represents a possible consequence. The event tree can be used to compute the probability of the various consequences, based on the probability and number of conditions in the sequence.

Example: A simple example of how software download can be examined is given. The intention is not to illustrate the proper download procedure, but to illustrate the principles of an event tree.

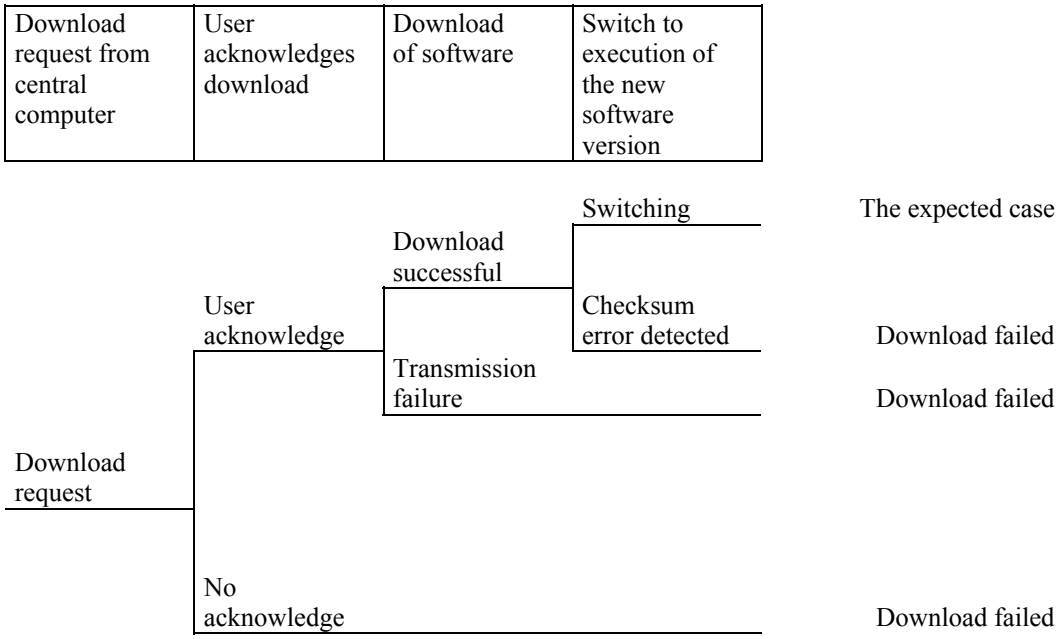


Figure 7: Event tree example

References: IEC 61508-7, clause B.6.6.3

4.13 Fault Tree Analysis

Aim: To aid in the analysis of events, or combinations of events, that will lead to a hazard or serious consequence.

Description: Starting at an event which would be the immediate cause of a hazard or serious consequence (the "top event"), analysis is carried out along a tree path. Combinations of causes are described with logical operators (and, or, etc). Intermediate causes are analysed in the same way, and so on, back to basic events where analysis stops. The method is graphical, and a set of standardised symbols (such as AND, OR) are used to draw the fault tree. The technique is mainly intended for the analysis of hardware systems, but there have also been attempts to apply this approach to+- software failure analysis.

Example: A simple example of a fault tree is given. Inadmissible modification of parameter values for engine control is chosen as top event. The fault tree can be used to indicate possible weaknesses in the design.

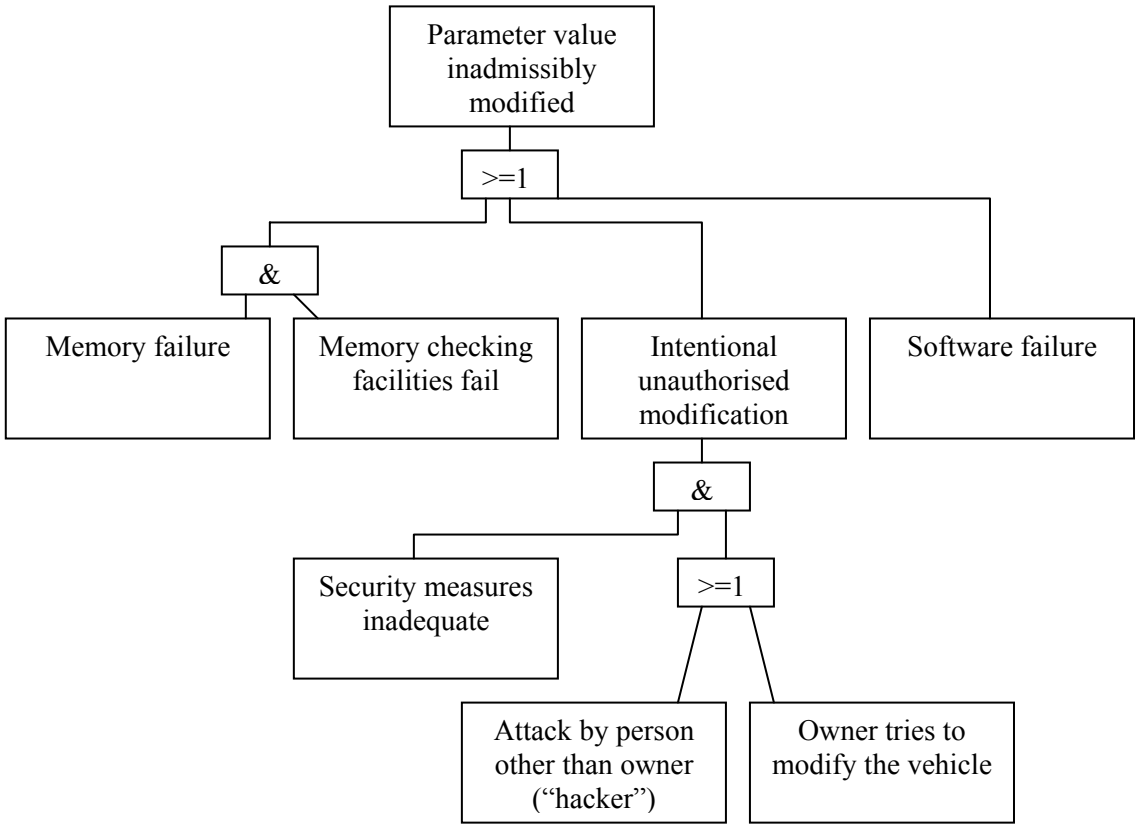


Figure 8: Fault tree example

Reference: IEC 61508-7, clause B.6.6.5
IEC 61025

4.14 Cause – Consequence Analysis

Aim: To model, in a diagrammatic form, the sequence of events that can develop in a system as a consequence of combinations of basic events.

Description: The technique can be regarded as a combination of fault tree and event tree analysis. Starting from a critical event, a cause consequence graph is traced backwards and forwards. In the backward direction it is equivalent to a fault tree with the critical event as the given top event. In the forward direction the possible consequences arising from an event are determined. The graph can contain vertex symbols which describe the conditions for propagation along different branches from the vertex. Time delays can also be included. These conditions can also be described with fault trees. The lines of propagation can be combined with logical symbols, to make the diagram more compact. A set of standard symbols is defined for use in cause consequence diagrams. The diagrams can be used to compute the probability of occurrence of certain critical consequences.

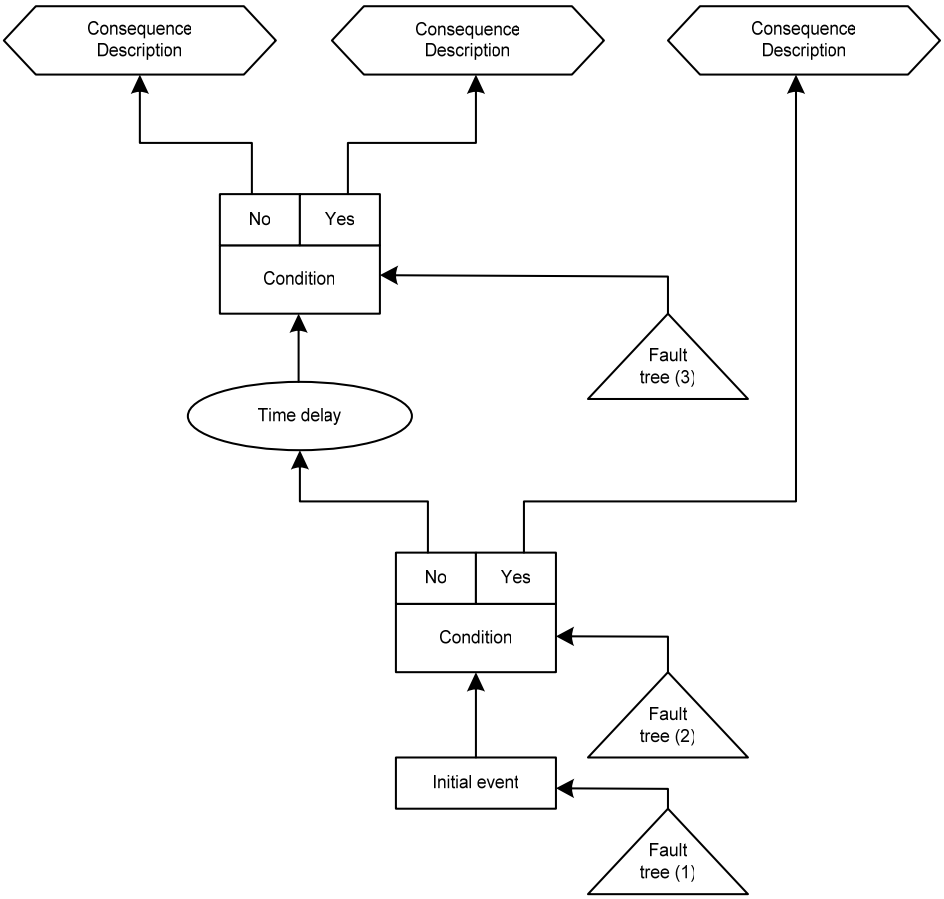


Figure 9: Cause – consequence example

References: IEC 61508-7, clause B.6.6.2

4.15 Reliability block analysis

Aim: To analyse the relationship between system functions and to predict how modifications impacts on the subsystem quality and/or reliability.

Description: Because the RBD model is developed with the requirement that each sub block shall be independent from all other sub blocks the RBD is very suitable to use for investigating whether a system truly meets the MooN redundancy requirement as required in the requirements specification. A RBD will rapidly reveal any unintended dependencies between sub blocks. However, it is crucial that the RBD model used in this analysis is composed at a suitable system level.

Once the RBD model is adequately composed and verified and all failure rates are properly derived the model can easily be used for analysing the effect of minor system changes/modifications. Questions that may be answered when performing such an analysis:
-Is it possible to exchange one expensive and very reliable component to a less expensive and more unreliable component without affecting the overall quality/reliability (sensitivity analysis) ?
-Is the total system reliability depending on one specific and limited part of the system which hence is to be subjected for further design effort ?

The RBD is also useful for supporting the creation of the Markov model.

Example: For basic reliability theory, please refer to IVSS Autoval Report 5.1, chapter 3.7.2. All failure rates are constant in the following examples.

Series systems: The RBD in figure 10 is a system of three series connected components (A, B and C), e.g. a sensor, a microcontroller and an actuator. This example system could function as a speedometer in a car where the sensor is a CAN-controller which receives the speed information from the vehicle main node and the actuator is the indicator on the panel which is controlled by the microcontroller.

For each included block a corresponding failure rate has to be defined. These failure rates could e.g. be retrieved from FMEA results.

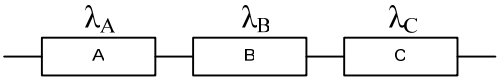


Figure 10: Series RBD example

The block diagram states that if either component A, component B or component C fails then will the complete speedometer fail in operation. Or the other way around, all components (A and B and C) have to function correctly in order for the speedometer to function correctly.

Expression for series systems:

$R_{SERIES}(t) = \prod_{i=1}^{n} R_i(t)$ The probability of system success.

and,

$F_{SERIES}(t) = 1 - \prod_{i=1}^{n} F_i(t)$ The probability of system failure.

The probability that the speedometer successfully operates at a specific time is $R_{Speedometer}(t) = e^{-(\lambda_A + \lambda_B + \lambda_C)t}$

Using $MTTF = \int_0^{\infty} R(t)dt$ gives that $MTTF = \frac{1}{\lambda_A + \lambda_B + \lambda_C}$

Parallel systems: The RBD in figure 11 is a system of parallel connected components (A and B). For an example, these blocks could represent electronic brake nodes on each wheel on a motorcycle.

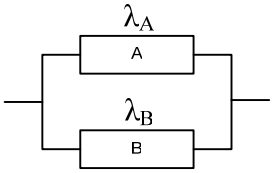


Figure 11: Parallel RBD example

The block diagram states that the motorcycle braking capacity is not lost until both brake nodes (A and B) have failed or the other way around, only one of the brake nodes has to function (A or B) in order to maintain sufficient braking capacity.

Expression for parallel systems:

$R_{PARALELL}(t) = 1 - \prod_{i=1}^{n} F_i(t)$ The probability of system success.

$F_{PARALELL}(t) = \prod_{i=1}^{n} F_i(t)$ The probability of system failure.

