Safety Critical Software

Prof. Dr.-Ing. habil. Josef Börcsök,

HIMA Paul Hildebrandt GmbH + Co KG, Germany

Introduction

This paper discusses the methodical analysis of hardware architectures used in safety-related applications. It provides an excursus on a safe computer system's software technology and specifies the overview in greater details. This integrates the last sections presenting the required test procedures. The excursus cannot, however, be complete because studies and methods have increased rapidly, particularly with respect to object-oriented software system's design and programming design.

For several years, no combined software-hardware solutions have been utilized in the high risky environments in which safety systems are usually used. The first developments in software environment did not follow methodical and/or structured procedures. Before methodical program development procedures and structured software design techniques were used, the program code was created when the programmer(s) believed that he or they had understood the problem. The result of this approach was a set of extremely expensive test procedures leading in many cases to reject these reprogramming and improvement measures and to completely rewrite the software or parts of it.

In the early Seventies, the prevailing idea was to approach software development in a structured manner. The well-known waterfall model with its severe phase concept made its arrival on the scene. The waterfall model's negative aspect is the considerable documentation effort leading to a great documentation quantity even before a single software part can function. In the last years, several software development and testing procedures became established. Because of their advantages and disadvantages, they cannot be useful applied in every project. IEC 61508, the general norm valid for safety systems, even prescribes tool support for designing safety-critical software gives a structured overview of the safe software development. The next sections briefly present the best-known models.

6.1 Waterfall Model

The software development phases in the waterfall model are strict separated from one another. The following phases are typical:

Requirements analysis,
Design,
Specification,
Implementation,
Integration,
Operation.

Following a rigid up-down method, a phase may only begin when the previous one has been completed successfully. If problems resulting from an upstream occur, development must continue from this phase; a consequence is that in some cases parts of the performed development work must be rejected. The waterfall model is characterized by a clear and systematic procedure juxtaposed to little efficiency and flexibility. In praxis, in fact, projects may not be elaborated in a such linear form and a kickback in already completed phases may be necessary.
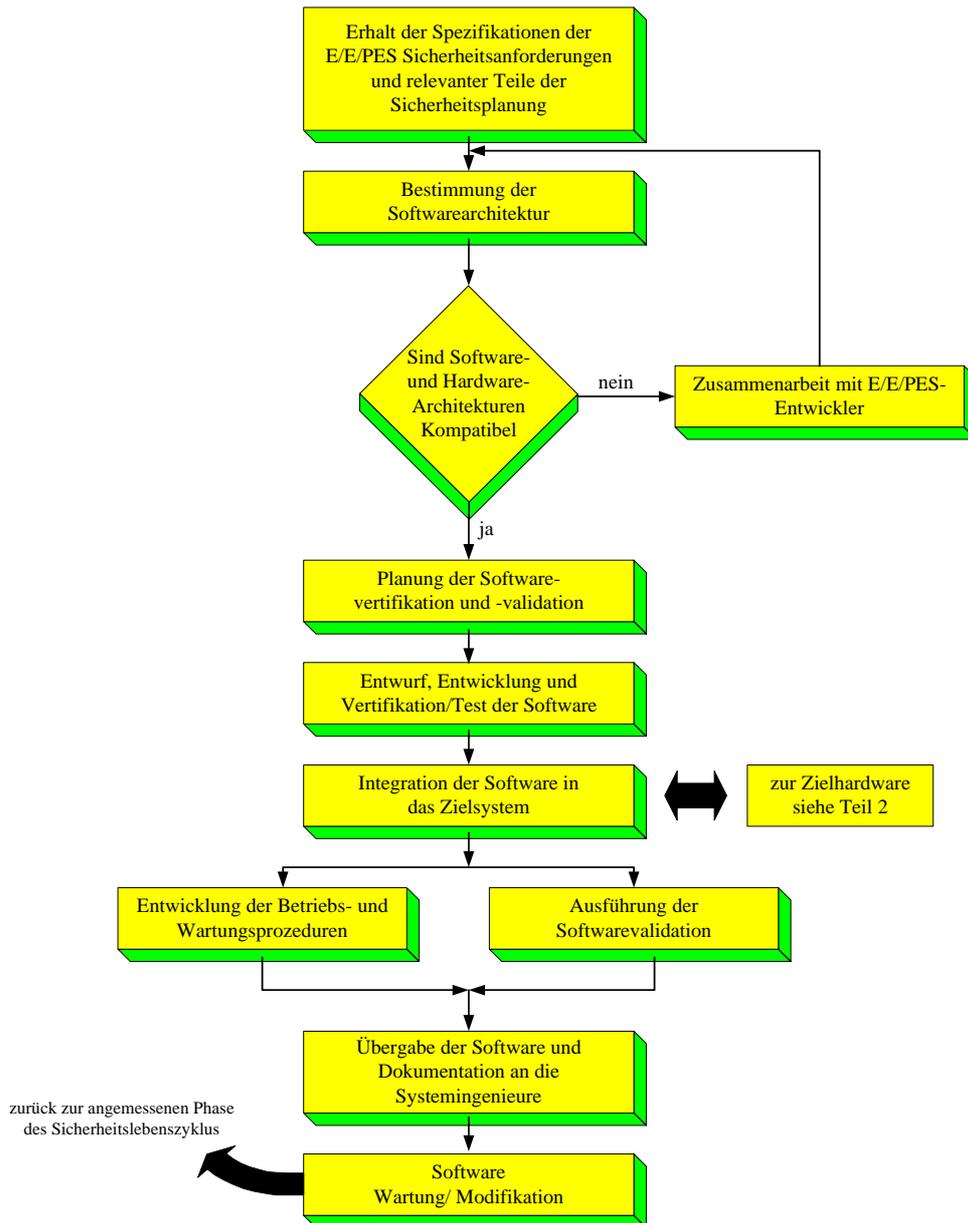
Figure 6.1: *Structural Software Creation for Safe Systems*