The reliability for a parallel system with two elements is:
$R_{PARALELL}(t) = R_A(t) + R_B(t) - R_A(t)R_B(t)$

The reliability function for the brake system: $R_{Brake}(t) = e^{-\lambda_A t} + e^{-\lambda_B t} - e^{-(\lambda_A + \lambda_B)t}$

Applying the formula for MTTF gives that: $MTTF = \frac{1}{\lambda_A} + \frac{1}{\lambda_B} - \frac{1}{\lambda_A + \lambda_B}$

Also notice that $MTTF \neq \frac{1}{\sum \lambda_n}$ for parallel systems

References: IEC 61508-7, clause C.6.5
Control Systems Safety Evaluations & Reliability 2nd Edition, William M. Goble

4.16 Markov analysis

Aim: To evaluate and verify that the Markov model correspond with the system specification and that the resulting availability corresponds to the requirements.

Description: The Markov model may be used in order to verify how different states of operation conforms with the system specification. An example: If a system enters one of its degraded states of operation, such as safe state it may be required that the system shall not return to other states unless repaired during the mission time. This may be verified using a Markov model if it is adequately composed. However, it is important that a correct system level is applied during the Markov modelling before engaging the analysis.

The Markov model may also be used for quantitative analysis and evaluation. There are three important parameters to be considered during such analysis: The probability of transiting between two states in the Markov model (λ – failure rate), the probability of returning from another state in the Markov model (μ – repair rate) and the design of the Markov model as such – the n x n transition matrix. If these parameters are correctly estimated, the probability of being in any of the states may be calculated at a given time t.

Using the Markov model the following aspects related to dependability could be analyzed:

- System reliability

- MTTF and MTBF
- Availability (or unavailability)
- Probability of being in a specific state at a specific point of time
- The probability of a certain sequence of states

Parametric studies can be performed to assess the impact of:

- different system mode of operation/configuration
- different rates and mission times (proof test interval)
- different repair times and strategies

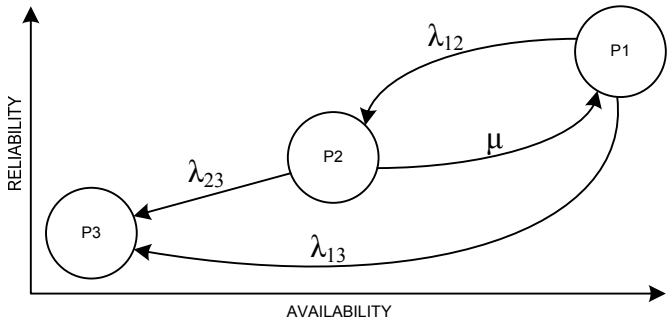


Figure 12: Markov model example

The n x n transition matrix The Markov-diagram in figure 12 may be represented as a transition matrix containing as many rows and columns as there is states in the Markov diagram. Each position in this matrix defines a transition from one state to another.

For example: The position of the first row and the third column represents the probability of transiting between the state P1 and the state P3 which according to figure 12 is λ_{13}. The position of the first row and the first column will in the same manner represent the probability of P1 transiting to P1 (i.e. not leaving the state P1) which is: $1-(\lambda_{12}+\lambda_{13})$. The third state in this Markov diagram is an absorption state which means that there are no exit transitions from this state, so the probability of staying in state P3 equals one P3 is entered.

The complete transition matrix T for figure 12 becomes:

$$T = \begin{bmatrix} 1-(\lambda_{12}+\lambda_{13}) & \lambda_{12} & \lambda_{13} \\ \mu & 1-(\mu+\lambda_{23}) & \lambda_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

The probability of being in one of the states always equals 1. The Markov diagram in this example will sooner or later end up in the absorption state and has no steady-state condition. Only Markov diagrams with no absorption states may reach steady-state.

Calculating probabilities using a discrete time Markov model Discrete methods can be used if the failure rates are constant in time. It can be shown that if the probabilities in the transition matrix T are given in per hour basis and the start condition (i.e. the probability of starting in respective state) is e.g. $\overline{S}_0 = [1 \quad 0 \quad 0]$ then will $\overline{S}_1 = \overline{S}_0 \cdot T$ provide the probability of being in any state after one hour. Continuing iterating will provide the probability of being in any state at the n:th hour in S_n.

The limiting state (steady-state) probabilities $\overline{S}_n^L$ for the n:th state of the Markov model may be obtained by solving the equation: $\overline{S}^L \cdot T = \overline{S}^L$

The MTTF may be calculated out of the discrete Markov model by counting how many time increments (iterations) the system dwells in each transient state, disregarding all absorbing states. The next time increment should in average lead to a transit to an absorbing state.
Step 1: Cross out all rows and columns of the absorbing states from the matrix T and call the resulting matrix Q.
Step 2: Subtract the Q matrix from the unit matrix I
Step 3: The number of time increments for each state is then given by $N = [I - Q]^{-1}$
Step 4: The sum of time increments in one of the rows in the N matrix is the MTTF provided that the system started in the corresponding state.

This procedure may also be used for calculating an analytical solution for the MTTF.

Calculating probabilities using a continuous time Markov model
For continuous time analysis the Markov model is transformed into a mathematical representation and becomes a linear differential equation system of the first order, according to: $\frac{d}{dt}\overline{P}_n(t) = A \cdot \overline{P}_n(t)$
Where P(t) is the probability of being in state n at a specific time t.

For figure 12 may the A matrix be derived by multiplying the transition matrix T with the vector P(t) and then analyze the probability of remaining in each state during the time interval

The A matrix may be obtained by the following expression: $A = T^T - I$ and becomes for figure 12:

$$A = \begin{bmatrix} -(\lambda_{12} + \lambda_{13}) & \mu & 0 \\ \lambda_{12} & -(\mu + \lambda_{23}) & 0 \\ \lambda_{13} & \lambda_{23} & 0 \end{bmatrix}$$

The analytical solution to the continuous Markov model may be difficult to calculate by hand, especially if the amount of states are larger than 4.

If the steady state solution is preferred the dynamic part of the equation is set to zero and the equation may be solved as a linear equation system. If the transient solution is preferred, is it possible to e.g. use the Laplace transform to analytically solve the equation. The solution will become polynomials consisting of a transient part and a steady-state part if no absorbing states are included.

Repair-rate (μ)
The repair rate denotes the probability that the failure will be repaired during a system inspection or service. This probability is usually estimated by two parameters: The average repair time (T_R) and the periodic inspection time (T_I). The average time to detect a fault with a periodic inspection time T_I is $T_I/2$ if the probability of detecting the fault is equal at all times. The total repair rate is given by:

$$\mu = \frac{1}{\frac{T_I}{2} + T_R}$$

Example: The Markov model in figure 12 could represent a two channelled control system according to:

Markov diagram states description	
P1	Normal operation. Both channels operate according to the specification.
P2	Degraded operation. A fault has occurred and the system operates only one of the two channels according to the specification.
P3	Dangerous operation. A fault has occurred and no channel operates according to the specification, which may consequence in a dangerous operation.

Reference: IEC 61508-7, clause C.6.4
MIL-HDBK-338b
Control Systems Safety Evaluations & Reliability 2nd Edition, William M. Goble

4.17 State transition diagram

Aim: To specify or implement the control structure of a system.

Description: A specification or design expressed as a finite state machine can be checked for
– completeness (the system must have an action and new state for every input in every state);
– consistency (only one state change is described for each state/input pair); and
– reachability (whether or not it is possible to get from one state to another by any sequence of inputs).
These are important properties for critical systems. Tools to support these checks are easily developed. Algorithms also exist that allow the automatic generation of test cases for verifying a finite state machine implementation or for animating a finite state machine model.

Example: A simple example of the power-on procedure and operation of a safety system for breaking assistance is given. The intention is to illustrate the diagram, not to give requirements for a correct function.

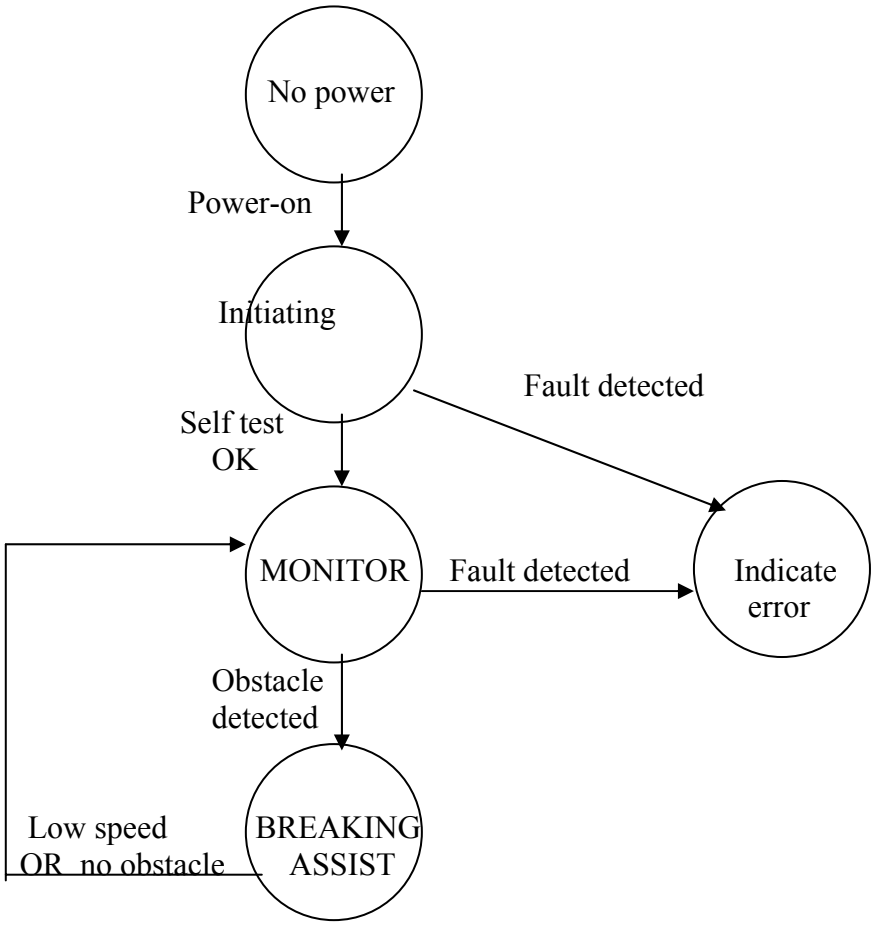


Figure 13: State transition example

Reference: IEC 61508-7, clause B.2.3.2

4.18 Data flow analysis

Aim: To detect potentially incorrect data structures.

Description: The dataflow analysis is a method for analysing specific data paths through the software. The method focuses on the parameters and processes or parts of the system which transforms a data value to another data value without regarding any conditions. Data flow analysis is a graphical and descriptive method that produces flow-charts.

When performing the method certain primitive symbols are used such as data stores (containing a data value), processes (transforming a data value), initiating event(s) and final event(s). The primitives are connected by arrows that show how data propagates through the system from the inputs to the outputs.

The outcome of the method gives a survey of interesting parts (functions) of the software and more or less detailed information on which parameters initiate the function(s) and on which parameters are finally affected by the function(s). The primary purpose of the analysis is to verify unwanted or especially critical data-paths or processes affecting the system functions.

The analysis can e.g. check for
– dependencies between different data flows
– variables that may be read before they are assigned a value – this can be avoided by always assigning a value when declaring a new variable;
– variables that are written more than once without being read – this could indicate omitted code;
– variables that are written but never read – this could indicate redundant code.
A data flow anomaly will not always directly correspond to a program fault, but if anomalies are avoided the code is less likely to contain faults.

The data flow may also be suitable for performing analysis on an entity model.

Example: The following example illustrates a data flow of a very simplified application handling a PWM-controlled function.

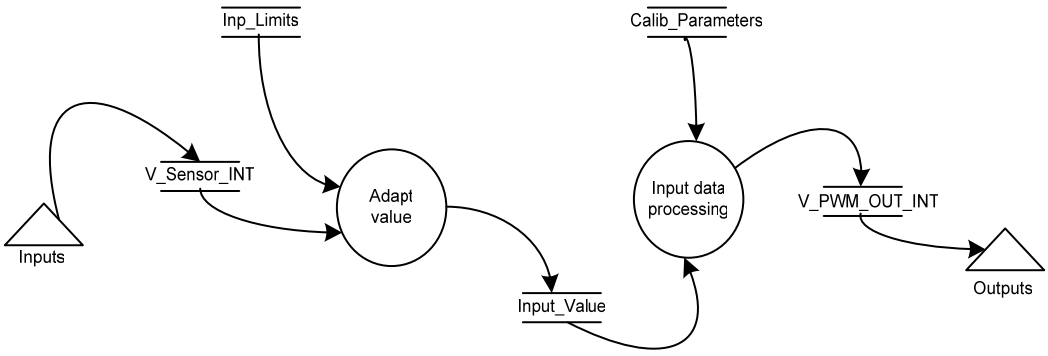


Figure 14: Data flow example

Reference: IEC 61508-7, clauses C.2.2 and C.5.10

4.19 Control flow analysis

Aim: To detect potentially incorrect program structures.

Description: The control flow analysis is a method for analysing the behaviours and states of a control system. The method focuses on the separate modules included in the system, their behaviour (states) and emphasis especially on the conditions for entering these states. Commonly used means for this analysis are representation of the system as a finite state machine and development of state transition diagrams in conjunction with system control flow charts. Regarding software the control flow analysis may also be used as a static testing technique for finding suspect areas of code that do not follow good programming practice. The program is analysed producing a directed graph which can be further analysed for e.g.
– inaccessible code, for instance unconditional jumps which leaves blocks of code unreachable;
– knotted code. Well-structured code has a control graph which is reducible by successive graph reductions to a single node. In contrast, poorly structured code can only be reduced to a knot composed of several nodes.

Example: The control flow may be displayed using different types of diagrams depending on which system level interesting for analysis. Common used diagrams are the state transition diagrams, the control flow graph and the call graph. In addition, networks with Boolean gates may be used in order to investigate more complex compositions of conditions.
The control flow may be used for e.g following questions:
-Does the system different modes of operation corresponds with the specification (does a detected failure always results in the specified safe-state ?)
-Is the watch dog implemented properly ?
-Is memory tests performed according to the requirements ?
-Are redundant variables used trough the complete design ?

The figure below only displays the graphical shape of a control flow graph and a call graph due to an example part of a source code to the left. For a real application these graphs have to be hierarchically designed, otherwise they become enormous.

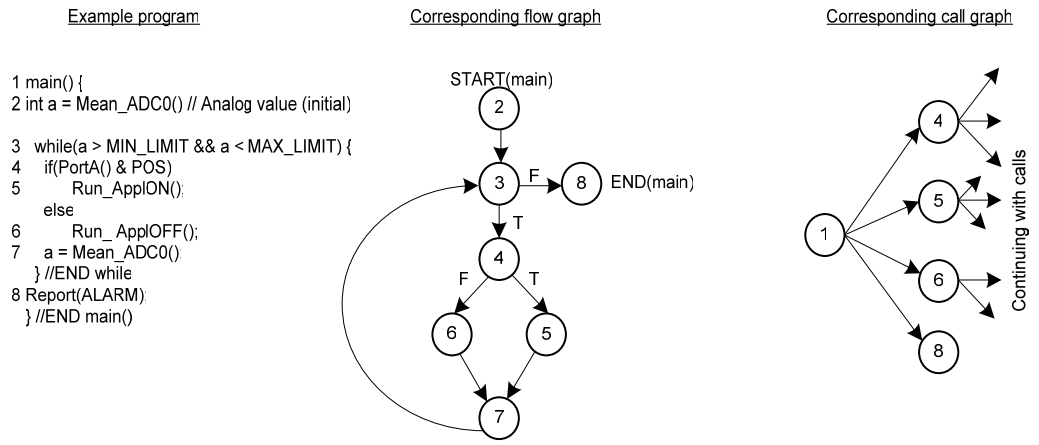


Figure 15: Control flow example

References: IEC 61508-7, clause C.5.9

4.20 Sneak circuit analysis

Aim: To detect an unexpected path or logic flow within a system which, under certain conditions, initiates an undesired function or inhibits a desired function.

Description: A sneak circuit path may consist of hardware, software, operator actions, or combinations of these elements. Sneak circuits are not the result of hardware failure but are latent conditions inadvertently designed into the system or coded into the software programs, which can cause it to malfunction under certain conditions. Categories of sneak circuits are
– sneak paths which cause current, energy, or logical sequence to flow along an unexpected path or in an unintended direction;
– sneak timing in which events occur in an unexpected or conflicting sequence;
– sneak indications which cause an ambiguous or false display of system operating conditions, and thus may result in an undesired action by the operator;
– sneak labels which incorrectly or imprecisely label system functions, for example, system inputs, controls, displays, buses, etc, and thus may mislead an operator into applying an incorrect stimulus to the system.
Sneak circuit analysis relies on the recognition of basic topological patterns with the hardware or software structure (for example, six basic patterns have been proposed for software). Analysis takes place with the aid of a checklist of questions about the use and relationships between the basic topological components.

Example: N/a

References: IEC 61508-7, clause C.5.11

4.21 Consistency analysis

Aim: To detect inconsistency between the requirement specification and the complete design and related set of documentation

Description: This systematic and possible computer-aided approach inspects specific static characteristics of the prototype system to ensure completeness, consistency, and clearness of the requirements specification (e.g analysing that information is similarly interpreted trough construction guidelines, system specification and the appliance data sheet).

Example: N/a

Reference: IEC61508-7, clause B.6.3, C.5.24

4.22 Impact analysis

Aim: To determine the effect that a modification to a component, subsystem or system will have to other components, subsystems or systems.

Description: Prior to a modification being performed, an analysis should be undertaken to determine the impact of the modification on the actual system, and also to determine which other components, subsystems or systems are affected. After the analysis has been completed a decision is required concerning the reverification or revalidation. Possible decisions are:

– only the modified parts of the system is reverified/revalidated;
– all affected system parts are reverified/revalidated; or
– the complete system is reverified/revalidated.

Example: N/a

References: IEC 61508-7, clause C.5.23

4.23 Protocol analysis

Aim: To examine the communication protocol used by the system for relevant functions, such as software download or transmission of critical data.

Description: The communication protocols used for communication with control system includes techniques and measures to make the data transfer as reliable as possible. Transmission of critical data will be affected by the properties of the communication protocol.

The technical documentation describing the communication protocol is analyzed due to e.g.:
-Communication topology
-Measures used for increasing the data integrity
-Message composition

FMEA may be applicable for analysing the higher levels of the communication protocol.

Example: N/a

Reference: IEC 61508-2, clause 7.4.8

4.24 Complexity metrics

Aim: To predict the attributes of programs from properties of the software itself or from its development or test history.

Description: These models evaluate some structural properties of the software and relate this to a desired attribute such as reliability or complexity. Software tools are required to evaluate most of the measures. Some of the metrics which can be applied are given below:
– graph theoretic complexity – this measure can be applied early in the lifecycle to assess trade-offs, and is based on the complexity of the program control graph, represented by its cyclomatic number;
– number of ways to activate a certain software module (accessibility) – the more a software module can be accessed, the more likely it is to be debugged;
– Halstead type metrics science – this measure computes the program length by counting the number of operators and operands; it provides a measure of complexity and size that forms a baseline for comparison when estimating future development resources;
– number of entries and exits per software module – minimising the number of entry/exit points is a key feature of structured design and programming techniques.

Example: N/a

References: IEC 61508-7, clause C.5.14

4.25 Worst case analysis

Aim: To avoid systematic failures which arise from unfavourable combinations of the operational load, the environmental conditions and the component tolerances.

Description: The operational capacity of the system and the component dimensioning is examined or calculated on a theoretical basis. The operational load conditions and environmental conditions are set to be the most unfavourable. Essential responses of the system are inspected and compared with the specification. In the context of software analysis, worst case can refer to memory needs and execution time.

Example: N/a

Reference: IEC 61508-7, clause B.6.7

4.26 Worst Case Execution Time analysis

Aim:
The purpose of Worst Case Execution Time (WCET) analysis is to provide a priori information about the worst possible execution time of a program before using the program in a system.

Description:
WCET estimates are used in real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times.
The goal of WCET analysis is to estimate the worst-case execution time of the program. To generate a WCET estimate, the program must be processed through the following phases:

Program flow analysis The program flow analysis phase determines the dynamical behaviour of the program. The result is information about which functions get called, how many times loops iterate, dependencies between if-statements, etc

Low level analysis The purpose of the low level analysis is to determine the execution time for each basic unit of flow (e.g. an instruction or a basic block), given the architecture and features of the target system. Examples of features to consider are pipelines and caches.

Calculation The calculation phase generates the final WCET estimate for the program, given the program flow and the low-level analysis results. Following are the three main categories of WCET calculation methods:

-Path-based calculation
In a path-based calculation, the WCET estimate is generated by calculating times for the different paths in a program, searching for the path with the longest execution time. The defining feature is that possible execution paths are explicitly represented.

-Tree-based methods
In Tree-based methods, the final WCET is generated by a bottom-up path through the tree representing the program. The analyses for smaller parts of the program are used to make timing estimates for larger parts of the program.

-Implicit Path Enumeration Technique (IPET)–based methods

IPET-based methods express program flow and atomic execution times using algebraic and/or logical constraints. The WCET estimate is calculated by maximising an objective function, while satisfying all constraints.

Example: N/a

References: None

5 Dynamic methods

5.1 Introduction

Aim: To detect specification failures by inspecting the dynamic behaviour of a prototype at an advanced state of completion.

Description: The dynamic analysis of a safety-related system is carried out by subjecting a near-operational prototype of the safety-related system to input data which is typical of the intended operating environment. The analysis is satisfactory if the observed behaviour of the safety-related system conforms to the required behaviour. Any failure of the safety-related system must be corrected and the new operational version must then be reanalysed.

Example: N/a

References: IEC 61508-7, clause B.6.5

5.2 Functional testing

Aim: To prove conformity between the actual system and the functional requirement specification.

Description: Functional tests are performed in order to verify that the specified characteristics of the control system have been achieved. The system is given input data which adequately characterises normal expected operation. The outputs are observed and their response is compared with the functional specification.

Functional testing may also be expanded in order to test the functional requirement specification due to unspecified inputs. The behaviour of the system is in then studied when applying input signals which are expected to occur rarely or outside of the functional specification (such as incorrect values).

Functional testing can be performed on sub-modules or on the complete system. However, functional testing requires that a physical unit (i.e. prototype or product) is available.

The result of a functional test may reveal present failures in software or hardware but not system design or architectural weaknesses. The functional tests performed are always limited to the chosen set of input data and the environment in which the tests have been conducted. These two parameters tend to be more or less ideal during test performance (e.g. due to the safety of the personal performing the tests or risk of damage to the target system).

Example (functional test): Test that the system corresponds if function with the user manual

Example (expanded functional test): Lowering the power supply to a test object to 5% below its rated minimum level and then exercise the function and verifying correspondence with the requirements.

References: Functional testing : IEC 61508-7, clause B.5.1
Expanded functional testing : IEC 61508-7, clause B.6.8

5.3 Regression test

Aim: To increase the efficiency and repeatability during verification by re-using previous test cases when adding changes/modifications to the system.

Performance: By building a library containing automated test cases originating from the initial attempt of applicable test cases. Each time a modification to the system is issued all test cases in the library are used. If the existing test cases are not applicable in order to test the actual modification a new test case is defined and added to the test case library. The test case library should periodically be reviewed in order to eliminate redundant or unnecessary test cases. Regression testing is a method of planning and carrying out tests and is not an explicit test method. The regression test will hence not produce new data but will increase the probability that no adequate test procedure is left out during verification.

Example: N/a

Reference: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconunittesting.asp

5.4 Black-box testing

Aim: Test whether the software or hardware unit functionality correspond to the requirement specification.

Description: The functions of a system or program are executed in a specified environment with specified test data which is derived systematically from the requirement specification. This exposes the behaviour of the system and permits a comparison with the specification. No knowledge of the internal structure or design of the system is used to guide the testing or the interpretation of the test results.
Black box testing may be used for e.g. module test, unit test, integration test or complete system test.

Example: N/a

References: IEC 61508-7, clause B.5.2

5.5 White-box testing

Aim: Test whether the software or hardware unit functionality corresponds to the requirement specification and to relate test results to the detailed system design.

Description: White-box testing means that the operation of the system is tested using knowledge on how the software is designed. Test cases for the functional testing will be selected based on knowledge of the software, detailed design speciation, and other development documents. These test cases challenge the control decisions made by the program; and the program's data structures including configuration tables.

Example: N/a

Reference: None

5.6 Interface testing

Aim: To detect errors in the interfaces of subprograms.

Description: Several levels of detail or completeness of testing are feasible. The most important levels are tests for
– all interface variables at their extreme values;
– all interface variables individually at their extreme values with other interface variables at normal values;
– all values of the domain of each interface variable with other interface variables at normal values;
– all values of all variables in combination (this will only be feasible for small interfaces);
– the specified test conditions relevant to each call of each subroutine.
These tests are particularly important if the interfaces do not contain assertions that detect incorrect parameter values. They are also important after new configurations of pre-existing subprograms have been generated.

Example: N/a

References: IEC 61508-7, clause C.5.3

5.7 Boundary value analysis

Aim: To detect errors occurring at parameter limits or boundaries on software or hardware units.

Description: The input domain of the function is divided into a number of input classes according to the equivalence relation. The tests should cover the boundaries and extremes of the classes. The tests check that the boundaries in the input domain of the specification coincide with those in the program. The use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention:
– zero divisor;
– blank ASCII characters;
– empty stack or list element;
– full matrix;
– zero table entry.
Normally the boundaries for input have a direct correspondence to the boundaries for the output range. Test cases should be written to force the output to its limited values. Consider also if it is possible to specify a test case which causes the output to exceed the specification boundary values. If the output is a sequence of data, for example a printed table, special attention should be paid to the first and the last elements and to lists containing none, one and two elements.

Example: N/a

References: IEC 61508-7, clause C.5.4

5.8 Avalanche/stress testing

Aim: To burden the test object with an exceptionally high operational workload (on the inputs) in order to show that the test object would stand normal workloads.

Description: There are a variety of test conditions which can be applied for avalanche/stress testing. Some of these test conditions are:
– if working in a polling mode then the test object gets much more input changes per time unit as under normal conditions;
– if working on demands then the number of demands per time unit to the test object is increased beyond normal conditions;
– if the size of a database plays an important role then it is increased beyond normal conditions;
– influential devices are tuned to their maximum speed or lowest speed respectively;
– for the extreme cases, all influential factors, as far as is possible, are put to the boundary conditions at the same time.
Under these test conditions the time behaviour of the test object can be evaluated. The influence of load changes can be observed. The correct dimension of internal buffers or dynamic variables, stacks, etc. can be checked.

Example: N/a

References: IEC 61508-7, clause C.5.21

5.9 Worst-case testing

Aim: To test the cases specified during worst-case analysis due to the worst case operational load, environmental conditions and component tolerances.