6.2 Spiral Model

In the spiral model, the software design is divided in phases running through over and over, until the desired result is achieved. The result is thus a cycle repeatedly run through instead of the waterfall model's linear procedure. The typical phases are:

Object definition,
Alternatives and risks analysis,
Selection of an alternative,
Development,
Testing and result assessment,
Next steps' planning, dependent on the test results,
Original objective's modification, if required.

Compared to the waterfall model, the spiral model is much more flexible.

6.3 Rapid Prototyping

Using this model, a prototype complying with a particular specification is created as soon as possible. The prototype is already equivalent to the end software product to a great extent and for some essential points, but it does not have the complete range of functions. The prototype is tested to find out if the design meets the specifications and requirements and if the chosen path takes the requirements into considerations. If necessary, a new or modified specification must be created. Finally, the developed software is further developed to the end software product in accordance with the actualized specifications. With complex systems is also typical to create different prototypes. The advantage in this case is that, with the person requiring the software accompanying the development process, the product corresponds exactly the customer's instructions and desires. Fault or undesired development approaches are early recognized and repaired. Because no fixed specification exists and the requirements are still modified in the implementation phase, the resulting software is „grown" and its design is not, therefore, optimal.

6.4 V Model

In the Federal Republic of Germany, the V model is increasingly popular when developing software for governmental or military customer. The V model is a generic description which does not prescribed any tools or methods. For a specific project, the actual procedure must thus first be determined. For this reason, the V model cannot be used as such like the other models, but it can be adapt perfectly to the objective defined. The V model uses specific models to support project management, system creation, quality management and configuration management respectively.

6.5 Software Development for Safety-Related Systems

In the industrial and scientific habit, the development of safety-related software proceeds in three general phases:

1. Defining objects for the reliability of the software to b developed
2. Taking appropriate measures for achieving the objectives
3. Using verification methods to quantify the success.

This division is generally not sufficient for formal structuring the process of safety-related software's development. Programming models adapt in greater detail to the specific problem are necessary. Nonetheless, the next sections describes fundamental measures and techniques for developing safety-critical software basing on this simplified division.

One of the most important phases, whether with software or hardware, is the phase of creating the specification. During this phase, not only the functional requirements for the program and/or hardware are described in detail, but also the required characteristics in terms of operating safety.

During the second phase, the actual software implementation, one should use not only techniques leading to a possibly fault free program code, but integrate specific functions. in the program. The latter do not help accomplishing the actual task, but detecting and repairing faulty states during operation (for other details, refer also chapter 7).

After completing the software, the third step begins with a testing phase to find out if specification has been correctly implemented. For software with high safety and reliability requirements in particular, it must be tested if the desired operating safety has been achieved. This may be done using different methods, such as correctness proof or empirical test procedures.

6.5.1 Determining the Safety Requirements

As mentioned, the reliability and safety requirements for a concrete system comprising hardware and software result from the intended application. The higher the hazard potential, the more severe the possible damages in case of incorrect functioning, the higher the requirements for developing the system.

Specific standards, such as DIN 31000/VDE 1000, DIN V 19250 or IEC 61508, describe therefore measures for quantifying the hazard potential and, starting from this point, a distinction in risk or requirement classes. As defined in chapter 2.1, for each class, it is specified, which safety requirement should be met.