Description: The operational capacity of the system and the component dimensioning is tested under worst-case conditions according to the result of the worst-case analysis. The environmental conditions are changed to their highest/lowest permissible marginal values. The most essential responses of the system are inspected and compared with the specification.

Example: Lowering the power supply to a test object to 5% below its rated minimum level and then exercise the function and verifying correspondence

References: IEC 61508-7, clause B.6.9

5.10 Equivalence classes and input partition testing

Aim: Test software adequately using a minimum of test data. The test data is obtained by selecting the partitions of the input domain required to exercise the software.

Description: This testing strategy is based on the equivalence relation of the inputs, which determines a partition of the input domain. Test cases are selected with the aim of covering all the partitions previously specified. At least one test case is taken from each equivalence class. There are two basic possibilities for input partitioning which are
– equivalence classes derived from the specification – the interpretation of the specification may be either input orientated, for example the values selected are treated in the same way, or output orientated, for example the set of values lead to the same functional result;

– equivalence classes derived from the internal structure of the program – the equivalence class results are determined from static analysis of the program, for example the set of values leading to the same path being executed.

The input data space is subdivided into specific input value ranges (equivalence classes) with the aid of the specification. Test cases are then formed from the
– data from permissible ranges;
– data from inadmissible ranges;
– data from the range limits;
– extreme values;
and combinations of the above classes.

Example: N/a

References: IEC 61508-7, clause C.5.7

5.11 Testing using test case classification

Aim: Test software adequately using a minimum of test data. The test data is obtained by specifying test cases for exercising the software.

Description: Test cases are created by using the Classification Tree Method (CTM) which is a systematic procedure that specifies test cases based on problem definitions. The purpose of CTM is to be able

- to visualise the test idea.
- to structure complex test problems by dividing them into classes.
- to check for gaps in the specification.
- to estimate the required test effort in terms of the minimum and maximum number of test cases.

CTM is useful both in the creation of test cases and in the development of the tested object since it gives feedback to the specification document. If the expected result of a test is not available in the document, there is something missing which needs to be added. The process of creating a Classification Tree is:

1. Specify the test objective and place it as the root node.
2. Identify the relevant test aspects of the root node. These aspects form the base classifications.
3. Continue this identification process with the new test aspects and expand the tree downwards.
4. The Classification Tree is created when new aspects are considered unnecessary. The leaves are the final classes which are not further divided into sub-classes.
5. The test cases are created by combining the leaves.
6. The minimal number of test cases is determined by counting the number of leaves in the tree. The maximal number is given by multiplying the number of leaves below each base class with each other.

There is no specification of concrete test data in the CTM. The test cases show combinations of classes and describe their content but not the exact data. When each class of a branch in the tree can be assigned an amount of data that is considered to generate a common result that branch is completed. Examples of such classes are negative and positive values. In these cases the sign is considered the only property of interest for the result.

Example: A start value and a length define a range of values. Determine if a given value is within the defined range or not. Only integer numbers are to be considered.

The root node of the tree is the main objective of the problem, i.e. is the given value in the defined range? This node is placed at the top and branches are derived from it. The tree classes beneath the root node in this example are the start point of the range, the length of the range and the position of the value.

To begin with, the tree below is created.

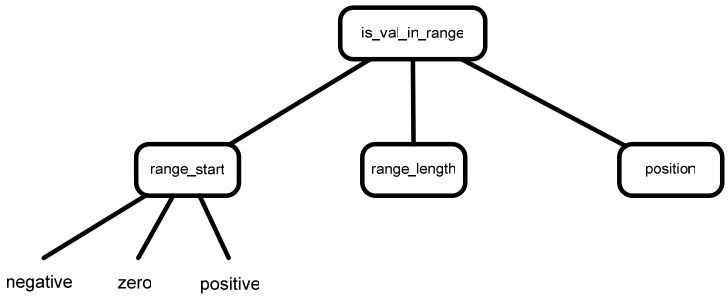


Figure 16: Base of the classification tree

When choosing test cases, boundary values are often of extra interest. These values tend to generate the most malfunctions and are hence the most interesting. To provide more exhaustive testing at the boundaries, the negative and positive values in the tree are divided into subclasses.

The range length is divided into the tree sub-classes negative, zero and positive. A negative range or a range that exceeds the limit of possible values (i.e. the finite limit of an integer data type) are also considered important test cases.

The same approach is applied when choosing test cases for the position of the given value. If the value is outside the range it is interesting to know if the value is on the border or distant from it. A value on the boarder could indicate a small miscalculation.

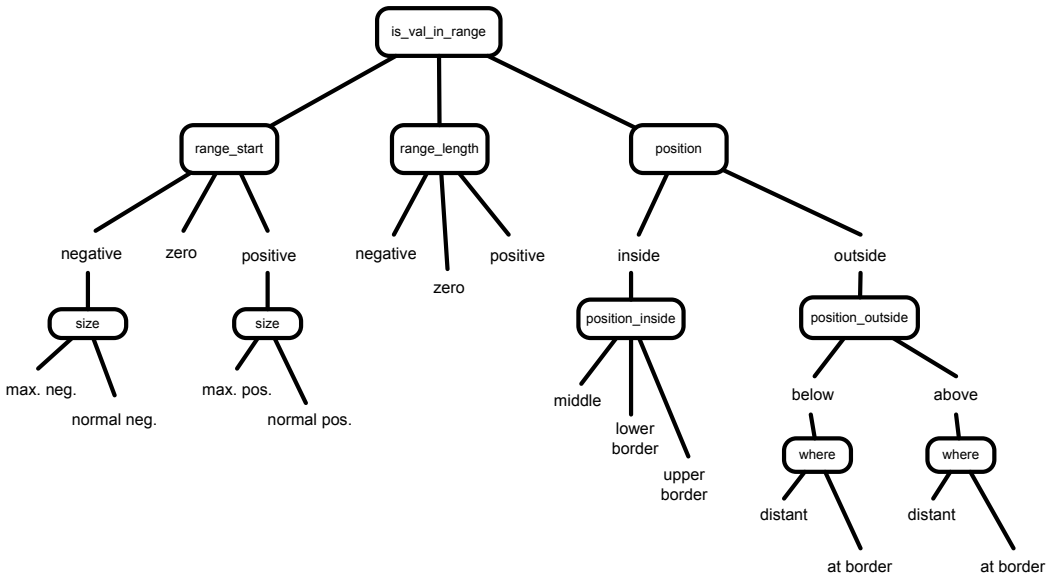


Figure 17: Complete classification tree

References: Mirko Conrad, Ines Fey, Sadegh Sadeghipour, *Systematic Model-Based Testing of Embedded Automotive Software, Electronic Notes in Theoretical Computer Science, SCIENCE DIRECT*

5.12 Structure-based testing

Aim: To apply tests which exercise certain subsets of the program structure.

Description: Based on analysis of the program, a set of input data is chosen so that a large (and often pre-specified target) percentage of the program code is exercised. Measures of code coverage will vary as follows, depending upon the level of rigour required.
– Statements: this is the least rigorous test since it is possible to execute all code statements without exercising both branches of a conditional statement.
– Branches: both sides of every branch should be checked. This may be impractical for some types of defensive code.
– Compound conditions: every condition in a compound conditional branch (i.e. linked by AND/OR) is exercised. See MCDC (modified condition decision coverage, ref. DO178B).
– LCSAJ: a linear code sequence and jump is any linear sequence of code statements, including conditional statements, terminated by a jump. Many potential subpaths will be infeasible due to constraints on the input data imposed by the execution of earlier code.
– Data flow: the execution paths are selected on the basis of data usage; for example, a path where the same variable is both written and read.
– Call graph: a program is composed of subroutines which may be invoked from other subroutines. The call graph is the tree of subroutine invocations in the program. Tests are designed to cover all invocations in the tree.
– Basis path: one of a minimal set of finite paths from start to finish, such that all arcs are included. (Overlapping combinations of paths in this basis set can form any path through that part of the program.) Tests of all basis path has been shown to be efficient for locating errors.

Example: N/a

References: IEC 61508-7, clause B.5.8

5.13 Statistical testing

Aim: To gain a quantitative figure about the reliability properties of the investigated system.

Description: This quantitative figure may take into account the related levels of confidence and significance and can give
– a failure probability per demand;
– a failure probability during a certain period of time; and
– a probability of error containment.
From these figures other parameters may be derived such as:
– probability of failure free execution;
– probability of survival;
– availability;
– MTBF or failure rate; and
– probability of safe execution.
Probabilistic considerations are either based on a probabilistic test or on operating experience. Usually, the number of test cases or observed operating cases is very large. Typically, the testing of the demand mode of operation involves considerably less elapsed time than the continuous mode of operation. Automated testing tools are normally employed to provide test data and supervise test outputs. Large tests are run on large host computers with the appropriate process simulation periphery. Test data is selected both according to systematic and random hardware viewpoints. The overall test control, for example, guarantees a test data profile, while random selection can govern individual test cases in detail. Individual test harnesses, test executions and test supervisions are determined by the detailed test aims as described above. Other important conditions are given by the mathematical prerequisites that must be fulfilled if the test evaluation is to meet its intended test aim. Probabilistic figures about the behaviour of any test object may also be derived from operating experience. Provided the same conditions are met, the same mathematics can be applied as for the evaluation of test results. In practice, it is very difficult to demonstrate ultra-high levels of reliability using these techniques.

Example: N/a

References: IEC 61508-7, clause B.5.3

5.14 Prototyping/animation

Aim: To check the feasibility of implementing the system against the given constraints. To communicate the specifier's interpretation of the system to the customer, in order to locate misunderstandings.

Description: A subset of system functions, constraints, and performance requirements are selected. A prototype is built using high-level tools. At this stage, constraints such as the target computer, implementation language, program size, maintainability, reliability and availability does not have to be considered. The prototype is verified against the customer's criteria and the system requirements may be modified in the light of this verification. This test may be efficiently used as part of the requirements analysis.

Example: N/a

References: IEC 61508-7, clause C.5.17

5.15 Standard test access port and boundary-scan architecture

Aim: To control and observe what happens at each pin of an IC.

Description: Boundary-scan test is an IC design technique which increases the testability of the IC by resolving the problem of how to gain access to the circuit test points within it. In a typical boundary-scan IC, comprised of core logic and input and output buffers, a shift-register stage is placed between the core logic and the input and output buffers adjacent to each IC pin. Each shift-register stage is contained in a boundary-scan cell. The boundary-scan cell can control and observe what happens at each input and output pin of an IC, via the standard test access port. Internal testing of the IC core logic is accomplished by isolating the on-chip core logic from stimuli received from surrounding components, and then performing an internal self-test. These tests can be used to detect failures in the IC.

Example: N/a

References: IEC 61508-7, clause A.2.3

5.16 Behavioural simulation

Aim: To predict the behaviour of a system under different operational conditions without having access to the physical system or the physical equipment controlled by the system.

The functionality of the system hardware, software or target equipment is simulated on a computer via a software behavioural model. The behavioural model response is then studied due to suitable applied input stimulus to the simulation model. The behavioural model is often designed using specific high level software tools. It is however important to keep in mind that the behavioural model will not completely correspond with the actual behaviour of the physical system or equipment.

Example: Using a computer to simulate the target equipment in order to test properties of a prototype control system

References: IEC 61508-7, clause B.3.6

5.17 Symbolic execution

Aim: To verify that the software corresponds with the specification

Description: Symbolic execution is a simulation technique where all input parameters (values/numbers) to the software is replaced with symbols and the value of all program variables are replaced with their corresponding symbolic expressions. The output values resulting from the computation of the program then becomes symbolic expressions formed as functions of the program input symbols. Symbolic execution is a formal technique which also may be simplified in order to use as support software testing.

Example: Annex B – PolySpace for C

References: IEC 61508-7, clause C.5.12
Alberto Coen-Porsini, Giovanni Denaro, Carlo Ghessi, Mauro Pezzé “Using Symbolic execution for verifying Safety-critical Systems”

5.18 Monte-Carlo simulation

Aim: To simulate real world phenomena in software using random numbers.

Description: Monte-Carlo simulations are used to solve two classes of problems:
– probabilistic, where random numbers are used to generate stochastic phenomena; and
– deterministic, which are mathematically translated into an equivalent probabilistic problem. Monte-Carlo simulation injects random number streams to simulate noise on an analysis signal or to add random biases or tolerances. The Monte-Carlo simulation is run to produce a large sample from which statistical results are obtained.
When using Monte-Carlo simulations care must be taken to ensure that the biases, tolerances or noise have reasonable values. A general principle of Monte-Carlo simulations is to restate and reformulate the problem so that the results obtained are as accurate as possible rather than tackling the problem as initially stated.

Example: N/a

References: IEC 61508-7, clause C.6.6

5.19 Fault insertion testing

Aim: To introduce or simulate faults in the system hardware or software and document the response.

Description: This is a qualitative method of assessing the system behaviour at fault. Preferably, detailed functional block, circuit and wiring diagrams are used in order to describe the location and type of fault and how it is introduced; for example: power can be cut from various modules; power, bus or address lines can be open/short-circuited; components or their ports can be opened or shorted; relays can fail to close or open, or do it at the wrong time, etc. In principle, single steady-state faults are introduced. However, in case a fault is not revealed by the built-in diagnostic tests or otherwise does not become evident, it can be left in the system and the effect of a second fault considered. The number of faults can easily increase to hundreds. The work is done by a multidisciplinary team and the vendor of the system should be present and consulted. The mean operating time between failure for faults that have grave consequences should be calculated or estimated. If the calculated time is low, modifications should be made.

Faults may also be injected into software, the response of the system is then studied by functional tests, preferably in conjunction with computer aided tools for monitoring the system behaviour. The failures may be simulated by directly altering the source code and download it into the system or by the means of a subprogram acting as a controllable fault injector. It is very important that a management process is initiated prior to the tests in order to ensure that the fault injections are adequately removed from the software after the test has been performed.

Example: N/a

References: IEC 61508-7, clause B.6.10

5.20 Error seeding

Aim: To ascertain whether a set of test cases is adequate.

Description: Some known types of mistake are inserted (seeded) into the program, and the program is executed with the test cases under test conditions. If only some of the seeded errors are found, the test case set is not adequate. The ratio of found seeded errors to the total number of seeded errors is an estimate of the ratio of found real errors to total number errors. This gives a possibility of estimating the number of remaining errors and thereby the remaining test effort.
errors real of number Total
errors real Found
errors seeded of number Total
errors seeded Found

The detection of all the seeded errors may indicate either that the test case set is adequate, or that the seeded errors were too easy to find. The limitations of the method are that in order to obtain any usable results, the types of mistake as well as the seeding positions must reflect the statistical distribution of real errors.

Example: N/a

References: IEC 61508-7, clause C.5.6

6 Methods regarding formality

6.1 Introduction

The approach here is to present verification and validation techniques from the perspective of formality. To contrast, some general aspects of informal methods are also included. Using an informal method means of course that all aspects of the other chapters are applicable. The main advantage of an informal method is that everyone could understand it without any training (as opposed to formal methods). The main disadvantage is that an informal method leaves room for (subjective) interpretations.

The view here is in principle orthogonal to many other aspects of this report and includes:

- Models; note that formal techniques could be applied to models listed above e.g. finite state models and Petri nets.
- Development issues i.e. how can design and implementation improve and support formal verification and validation
- Levels of formality
- Issues directly addressed by formal methods e.g. creating a formal model

In a project it is necessary to survey the extent and scope of items to be handled from a formal perspective. In principle one has to consider the whole lifecycle of the product when answering the following questions:

- Which parts are candidates? Why? What shall be expressed (e.g. architecture, logical behaviour, real-time aspects etc.)
- What are the reasons for using formal methods? Are there any alternatives?
- What efforts are needed for making effective use of formal methods e.g. concerning training, organisation etc?
- Are there other benefits e.g. carry-over to other projects?

We next have to define the meaning of *formal* and make this by specifying different levels of formal notation:

- by *formal notation* is meant a strict notation with the possibility of making proofs. A proof is the (more or less mechanical) application of logical rules making a deduction from axioms to a conclusion. An important aspect is if manual guidance is needed or if proofs can be made automatically.
- by *rigorous argument* is meant a logical reasoning, possibly as a proof, but not requiring a specific notation e.g. plain English could be acceptable.
- by *semi-formal notation* is meant a strict notation but without the possibility of making proofs.
- by *informal notation* is meant a description given in a common language e.g. English. There are no requirements concerning proof capability. Even though there is no strict notation the quality of a document could be high e.g. for a requirement specification by:
  - structuring the document in blocks, hierarchies etc.
  - correcting ambiguous formulations e.g. replacing *should* with *shall*
  - only stating requirements that are verifiable e.g. replacing *sufficient* with exact values
  - following templates

All four levels could be important complements and one example is the formal language Z where comments in ordinary language (e.g. motivations and explanations) can be (and shall be) interspersed.

Informal methods are not further commented below.

6.2 Means

6.2.1 General

The means described below are *logical proofs*, *model checking, rigorous argument* and *semi-formal method*. These means could be applied for different purposes, e.g. when creating a requirement specification, when making a design etc, and the means are in principle orthogonal to their use.

6.2.2 Logical proofs

Aim: To specify or describe aspects using a formal notation with the possibility of proofs.

Description: By expressing functionality in a suitable formal language verification and validation can be effectively performed e.g. verification of requirements that can take place in an early stage of development i.e. a very cost-effective solution. Aspects could be checked for consistency and proofs could be made. There exists many different languages and a very good reference for a start survey is http://vl.fmnet.info/. Some commonly used languages/methods are:

- B – supports a stepwise and seamless refinement process from specification to source code generation
- Z – a specification language with proof support e.g. using the tool Z/EVES
- CCS (Calculus of Communicating Systems R. Milner), CSP (Communicating Sequential Processes C. A. R. Hoare) – examples of process algebras i.e. considers parallel processes and communication between them
- Temporal logic – a first order logic with the possibility of expressing aspects related in time e.g. “there will always be” and “sometime in the future”.
- Petri net – described above as model. Some aspects that can be analyzed are:
  - Reachability – can a place receive a token?
  - Boundedness – is the number of tokens within limits?
  - Conservation – does the number of tokens remain constant?
  - Liveness – can a transition be fired?
- CTL (Computation Tree Logic) – uses a state representation and can express e.g. “it is possible to reach a state”, “this always hold” etc.

Extensive training is generally needed.

Example: For just giving a flavour of logical proofs a trivial example is shown here. The statement to prove is:

$$\forall x(P(x) \wedge Q(x)) \quad \Rightarrow \quad \forall x P(x) \wedge \forall x Q(x)$$

and the proof proceeds from top to bottom and is:

$$\dfrac{\dfrac{\dfrac{\forall x(P(x) \wedge Q(x))}{P(c) \wedge Q(c)}}{P(c)}}{\forall x P(x)} \qquad \dfrac{\dfrac{\dfrac{\forall x(P(x) \wedge Q(x))}{P(c) \wedge Q(c)}}{Q(c)}}{\forall x Q(x)}$$

$$\dfrac{\forall x P(x) \qquad \forall x Q(x)}{\forall x P(x) \wedge \forall x Q(x)}$$

The first line of the proof is the start of the statement to prove, the second line contains an elimination rule for the free variable x, the third line contains an elimination rule for terms, the fourth line contains an introduction rule and the last line is the end of the statement to prove.

References:

- IEC 61508-7 B.2.3.3, C.2.4.2, C.2.4.3
- http://vl.fmnet.info/

6.2.3 Model checking

Aim: To verify a model with requirements.

Description: Model checking implies both a representation and principles for verifying requirements and model. Model checking uses a state representation when creating a model. Application requirements are set up and they are checked in each state automatically by using a tool. There is a risk of generating an enormous number of states but specific techniques have been developed in order to limit them. Some example tools (commonly used) are given in the references. Some training is needed.

Example: This example is from the SMV reference.

```
MODULE main
VAR
    request : Boolean;
    state : {ready, busy}
ASSIGN
    init(state) := ready;
    next(state) := case
                        state = ready & request : busy;
                        1 : {ready, busy};
                   esac;
SPEC
    AG(request -> AF state = busy)
```

The initial value of *state* is *ready*. The next value is given by the *case* statement. If "state = ready & request" evaluates to true the *busy* value will used otherwise the set {ready, busy} is used i.e. no information gained. The variable *request* could be seen as an input to the program since there is no assignment to it in this example. Under SPEC there is a specification, in CTL notation, that states: if *request* is true then *state* must be *busy*.

References:

- SteP (Stanford Temporal Prover) http://www-step.stanford.edu/
- SMV http://www.cs.cmu.edu/~modelcheck/smv.html
- KRONOS http://www-verimag.imag.fr/TEMPORISE/kronos

6.2.4 Rigorous argument

Aim: To prove properties of the system with mathematical rigor but without requiring a formal notation.

Description: Rigorous argument is also a proof technique but is not applying logical rules in a, more or less, mechanical ways as described by *logical proof* above. Instead this is the type of proof used e.g. by mathematicians where a knowledge is needed in order to understand the reasoning. The proof does not necessarily involve mathematical symbols or described using logic notation. Instead the proof could be made in plain English, however, there should be no questioning of the correctness of the proof i.e. the quality shall not violated. Some training could be needed.

Example: A small example of rigorous argument concerns a tennis tournament. Question: How many matches are played when there are n participants? Answer: For each played match one participant is eliminated and since only the winner remains at the end (n-1) matches are played. Thus, even if the solution is not expressed using a formal notation the logic cannot be questioned.

References:

- http://www.sps.ed.ac.uk/staff/Royal%20Society%20paper.pdf

6.2.5 Semi-formal method

Aim: To specify or describe aspects using a notation without the possibility of proofs.

Description: Semi-formal methods are an alternative when the strictness and proof capability of formal methods are not needed. There could also be other advantages e.g. support from tools and development processes (like UML and SDL). The description can in some cases be analysed and simulated. Semi-formal methods could be a good compromise and also a step towards formal methods when the environment is not mature enough. Some training is needed.

Example: (a possible excerpt from a requirement specification)
:
The control function is defined by A = (B and C) or (D and F) and the function is applicable when current system mode is one of the following {Normal, Power-up, Error, Power-down}
:

References:

- IEC 61508-7 B.2.3
- http://www.uml.org/
- http://www.sdl-forum.org/

6.3 Specification

Aim: To use formal specification techniques to improve specification quality.

Description: The natural use of formal methods is for creating specifications e.g. concerning functional and temporal behaviour, architecture, design, tests etc. The specification can be checked in one or more of the following ways:

- Check of consistency i.e. that there are no contradicting aspects.
- Check of completeness. This can be made by adding new requirements (perhaps implicit or assumed to hold) and check if they can be derived or if they are new.

The natural approach is to use logical proofs but rigorous argument and semi-formal methods could be used, however, the quality might be lower.

Example: (this example is from Z reference manual http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.pdf)

$\cup$_ *BirthdayBook* _____
$\rightarrow$*known* : Π *NAME*
$\rightarrow$*birthday* : *NAME* $\clubsuit$ *DATE*
$\cap$_____
$\rightarrow$*known* = dom *birthday*
$\angle$_____

The purpose of this example is to specify a book containing names and associated birthdays. Just the start is shown above. The construct *BirthdayBook* is a *schema* and contains:

- the set *known* which belongs to the power set of *NAMES* i.e. any set of names.
- the function *birthday* that takes a *NAME* and returns a *DATE*. The arrow shows that a *NAME* corresponds to exactly one *DATE*.
- the last line which is an invariant and states that the domain of *birthday* always is the set *known*. The idea is that the set *known* contains all names which have an associated birthday and thus the function *birthday* could be applied for these names (several other constructs are needed but not shown here).

References:

- IEC 61508-7 B.2.4.1
- IEC 61508-7 B.2.2

6.4 Check of model

Aim: To create a formal model for verification and design.

Description: One use of formal methods is to create a formal model that is used in one or more of the following ways:

- Check of consistency i.e. that there are no contradicting aspects.
- Check of requirements i.e. that there are no cases when requirements are not fulfilled. Some examples are no deadlock, liveness, no safety violation etc.
- Check of completeness (challenging the model). This can be made by adding new requirements (perhaps implicit or assumed to hold) and check if they are fulfilled. To be of value the model should be considered complete before check of completeness takes place otherwise there is a risk of self-fulfilment i.e. the model is adjusted as much in order to fulfil the new requirements. Since finding

requirements can be a “brain-storm” related activity an N-version handling could be of value i.e. a number of independent teams figuring out these requirements. The natural approach is to use logical proofs or model checking but rigorous argument and semi-formal methods could be used, however, the quality might be lower.

References:

- IEC 61508-7 B.2.2

6.5 Assertions

Aim: To prove the correctness of a function.

Description: The idea is to specify pre-conditions and post-conditions (assertions) of a function and to prove that the pre-conditions are transferred to the post-conditions by a number of rules and that the handling terminates. The function could be implemented by software or by a number of well-defined sub-functions thus enabling a hierarchical approach. This technique is natural when only selected parts of the system will be thoroughly evaluated. Logical proofs, model checking, rigorous argument and semi-formal method could all be considered for this technique.

References:

- IEC 61508-7 C.5.13
- IEC 61508-7 B.2.2

6.6 Software generation

Aim: To transfer a formal model into source code.

Description: One use of formal methods is to create a formal model that is used as a high level design that can be refined in steps down to implementation possibly maintaining the same quality. One such suitable method is the B method that supports a seamless development chain i.e. the proved properties of the model are valid in all steps down to implementation. In principle logical proofs, model checking, rigorous argument and semi-formal method could all be considered for this technique but for seamless transforms logical proofs is the natural candidate.

References:

- IEC 61508-7 B.2.2

6.7 Automatic test case generation

Aim: To generate test cases from a formal specification.

Description: Normally it takes a lot of effort and resources in order to create suitable test cases. The idea here is to let a tool automatically generate test cases from a functional requirement specification expressed using a formal language. In this way large savings can be made. Since the specification normally does not contain implementation aspects the test cases are more related to a “black-box” view of the system and thus the tests will be mainly used for integration, system tests and for validation.

References:

- http://www.cis.ksu.edu/~kcrea/cis841/

7 Development methods

7.1 Introduction

Since quality cannot be created by extensive testing (since not everything can be tested) it is necessary to have quality built-in in previous phases of the development process. This is the motivation for considering development methods for verification and validation i.e. in what ways can development methods:

- decrease the extent, the number of requirements and need for resource at verification and validation
- simplify, improve and support verification and validation

There exist many development methods and corresponding processes often related to commercial tools and/or standard notations; two examples are UML and SDL. Also there are many general methods and principles that improve verification and validation but are not directly related to specific processes (but normally included in them).

Use of more or less formal methods was described in a separate chapter above and is not further commented here, instead everything is considered informal here.