Analysed the safety requirements for the system to be developed, it must be verified, which essential feature is resulting for the software. These features must be then described as a specification's part and taken into consideration when applying the test procedures.

6.6 Implementation Procedure

Generally to maintain a safe and reliable software during implementation, one should follow the common rules contributing to a fault poverty and code readability. The thorough application of correct programming techniques is also economically profitable. The next paragraphs present briefly structured programming, modularization, orientation to the object, coding rules and proving the software reliability.

6.6.1 Structured Programming

For applications with safety-technical background a great range of different programs are essential. When developing safety-relevant software, a structured programming style must be found. In this case, just three different sequence constructs are allowed:

Sequence (normal command sequence)
Branching (for example *if...then...else*)
Repetition (for example loops with *for* or *while*)

Limiting the constructs to these three fundamental has not only the advantages to reduce programming faults and improve the code readability, but the fact that it is easier to provide correctness proof using formal methods.

6.6.2 Modularization

A structured programming alone cannot help controlling the increasing program complexity. This and the fact that several programmers must work on the safety-related program simultaneously are the reasons for the software modularization.

Each software part represents then a more or less cohesive bloc comprised of data and algorithms belonging together thematically or functionally.

Due to the obvious smaller module size compared to an individual program, such a module can be far better managed and tested. Faults can therefore be avoided and/or detected more easily. In this way, for each module a rather high reliability level may be achieved. A big and complex program comprising several modules will consequentially have a reduced number of implementation faults. To achieve optimal results in accordance with this concept, some specific procedures must be followed.

A module's interfaces must be clear defined and documented. A developer wishing to use an outside module and access it from his module, must be able to consider the other module as a „black box". This means that he must be able to use the outside module just supported by knowing the interfaces and not having any idea on its internal functions (capsuling). The additional resulting advantage is that the module's internal algorithms may be modified and optimized at any time, without other program parts having to be adapted as long as the conformity with the interface specification is given.

If the different modules' coupling is kept as small as possible („loose coupling"), the faults occurring in a module do not affect the other module's functioning. The module interface must be design such that an access from outside is only allowed for essential data and functions.

6.6.3 Orientation to the Object

The approach of the object-oriented programming style is to create the program as problem reproduction such that the structure and commands of the program conceptual formulation is reproduced.

Further, the problem is divided in entities representing each a possibly cohesive sub-aspect of the problem. Every entity is then modeled in the program code and represented by an object. The same objects belong to the same class.

A class consists in actual things or ideas abstracted from the problem context and all having the same characteristics or features. Starting from a quite general base class, further concrete classes can be derived by inheritance. Features defined for the base class are therefore automatically the derived class's features. It thus results that common features are resumed in common base classes and that the classes implicitly derived from them also have such features plus some additional peculiarities. A base class describes a characteristic or capacity more abstract, whereas the description provided for the derived classes is always concrete.

Base classes often only helps describing general concepts; parameters/objects actually contained in the program exist only because of the derived classes. A class represents therefore a cohesive description of a sub-problem. Requirements specific for a module such as object minimization in an interface or practicability of the „black box" principal, must be met also during the class implementation. Thanks to the object-oriented programming, fulfilling modern requirements such as data capsuling, avoidance of global data structures, repetitively, readability, maintainability etc. is possible in a particular elegant and easy manner. All these features represents some main requirements for safety-related software development.

6.6.4 Coding Rules

When developing safety-oriented software, coding rules are prescribed by the different test houses, also if both approaches of structured programming and orientation to the object were applied consistently. Generally, during the implementation phase, it is useful to apply an up-down method. Starting from the system specification or requirements specification, a preliminary design should be accomplished in which the general program architecture has been defined and the problem space has been divided in classes. Only at this point, the detailed design containing the single functions and data structures may be build up. Modern tools support this procedure.

When implementing algorithms, clear structures should be preferred, as particular artful constructs do not really lead to increased efficiency, but the code can be understood only with difficulty.

The actual program code's implementation should meet certain conventions specified in the coding rules. The result is a good readable and easily verifiable source code, two qualities which are essential and desirable with several proof procedures for controlling the program correctness.

Without claiming to be complete, the next paragraphs list further guidelines which a programmer should consider when creating safe and reliable software to fulfill the testing requirements demanded by a test house:

The testing and verification methods should be already considered during the implementation phase.

The documentation must be detailed and accompany the project.

Data correctness is verified also during operation.

If there are tested function or class library for a specific task, these should be preferred to an own new implementation.