7.2 Means

7.2.1 General

The means described below are *separation of concern*, *hierarchies*, *dependencies* and *use of standards*. These means could be applied for different purposes, e.g. when creating a requirement specification, when making a design etc, and the means are in principle orthogonal to their use. The means could also be combined e.g. use of both *separation of concern* and *hierarchies* together.

7.2.2 Separation of concern

Aim: To separate aspects in order to make them easier to handle.

Description: In complex systems it is necessary to make partitions. This concerns all phases of development but also operation and maintenance i.e. the whole life cycle of the product. Separation of concerns applies to software, hardware and various forms of documents and is related to *encapsulation*. Separation of aspects can be made from different perspectives, e.g. from

- structure perspective: separation into modules or subsystems
- dependability perspective: separation of safety-related and non-safety-related parts, separation of redundant parts (for design and implementation but also for n-version handling)
- functional perspective: separation into functional blocks

By separating aspects it is easier to identify independent parts and also to specify interfaces and the types of dependence. This implies easier design, implementation and evaluation. Further, testing is simplified and the quality can be improved since the separated parts can be tested individually before being assembled.

References:

- IEC 61508-7 B.1.3
- IEC 61508-7 C.2.9

7.2.3 Hierarchies

Aim: To group aspects in hierarchies in order to make aspects easier to handle.

Description: As opposed to *separation of concern* hierarchies use layers for simplifying complex systems. Each layer corresponds to a level of abstraction i.e. information concerning lower layers is not visible at the current layer. Hierarchies can be applied for all phases of development but also for operation and maintenance i.e. the whole life cycle of the product. Hierarchies apply to software, hardware and various forms of documents concerning e.g. structure, functionality, implementation etc. By using hierarchies it is easier to design, implement, test and evaluate systems. Further, incremental development is supported since details (in lower layers) could be specified later on without affecting the higher layers. At implementation a lower layer can be built first and verified before continuing to the next higher layer making it possible to always have a stable basis to work with.

References:
- IEC 61508-7 B.2.4.3

7.2.4 Dependencies

Aim: To describe the system or parts of it as groups of data with associations between them.

Description: This idea is related to *separation of concern* as described above. For parts of design and implementation the *entity* (also called *conceptual*) structure describes groups of data (entities) and how these entities are related. The structure is used for a specific level of abstraction and the corresponding data level of abstraction is used. A visual presentation is made in an entity relationship diagram (ERD). For each entity attributes can be included. Apart from describing each relation there are also associated number relations e.g. one-to-many, one-to-one etc.

Corresponding information is also given in the UML class diagram.

References:
- IEC 61508-7 B.2.4.4
- http://www.winnetmag.com/Articles/Print.cfm?ArticleID=8589

7.2.5 Use of standards

Aim: To enforce a common way of performing tasks verifiability, to encourage a team-centred, objective approach and to enforce a standard design method.

Description: Standards can be applied for different levels of scope, detail and strictness. Some examples are development standards (e.g. IEC 61508), coding standards (how to implement software), testing standards, application sector standards etc. Standards could be official, company related, group related etc. Checklists and templates could be created to support the introduction of standards. The use of standards:
- Improves quality by giving rules how to handle different aspects.
- Improves quality by making all involved people following the same principles. A new project builds on previous projects since e.g. people know their roles and the associated tasks.
- Supports the possibility of providing proofs for the achieved quality.
- Needs organisation support (especially from management).

Thus standards are made to allow for ease of and improved development, verification, assessment and maintenance. Therefore they should take into account available tools.

References:

- IEC 61508-7 C.2.6.1, C.2.6.2

7.3 Requirements

7.3.1 Partial requirements

Aim: To reduce complexity by creating a hierarchical structure of partial requirements.

Description: This technique structures the functional specification into partial requirements, i.e. requirements that have to refined later, such that the simplest possible, visible relations exist between the partial requirements. This procedure is successively refined until sufficiently small clear partial requirements can be distinguished. The result is a hierarchical structure of partial requirements which provide a framework for the specification of the complete requirements. This method emphasises the interfaces of the partial requirements and is particularly effective for avoiding interface failures.

References:

- IEC 61508-7 B.2.1

7.3.2 Use of system

Aim: To specify the functional behaviour as seen by the user of the system.

Description: The idea is to keep the interior of the system invisible i.e. to consider it a "black box". Instead the behaviour is defined by the inputs and outputs of the system and not considering any internal states. Thus the perspective is from the functional use of the user of the system i.e. what the user sees. Properties have to be added e.g. concerning response time, security aspects etc. *Use cases* and corresponding *scenarios* (as defined by UML) are normally sufficient for describing the overall use of system and more than approximately fifteen *use cases* shall not be needed (otherwise they have been defined at a too detailed level). A use case is a group of related scenarios which describe specific interaction sequences with the system. Since many combinations are possible there is a big risk that there will be an "explosion" of scenarios and special considerations have to be taken in order to limit them. Some typical characteristics of *uses cases* and *scenarios* are:

- A scenario can include pre-conditions, sequence of actions, post-conditions, exceptions, properties and descriptions.
- Use cases and scenarios are directly understandable by anyone.
- Use cases and scenarios are especially effective for requirement specification and validation but also important during design and implementation.

Note that *uses cases* are functional and UML handles object-oriented design. Thus, somewhere, when *use cases* and *scenarios* are more and more detailed, one has to change perspective from functional to object-oriented handling.

References:

- http://www.uml.org/

7.4 Design and implementation

7.4.1 Structure

Aim: To describe the design and implementation structure of the system.

Description: A design structure is static and contains a description of how the high level implementation is made. The implementation structure is a refinement of the design structure. The overall structure normally contains several structures, created from different perspectives, and describes how the application is constructed. The structures may concern:

- Use of trusted components (software and hardware)
- Definition of software components (in house developed or COTS) and their mapping to physical components (called component and deployment diagrams in UML)
- Graceful degradation principles
- Error propagation and confinement principles
- Reconfiguration principles

For object-oriented design there are also:

- Class structures and objects created from them (called class and component diagrams in UML)
- Packages (UML specific)
- Inheritance hierarchies

For real-time applications there are also:

- Mapping of functions and software components to processes
- Definition of communication between processes

The purpose of the structures is to get an overview and enable analyses e.g. concerning data flow, worst case execution time, dependability etc.

References:

- http://www.uml.org/
- IEC 61508-7 B.3.2, B.3.4, C.2.1, C.2.7, C.2.8, C.2.9, C.2.10, C.3.11, C.3.13, C.4.5
- http://bdn.borland.com/article/0,1410,31863,00.html
- http://www.sintef.no/time/elb40/html/elb/sdl/sdl_t01.htm

7.4.2 Behaviour

Aim: To describe the system behaviour as a sequence of messages.

Description: The idea is to describe how different parts cooperate by considering the messages sent from one part to another according to time. Both SDL and UML specify this idea. In SDL it is called *message sequence chart* and in UML *sequence diagram* (*collaboration diagram* also exists in UML with the same content but expressed differently). In the UML *sequence diagram* objects are placed horizontally with the time axis vertically. Transfers of messages are shown, in the correct sequence of time, as arrows from sender object to receiver object. The connection between *use case* and *sequence diagram* and the latter can be seen as a refinement of the former. Apart from analysis the *sequence diagram* could also be used for trouble shooting and test case generation.

References:

- http://www.sdl-forum.org/MSC/index.htm
- http://www.uml.org/

7.5 Process

7.5.1 Administration

Aim: To manage the development process in a controlled way.

Description: To handle the development of a product in a controlled way is of course the basis for producing dependable products and to ease verification and validation. Without a controlled and documented development process verification and validation will be of limited value since:

- Everything cannot be verified and validated, instead quality has to be built-in in all phases of the life cycle of the product.
- The quality cannot be proved.

Apart from the development process administration concerns the company organisation, maturity and quality standards and more specifically:

- Project management – adoption of an organisational model, rules and measures for the whole life cycle of a product.
- Documentation – the documentation of each phase of the life cycle of the product and thus the corresponding input and output documents for each phase. This includes data, decisions and rationale in order to allow for easier verification, validation and assessment.
- Configuration management – handling versions, changes and releases.

References:

- IEC 61508-7 B.1.1, B.1.2, C.5.2

7.5.2 Established process

Aim: To improve quality by following an established development process.

Description: The purpose of following a development process concerns two aspects:

- to increase quality
- to prove the quality

The latter point is important e.g. at certification, investigations after an accident etc. To follow an established has further advantages:

- tool support exists
- experience exchange with others is possible
- documentation, courses and training are available
- updates (improvements) are continuously made
- short start-up time before full use

Two examples of established processes are Cleanroom and Rational RUP (see references).

References:

- http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr022.96.pdf
- http://www-306.ibm.com/software/awdtools/rup/support/index.html
- IEC 61508-7 C.2.1

7.5.3 Tool support

Aim: To improve quality and performance by using established tools.

Description: In order to handle complex and extensive applications tool support is generally needed. Tools can be used for:

- Modelling
- Design
- Implementation e.g. debugger, compiler etc.
- Verification and validation e.g. automatic test case generation
- Administration e.g. version handling

There is a trend towards more and more automatically generated code e.g. from graphical representations such as state diagrams or from high level languages. There is also a trend towards model based development i.e. a (single) model contains all information and changing one part or view implies an automatic update of dependent aspects. Tools exist that can be connected creating a consistent whole e.g. a change of a requirement will be signalled in design, implementation and test cases by other tools. Since the dependence on the quality of the tool is high it is necessary to evaluate if the tool can be trusted and there are in principle two ways this can be made:

- by certification – the tool is then evaluated by a an independent company.
- by “proven in use” – the tool is considered reliable due to its extensive use

References:

- IEC 61508-7 B.3.5

7.5.4 Reuse

Aim: To reuse components that already have been verified and validated.

Description: The idea is to have some kind of library with previously developed and verified application related components. The extreme of this is to have a fully developed *component based development* where one group of the company is responsible to administrate and develop components and another group connects subsets of components generating products. The crucial aspect is the specification of interfaces. A reused component can be in-house developed or commercial (COTS). In the ideal case the reused component does not have to be verified at all and focus can be set on newly developed parts instead, however, in practise this is not recommended due to the following possible reasons:

- the component was verified and validated with respect to its previous role and features may have been overlooked or been considered as irrelevant
- the new use of the component is not the same as the old use
- the component may have side effects and implicit dependencies not realized previously

Thus the situation is more complex than first imagined, however, a cautious reuse have the potential of saving substantial efforts and improve quality since resources could be spent on other (new) parts instead.

References:

- -

7.5.5 Skill

Aim: To improve capability of involved persons.

Description: Apart from purely technical aspects the capability and motivation of persons involved strongly affect the quality of verification and validation. Some examples of persons involved are:

- those using the development process
- those responsible for quality
- those responsible for the development process
- management

Related aspects are:

- organisation of company, project groups etc.
- education and training e.g. what shall be learned, how often repeated etc.
- learning from experience e.g. lessons learned, evaluation of previous projects
- the understanding of the users e.g. how they react, what they can etc.

References:

- -

7.5.6 Design for testability

Aim: To consider testing aspects in all phases of development.

Description: The idea originated from hardware design (e.g. use of boundary scan) but software design is also included. One example concerning software is design by *software contract* which uses assertions for functions for input (pre-conditions) and output (post-conditions) parameters. *Design for testability* (the same purpose as *test driven development*) must be considered right from the beginning i.e. when specifying requirements, making the design and implementation. The general principle is to improve the visibility and controllability of internal states and interfaces. The latter coincides with encapsulation and object-oriented approaches. Encapsulation, which can also be used in not object-oriented systems, means a separation of concerns, i.e. a more independent structure, and thus makes the system easier to verify (however, there could be increased problems for verifying the interior of the encapsulated unit). Other aspects of *design for testability* are also applicable e.g. direct inclusion of tests in the system and transmission of status information at various points. One example of *design for testability* could be to construct the HMI part of a system as an absolute minimum of functionality and complexity (since manual testing is needed) and to test the interface to the rest of the system using automatic testing. The interface could consist of input and output tokens and the HMI part will then be very stable and easy to test.

References:

- http://www.cs.colostate.edu/~rta/publications/ootestability.pdf

Annex A: References

[ADELARD]
Homepage of Adelard , www.adelard.co.uk

[DEFSTAN00-56]
Ministry of Defence Interim Defence Standard 00-56
Safety Management Requirements for Defence Systems
Issue 3 Publication Date 17 December 2004
[Download from www.dstan.mod.uk]

[ECE-R13]
E/ECE/324, E/ECE/TRANS/505
United Nations Regulation No. 13
UNIFORM PROVISIONS CONCERNING THE APPROVAL OF VEHICLES
OF CATEGORIES M, N AND O WITH REGARD TO BRAKING
Annex 18, Special requirements to be applied to the safety aspects of complex electronic vehicle control systems

[ECE-R79]
E/ECE/324, E7ECE/TRANS/505
United Nations Regulations No.79
UNIFORM PROVISIONS CONCERNING THE APPROVAL OF VEHICLES WITH REGARD TO STEERING EQUIPMENT

[IEC60812]
International standard IEC 60812
"Analysis techniques for system reliability –
Procedure for failure mode and effects analysis"
Possible to order at www.iec.ch

[IEC61025]
International standard IEC 61025
"Fault tree analysis"
Possible to order at www.iec.ch

[IEC61508]
International standard IEC 61508, part 1- 7
"Functional safety of electrical/electronic/programmable electronic safety-related systems"
Possible to order at www.iec.ch , More information at www.iec.ch/functionalsafety

[GRRF/2003/27]
TRANS/WP.29/GRRF/2003/27 Proposal for a new draft regulation uniform technical prescriptions concerning the approval of complex electronic control systems affecting the direct vehicle control by the driver (Download at
http://www.unece.org/trans/main/welcwp29.htm)

[Kelly04]
Kelly, Tim "Defining the safety case concept", The Safety-Critical Systems Club Newsletter, September 2004 (more information of SCSC at http://www.safety-club.org.uk/)

[MISRAcontr]
MISRA Technical Report
Hazard Classification for Moving Vehicle Hazards, Controllability
Version1, May 2004 (Download from www.misra.org.uk)

Annex B: PolySpace for C

This annex gives an overview of PolySpace - a tool for static analysis of source code using symbolic execution. The tool performs analysis of source code written in Ada, C and C++. This annex is focused on the support for the programming language C. A short description of the algorithm behind the tool as well its purpose is presented. The recommended methodology for using PolySpace and some guidelines for a selection of the pre-analysis configurations are also given in a short form.

B.1 Introduction

PolySpace is one of many tools for static analysis of source code. Tools in this domain examine code looking for errors without executing the binary built from it. The level of sophistication among these tools ranges from verification of individual statements and declarations to the complete source code of programs. More comprehensive analysis is necessary to reveal errors caused by dependencies between different parts of code.

In the wide range of tools for static verification PolySpace is found among the more advanced and utilise a technique called symbolic execution. Its method is based on data-flow analysis where dynamic properties are statically verified. PolySpace uses an approximation of the source code and transforms it into mathematical equations expressing its different states. The approximate solution is a superset of the exact and thus excludes no information. A graphic illustration is given in the Example 1 contained in the next chapter.

The use of mathematical equations to describe software allows the results to hold during any execution. The analysis is performed using all possible input values and is therefore valid for all inputs. This is perhaps the main difference compared to run-time debugging where test cases with predefined data are used.

B.2 Area of application and technique

The purpose of PolySpace is to detect run-time errors by analysing the dynamic properties of software. It is not to be mixed up with static verification tools that focus on the syntax of source code. Run-time errors arise when the program is executed and include among others, out-of-bounds array indexing and access to non-initialised variables. A complete list of the run-time error checks performed by PolySpace is given below:

Table 4: Checks performed by PolySpace

Nr	Description	Short
1	Illegal pointer access to variable or structure field	IDP
2	Array conversion must not extend range	COR
3	Array index within bounds	OBAI
4	Initialized Return Value	IRV
5	Non-Initialized Variable	NIV
6	Non-Initialized Pointer	NIP
7	Power arithmetic	POW
8	User Assertion	ASRT
9	Scalar and Float Underflows	UNFL

10	Scalar and Float Overflows	OVFL
11	Scalar or Float Division by zero	ZDV
12	Shift amount not exceeding bit size	SHF
13	Left operand of left shift is negative	SHF
14	Function pointer must point to a valid function	COR
15	Wrong type for argument	COR
16	Wrong number of arguments	COR
17	Pointer within bounds	IDP
18	Non Termination of Call or Loop	NTC, NTL
19	Unreachable Code	UNR
20	Value On Assignment	VOA
21	Inspection Points	IPT

Not all of the checks reveal certain run-time errors. The checks *User Assertion* (ASRT), *Non Termination of Call or Loop* (NTC, NTL), *Unreachable Code* (UNR), *Value On Assignment* (VOA) and *Inspection Point* (IPT) reveal no errors but provides useful information. Check number eight, ASRT, is applied when PolySpace encounters a user defined assertion in the code. Assertions are added prior to analysis by the user as a line of code in the source code to verify a property. If the property is false when PolySpace performs the ASRT check the assertion will be treated as a run-time error. Like the ASRT check the *Non Termination of Call or Loop* does not indicate a certain error. If the result of a PolySpace analysis presents this check as an error it simply means that the call or loop never terminates. The purpose of check number nineteen, UNR, is quite self-explaining but its relation to dead code is perhaps not obvious. Unreachable or dead code is presented in grey colour in the result. The complete number of greys is not equivalent to the number of dead lines of code and the two are not comparable. The colour grey indicates that an assignment, return, conditional branch or function call is not reached. This is not the same as a dead line of code where the complete unexecuted line is said to be dead. In Example 2 below, it is illustrated how PolySpace deals with dead code. Apart from the clear difference between grey code and dead lines of code, the example also shown that dead code is unchecked code. The final two checks in the table *Value on Assignment* and *Inspection Points* present the range or value of variables. The first, VOA, establishes the range or value of variables on assignment and compared to IPT, it requires no modification of the source code. The limitation of VOA is that an assignment is required to reveal the information. In order to obtain the range or value of a variable at an arbitrary point in the code, a pragma that triggers the IPT check can be added. This pragma instructs PolySpace to establish the range or value on that line in the code.

In addition to the detection of run-time errors, PolySpace also include a MISRA-C checker. This module verifies that the coding rules of the MISRA-C: 2004 guidelines are respected. The guidelines are a combination of 141 advisory and required rules. PolySpace has full support for 101, partial support for 20 and no support for 20 of these rules.

Example 1

This is an example of an analysis of the equation $x = \frac{x}{x-y}$ which is part of a program.

The equation contains two variables and is therefore two-dimensional. Additional variables will not affect the procedure of analysis other then adding dimensions to it.

The possible errors are identified as; x and y can be non-initialised, division-by-zero can occur if $x = y$, the subtraction $x - y$ can overflow and the division $\frac{x}{x-y}$ can overflow.

This example focuses on the possible occurrence of a division-by-zero error. The values of x and y causing the denominator to become zero are illustrated as the line in the diagram (Figure 18). The dots in the same diagram indicates the values that x and y can take in the analysed software.

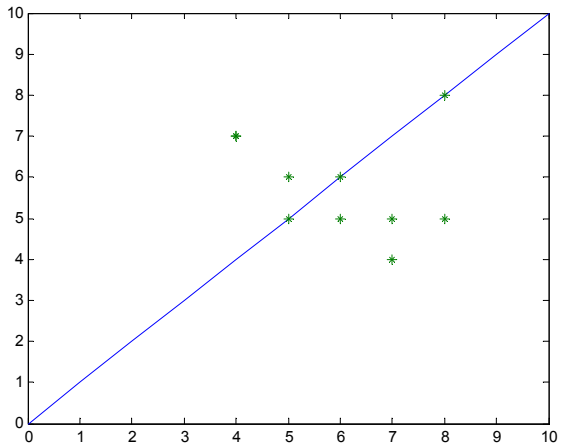


Figure 18: The line y=x and the possible values of x and y

Because this is a very simple example it is trivial to find that x and y equal to five, six or eight cause an error. With a more complex problem for example with additional dimensions another approach is necessary. The method used by PolySpace involves creation of areas defining zones which replaces the dots. A rectangle containing all the values is the simplest possible area. It is defined by (xmin, ymin) and (xmax, ymax) and is used instead of the individual possible values that x and y can take (represented by dots in Figure 19).

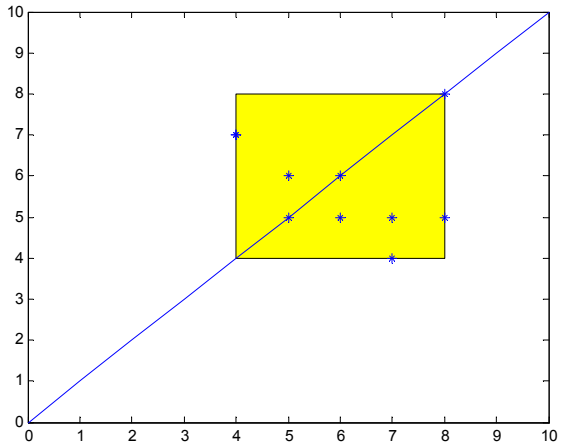


Figure 19: Zone to replace individual dots

It is necessary to refine the areas since a rectangle will give many false alarms. As can be seen in figure 19 the zone captures much more then just the possible values. PolySpace therefore split the single rectangular area into smaller parts. To further improve the analysis the zones can take the form of polyhedrons and lattices. The final result can look like the illustration in figure 20.

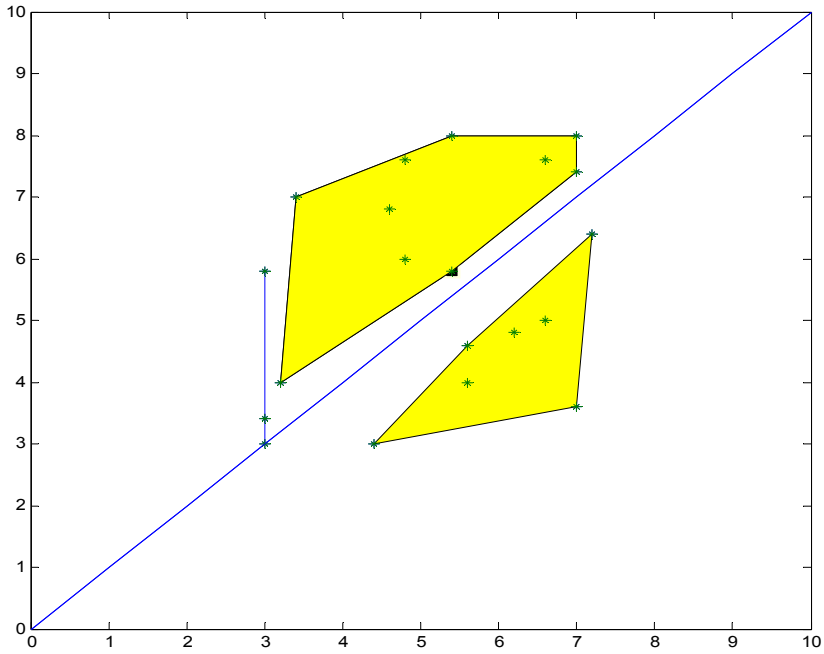


Figure 20: Illustration of polyhedrons and lattices

Now the areas are divided and adjusted to fit the values more precise. Instead of including a single large zone the areas are reduced to polyhedrons and lattices. When an analysis is performed on a refined approximation like the one in figure 20 fewer false alarms are given compared to the earlier analysis on a single rectangle.

The calculation of polyhedrons and lattices require some computer power and results in a trade-off between performance and accuracy. Advanced calculations take time but provide fewer false alarms. It is up to user to choose the level of precision for the analysis but it should be remembered that many false alarms might be distracting when interpreting the results.

Example 2
An illustration of how PolySpace present dead code. The example contains the declaration of two Boolean variables which are later used in an if-statement.

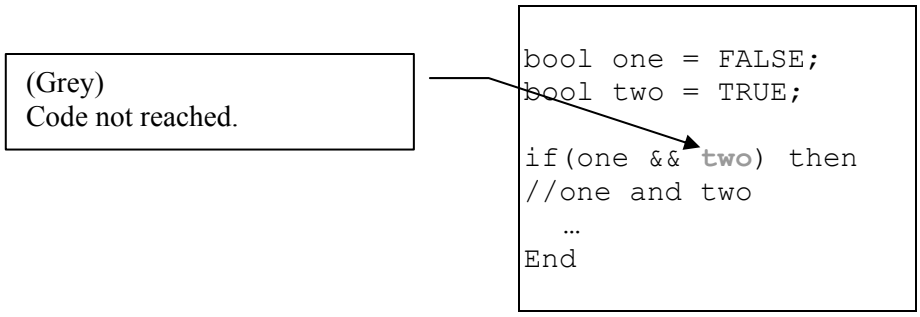


Figure 21: Example of result from analysis with grey code in a conditional branch

The listing above show grey code in a conditional branch. Because the first operand in the if-statement is FALSE the second need not be evaluated to establish the condition of the expression. Therefore the second part of the condition is never executed and the variable *two* is coloured grey.

B.3 Setting up an analysis

To minimize different behaviour between the software submitted to analysis and the one used in the final product, the source code should be identical in the two cases. In practice this is not always achievable since it is often necessary to add some lines of code before the analysis. Source code containing assembler instructions is one such example. Since PolySpace has no support for assembler, the instructions have to be removed or replaced with c code that implements the functionality. A similar situation appears when function-calls are made to external software that is not part of the analysis. The process of adding software to replace the functionality of assembler instructions and missing functions is called stubbing.

B.3.1 Stubbing

When functions outside the scope of the analysis are called or if assembler instructions are embedded in the code it is necessary to introduce stubbing. Stubbing allows the analysis to take into account the possible effect that the external code or assembler instructions would have on the analysis.

Two approaches can be taken when performing stubbing. Either the added code implements the functionality of the code it is replacing or it returns the full range of all variables it is associated with. The first of the two techniques require knowledge of the substituted code. The basic idea is that the user writes a new function with full or partial implementation of the functionality it is replacing. The second approach only requires the function header with the arguments and prototype. With this limited amount of information a different tactic is applied. To assure that no possible values are left out of the analysis all variables assumed to be affected by the function are given the full range of their data type. Compared to the first techniques this type of stubbing is often applicable and a sufficient solution. One disadvantage of it is that the analysis could become unnecessarily extensive and time-consuming. When the ranges of variables exceed the boundaries present during non stubbed execution, irrelevant values are included in the analysis.