A part or the whole machine code can be stored in a ROM rather than in a RAM to ensure the program inalterability.

No undocumented or undefined language behaviors should be exploited, as the program functioning with another compiler, libraries from other manufacturers or another operating system's version is not ensured.

When using real-time applications and multitasking, one should pay attention that the resources are blocked possibly for a short period. The different, synchronous executed program threads must be correctly synchronized when using common data or resources.

6.7 Proving the Reliability

Informal proof procedures may be already used during the development phase. Thank to this measure, faults can be detected in early stages, leading thus to improved quality and economic advantages. Among the informal procedures rank inspection, review and Walkthrough.

Using appropriate tools, structural analysis may be performed. An extensive automated semantic study of the control and data streams may be performed on the basis of the program code.

The program correctness proof is carried out in accordance with severe formal and mathematical methods. To provide such a proof, it is reasonable and often partly necessary to make the program "proofable".

A fundamental and old familiar method for testing the software quality is the execution of tests aiming to verify if the program with a specific input behave in accordance with specification. When testing complex programs, however, the procedure cannot be complete, i.e. it cannot be executed for the whole input range.

6.7.1 Inspection

During an inspection a group of developers or system designers verifies the current software project state using check lists. The attention is turned rather to the formal and content document correctness and less to the program correctness. Documents compiled in different project phases are compared with one another to verify the desired further development.

During the actual inspection meeting, the detected defects are gathered. The actual program developers have no firm task during the inspection; they should neither explain not describe in details their work as the documentation comprehensibility is also checked.

In the inspection's wrap-up phase, a to-do list for repairing the detected defects is created. The meeting moderator should take care of the defined changes' execution, but does not have to control its success as this will be part of the next inspection.

6.7.2 Review

During a review, not only quality and state of the program to be developed are inspected, but suggestions for improvement are made and discussed. In this phase modifications or integrations to the project guidelines are possible.

A review takes place at the end of the development phase and is carried out by the project team members. It is verified if the intermediary results defined in the project plan have been achieved and if the result corresponds to the requirements specification in form and content. The review results including the suggestions made and the new decisions taken are put down in writing.

6.7.3 Walkthrough

During a walkthrough, the program is tested step by step in terms of functional or source code level. The software developer explains his program and, following his instructions, other persons check faults or inconsistencies. In contrast to inspection and review, the software developer is always active during a walkthrough. This kind of proof procedure includes a more detailed analysis of the object to be tested, if compared with both previously mentioned procedures. Walkthroughs rank among the best modern instrument to proof the design conversion correctness. Hidden implementation faults can be thus early detected, but a great effort is involved.

6.7.4 Structural Analysis

During the structural analysis, the formal software's features are investigated using automated tools. Using this procedure, one can receive statements about the control stream, the Data stream and the program semantic. The analysis tool creates an output in form of pseudo codes, formulas, algorithms or graphical structure image which can be tested using the specification. If the correctness of such an image can be immediately verified, this procedure cogency can be roughly compared with that of the program correctness proof. Further, such analysis can provide information about the code cover and help deducing reasonable test procedures for white-box or black-box tests.

Table 6.1: Comparisonof Informal Proof Procedures

Criterion	Inspection	Review	Walkthrough	Desk test
Group composition	Moderator, author, tester, user	Project team, client, contractor	Author, colleagues, testing person	Author, testing person or QA Department
Member number	3-6	5-15	2-6	1
Duration	2 hours max. plus preparation	1-2 days	2 hours max.	4 hours max.
Objects	Documents, including agreements, codes	Phase products, project plans, project reports, problem field	Documents, SW preliminary design, SW detailed design (e.g. codes)	Documents, SW preliminary design, SW detailed design (e.g. codes)
Objective	Documents' testing	Analysis of project states, problem solution, measures	Detecting faults and incompleteness	Fehler und Unvollständigkeiten feststellen
Moment	After document creation	Phase ending, milestone	After document creation	After document creation
Execution	Check lists	Agenda	Idea exposition in front of a public	Idea exposition, formalities check
Result	Protocol with tested documentation, new inspection if necessary	Protocol with decisions	Protocol with faults, incompleteness	Protocol with faults, incompleteness, questions, inconsistencies

6.7.5 Program Correctness Proof

The program correctness can be proved using scientific mathematical procedures. Should the proof fail, a faultiness is not automatically demonstrated. One can, however, deduce from the failure an example situation which could lead to program fault. To use this procedure, an intensive training is essential. Further, it is useful if not necessary to design from the beginning the form of the program to be examined such that this proof procedure can be easily applied. By using this concept appropriately, the program efficiency and development can be improved. On the other hand, the efforts increase disproportionate to the dimensions of the program to be examined. Consequentially, this procedure cannot be always utilized.