The functions manually stubbed by the user are included in the analysis together with the rest of the source code. If a function is missing in the code submitted for analysis, PolySpace will try to automatically stub it. The function created by PolySpace applies the second of the two techniques described above. It returns the full range of the data type declared as the function's prototype. If the address to a variable is passed as an argument to the function, this variable will also be assigned the full range of its data type. See the third column of table 5 in example 3 for further explanation.

B.3.2 Multitasking

To allow an analysis of multitasking code that involves interrupts, there are some requirements that have to be considered. Two primary requirements concern the configuration of the code.

1. The main procedure must terminate before an entry-point (access point of a task) can be started.
2. All entry-points must be started with no predefined basis regarding sequence, priority or pre-emption.

If the two requirements are not met, some modifications of the source code will solve the issue. The code will fail on the first point if it contains an infinite loop. Real-time systems with a *while(1)* loop is one such example. The matter is solved by simply removing the iteration.

The second requirement only applies to the entry-point itself. Sequential or pre-emptive multitasking is therefore possible if the functionality is encapsulated inside a new function. The new function implements the correct behaviour and makes the calls to the tasks. These functions then replace the individual tasks as entry-points. See example 4 for an illustration.

Example 3

This example demonstrates the differences between the available stubbing methods.

The function `missing_function` needs to be stubbed. It has two arguments, one pointer `*x` and one variable `y`. The function does not return any value but should write the value of `y` to the pointer `x`.

The second column shows the effects on a function where the functionality is ignored. Without proper function stubbing the analysis will not cover all possible states of the software.

Table 5: Affects of different stubbing techniques

	Functionality ignored	Manual stubbing	Automatic stubbing
Source code	void missing_function (int *x, int y) { }	void missing_function (int *x, int y) { *x = y; }	Generated by PolySpace
Affect on the two arguments	A call to missing_function has no affect on either argument.	A call to missing_function will give the first argument the value of the second. The second argument will stay unaffected	A call to missing_function will give the first argument the full range defined for the data type *int*. The second argument will stay unaffected.

Example 4

This example describes the configuration required for two cases of multitasking code.

Both cases involves three tasks; task1, task2 and task3.

Case 1 – Single entry-point:
The three tasks are sequential and operated on the same sampling rate without the ability to pre-empt each other. The single entry-point shown in table 5 will instruct PolySpace to accurately implement this behaviour. The entry-point calls the tasks in sequential order one after the other. This way both the order and pre-emptive behaviour is ensured.

Case 2 – Multiple entry-points
There is no relative order between the three tasks in this example. The only relative behaviour is the restriction that task1 and task2 can not pre-empt each other. But task1 and task3 as well as task2 and task3 can pre-empt each other. The solution to this configuration is to place task1 and task2 under the same entry-point and task3 on its own under a second entry-point. This allows the tasks to pre-empt each other according to the specification. Finally, the if-statements ensure the non deterministic relative order of execution.

Case 1 - Single entry-point

```
void main()
{
…
}

entry-point1()
{
  while(1)
  {
    task1();
    task2();
    task3();
  }
}
```

Case 2 - Multiple entry-points

```
void main()
{
…
}

entry-point1()
{
  while(1)
  {
    if(random) task1();
    if(random) task2();
  }
}

entry-point2()
{
  while(1)
    if(random) task3();
}
```

Figure 22: Configuration of entry-points

B.4 Reviewing results

The results of an analysis are automatically presented by PolySpace as shown in figure 22. All checks performed by PolySpace are listed below each analysed file. By selecting a file or a check the code relating to it is displayed in a separate window. Other features of this interface include a call tree and a list of read and write accesses to global variables. This interactive presentation is the recommended method to review the results of an analysis. An alternative approach is to use the excel-macro to generate a Microsoft Excel spreadsheet with tables and diagrams of the results. Compared to the first this approach allows for easy export of data and does not require the PolySpace software for reviewing.

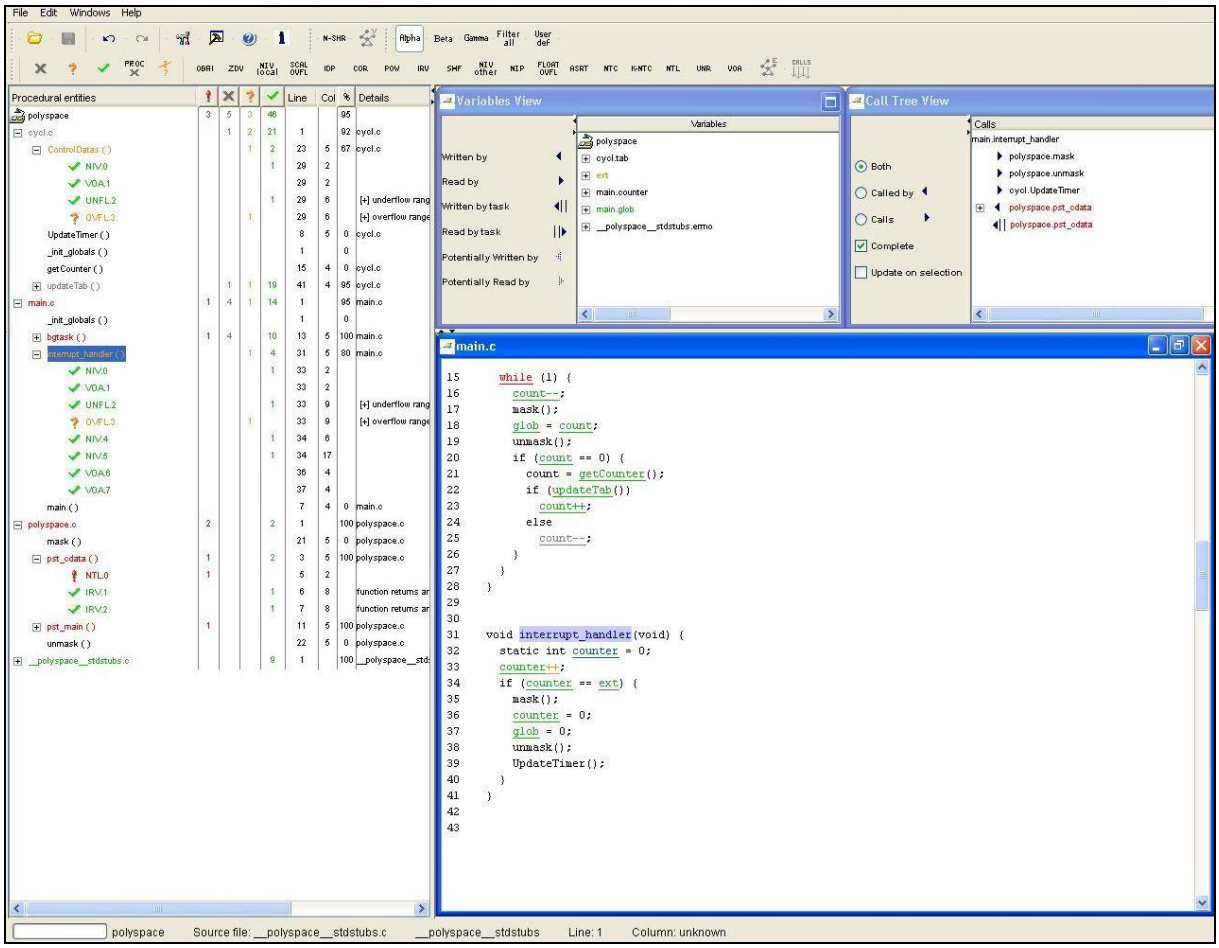


Figure 23: Screenshot of GUI presenting results from analysis

B.5 Methodology

The intended benefits from using PolySpace are described in the manual as; "improvement of the software quality" and "reduction of testing and validation costs". These two phrases are created assuming that the primary goals are to maximise productivity while minimising quality defects. To concurrently achieve both of these goals a methodology is included in the manual. The suggested approach with focus on its fundamental aspects is given below.

If to much focus is concentrated to one of the desirables productivity or quality defects the other can be negatively affected. The solution suggested in the manual for achieving high software quality at a minimal cost is based on selective debugging. By centring the effort spent on improving the software to specific areas a maximum number of errors can be found.

To efficiently review the result of a PolySpace analysis it is strongly recommended that a working procedure is defined. PolySpace use the colours red, grey, orange and green to indicate the result of each check. The three colours red, grey and orange signal an error while green means that the operator or operand passed the check. An example of the different colours is given in listing 2.

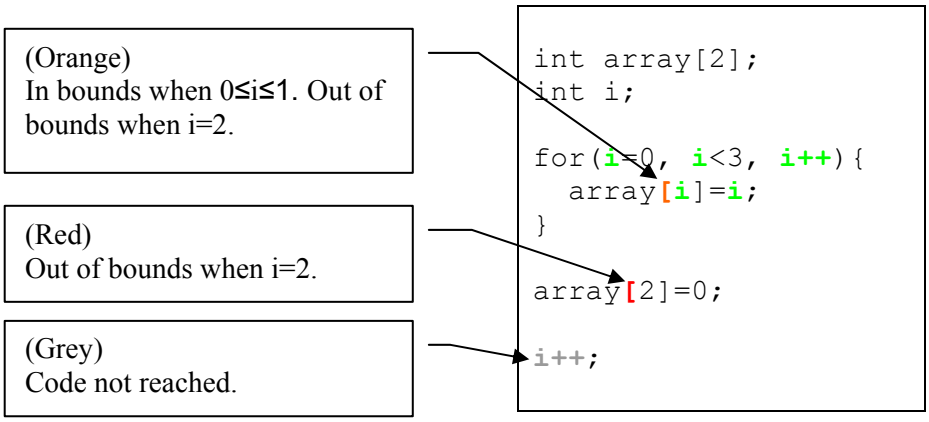


Figure 24: Example of result from PolySpace analysis

The manual suggests an order of priority for this colour scheme. (Red has the highest priority and orange the lowest)

1. Red (certain error)
2. Grey (dead code)
3. Orange (warning)

Red code is given the highest priority partly because it indicates a line of code where an error is generated regardless. Another reason for giving red errors extra attention is that code subsequent of it is not analysed. Unanalysed code can contain errors not revealed by PolySpace. This is also the reason why the second highest priority is given to grey code.

The orange colour indicates code that depending on the inputs will execute correctly for some values and generate an error for other values. These warnings can be difficult to interpret and should to start with, only be given a limited amount of attention. If time permits extra thought can be dedicated to the more complex orange warnings.

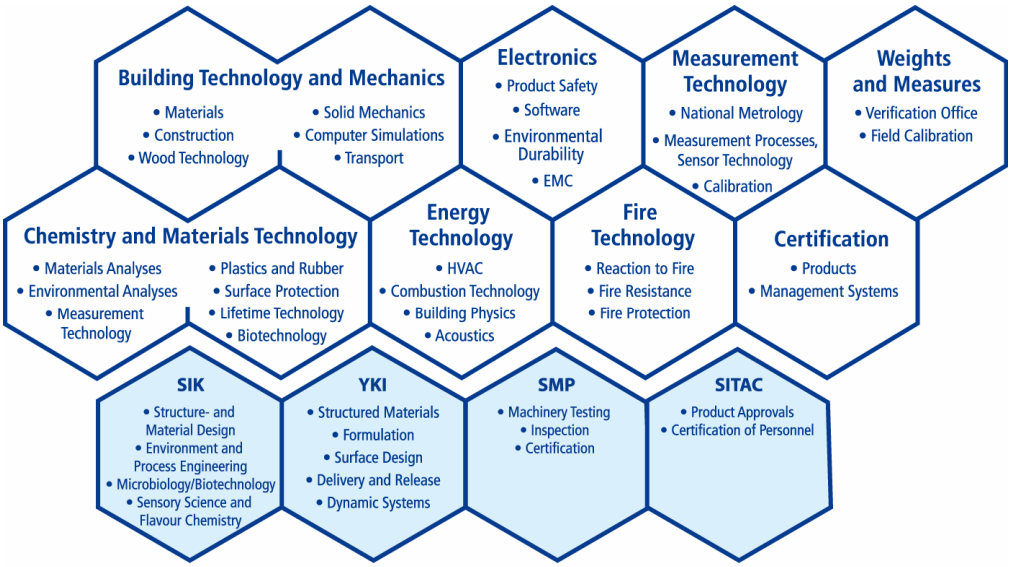
Finally, it should be pointed out that the intention is to use PolySpace during development, to successively improve the software quality and the development process. This is not only reflected by the high purchase price of the tool but also in the positive effects it has on the development life cycle. There are of course immediate results on bug detection when introducing PolySpace in a project but the long term effects are perhaps not revealed until subsequent project applying the tool is undertaken. From an economic point of view the investment is also further motivated if the tool is used in more then one project.

B.6 References

PolySpace http://www.polyspace.com/
PolySpace for C documentation

SP Technical Research Institute of Sweden develops and transfers technology for improving competitiveness and quality in industry, and for safety, conservation of resources and good environment in society as a whole. With Sweden's widest and most sophisticated range of equipment and expertise for technical investigation, measurement, testing and certification, we perform research and development in close liaison with universities, institutes of technology and international partners.

SP is a EU-notified body and accredited test laboratory. Our headquarters are in Borås, in the west part of Sweden.



SP is organised into eight technology units and four subsidiaries

SP Technical Research Institute of Sweden
Box 857, SE-501 15 BORÅS, SWEDEN
Telephone: +46 10 516 50 00, Telefax: +46 33 13 55 02
E-mail: info@sp.se, Internet: www.sp.se
www.sp.se

Electronics
SP Report 2007:14
ISBN 978-91-85533-82-4
ISSN 0284-5172
A Member of
United Competence