6.7.6 Testing

A test can be defined as a process for verifying and validating a system or its components. When developing safety-related systems, the testing of safety-critical software - and here are not meant general functional tests – is one of the most important requirement together with the testing of hardware.

The test procedure's results can be used to evaluate integrity or safety. Test processes detect faults that are repaired for increasing reliability. The testing of safety-related hardware and software is complex. Generally, tests are performed in different phases during the system development. They are referred to as:

Module tests
System integration tests
System validation tests

Module tests include the assessment of small, clear hardware and/or software functions. Faults detected at this level are often easy to locate and repair due to the program or component simplicity.

System integration tests investigate the characteristics of a modules' collection n and aim to an actual module interaction. Faults detected in this phase are probably more difficult to repair than the ones detected during a module test because the test arrangement is more complex by definition.

Table 6.2: *Test Procedures in Different Life Cycles*

Life Cycle Phase	Dynamical Testing	Structural Testing	Modeling
Analyzing and specifying the requirements		✔	✔
Top level design		✔	✔
Detailed design		✔	✔
Implementation	✔	✔	
Integration tests	✔	✔	✔
System validation	✔		✔

System validation tests should demonstrate that the whole system meets the system requirements. If defects occur, they are often the result of lacks in the specification. They are extremely difficult to remove, independently whether the faults result from the specification or the requirements, because these documents may only be modified reviewing the whole development process. The test can be performed in a dynamical or structural manner, or they may be based on a mathematical model.

Modeling means to use a mathematical representation of the system behavior. The model is used to preserve insights in the probable system characteristic and can be performed manually or using a computer system. Modeling is normally implemented in the first stages to investigate the fundamental nature of the suggested system or of the environment in which the system is used.

At the end of this chapter, so-called black-box and white-box test procedure are briefly presented. Test methods are often classified using the information available to the persons performing the work.

In black- box tests, the test engineer does not know anything about the system implementation and must use the information about specification as basis for the tests. The black-box test is probably the purest form of assessment, as it is just tested whether the system provides the reactions defined in the specification. The individual module are neither identified nor analyzed as the internal structure cannot be seen. Black-box tests provide the highest level of independence between developer and the person executing the tests and are thus for independent validating extremely profitable.

In white-box tests, information about the system implementation is known. Most test methods are based on the white-box approach. This technique can be applied in all development stages and to hardware as well as to software. Knowledge of a unit's internal structure simplifies extremely the dynamical testing as tests can be developed specifically for each module to be examined. Company-specific knowledge is useful for selecting the test conditions and trying to minimize the test points.

2 Literature

[1]   IEC/EN 61508: International Standard 61508 Functional safety: Safety-related System. Geneva, International Electrotechnical Commission

[2]   Börcsök, J.: International and EU Standard 61508, Presentation within the VD Conference of HIMA GmbH + CO KG, 2002

[3]   Börcsök, J.: Elektronische Sicherheitssysteme, Hüthig publishing company.

[4]   Börcsök, J.: Sicherheits-Rechnerarchitektur Teil 1 und 2, lecture of University of Kassel, 2000/2001.

[5]   Börcsök, J.: Echtzeitbetriebsysteme für sicherheitsgerichtete Realzeitrechner, lecture of University of Kassel, 2000/2001.

[6]   DIN VDE 0801: Funktionale Sicherheit, sicherheitsbezogenener elektrischer/elektronischer/programmierbarer elektronischer Systeme (E/E/PES), (IEC 65A/255/CDV: 1998), Page: 27f, August 1998

[7]   DIN V 19250: Grundlegende Sicherheitsbetrachtungen für MSR-Schutzeinrichtungen. Beuth publishing company, Berlin 1998

[8]   DIN VDE 0801/A1: Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben, Beuth publishing company

[9]   IEC 60880-2: Software für Rechner mit sicherheitskritischer Bedeutung, 12/2001

Prof. Dr.-Ing. habil. Josef Börcsök is executive vice president for research and development of the company HIMA + CO KG industry automation. He has been operating in the field of safety-related computer technology for several years and is member of several DKE committees. He lectures at universities, as well as at universities of applied sciences with lectures of automation systems, computer technology, realtime systems and safety-related computer technology.

Address:
HIMA GmbH + Co KG, Albert Bassermann-Str. 28, D-68782 Bruehl in Mannheim,
Tel.: 06202-709-270, email: j.boercsoek@hima.com