Rijkswaterstaat
Ministerie van Infrastructuur en Milieu

TOPAAS: Part 1 Guide (Framework)

Date	28 March 2018
Status	Definitive

Publication details

Name Standard	TOPAAS Framework
Description:	The TOPAAS framework describes a structural approach to failure probability analysis of software-intensive systems.
Status:	Definitive
Date	28 March 2018
Version number:	1.0
Type:	Framework
Responsible PE:	Jean-Luc Beguin
Use in process:	Construction and Maintenance
Network:	HVWN, HWS and HWN
Object:	All RWS infrastructure
Main discipline:	Asset management
Discipline:	Risk-based Management and Maintenance (RGBO)
Information:	Probo@rws.nl
Department responsible:	RWS GPO - Technical Management Advisory Department (ATM)
WW RWS Number:	1319

Overview of changes

Version	Date	Changes
0.7	10 Jan 2013	Version number 0.7, definitive
1.0	28 Mar 2018	Textual update, model is unchanged, explanation is more extensive, see overview comments.

Contents

1 Introduction

1.1 Context

In 2010, Rijkswaterstaat decided to introduce risk-based management and maintenance (RGBO) within asset management (AM). With RGBO, all the risks for the functioning of an object are mapped out, allowing them to be controlled in a transparent and well-considered manner. This is in contrast to traditional maintenance, which is often condition-driven, and focuses on maintaining a certain technical level.

The aim of RGBO is to control the risks in the functioning of the three networks by controlling management and maintenance actions in such a way that the agreed performance is delivered at minimum (lifetime) costs. RGBO makes the relationship between network performance and maintenance explicit. In 2013, the RWS Board decided to develop RGBO further in order to gain full control by means of a follow-up RGBO process, with a subsequent re-evaluation in 2016.

In 2016, Rijkswaterstaat drew up the Performance-driven Risk Analyses (PRA) guide for this purpose to make risk-based thinking applicable to all infrastructural assets managed by Rijkswaterstaat. This guide integrates and replaces the RAMS Guide and the Risk-based Management and Maintenance Guide.

Performance-based risk analysis (PRA) is an important tool. The PRA shows the balance between the performance of an object, the risks that affect performance and the costs of maintaining performance. With the help of PRAs, Rijkswaterstaat can make well-founded decisions on construction, management and maintenance.

In addition to this guide, various methods have been further elaborated in terms of content and documented in separate guides. That includes this standard, which describes a structural approach to failure probability analysis of software-intensive systems.

Software is increasingly part of the structures that Rijkswaterstaat builds and manages. This software plays a crucial role in operating, controlling and securing many civil and technical objects: without properly functioning software, the object may be unexpectedly unavailable or even unsafe for people and the environment. In practice, this software could also be the cause of the failure of an object and must therefore be included in the quantitative underpinning of the RAMS analysis, in order to provide a good insight into future performance.

However, including software in the RAMS analysis is problematic. In many cases, science currently has no practical answer to the question 'what is the probability of failure of the supplied software?'. So, in contrast to civil and electrical engineering where there are scientifically proven methods to provide an object's probability of failure in a structured way, this is often not the case with software. TOPAAS is intended for that specific situation.

An alternative to calculating the probability of failure is to have it estimated by a group of experts. This is an accepted approach among risk analysts, but there are practical objections involved. For example, it is rather cumbersome to convene a

group for each project and there is a significant chance that the experts will not agree on the project.
TOPAAS approaches the estimates of a group of experts by means of a structured questionnaire. This allows the software failure probability to be estimated within an order of magnitude.

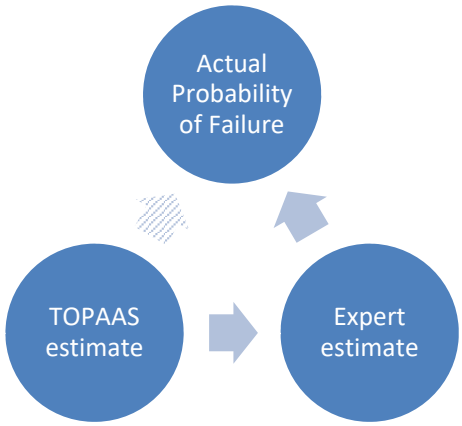


Figure 1: Relationship of TOPAAS to the actual probability of failure

1.2 Objective

This guide is intended for RAMS specialists who are involved in the assessment of software. The process described here is seen by Rijkswaterstaat as the accepted method for carrying out software reliability analyses. In addition to the description of the necessary steps in an analysis, the preconditions and the necessary documentation are also described.

This guide is the second iteration of both the model and the guide, and incorporates some 10 years of practical experience of using TOPAAS in the field. This guide replaces all previous descriptions of the TOPAAS model.

This guide describes how the second iteration of TOPAAS must be applied and omits the why. An explanation of the substantive structure of the model is described in the maintenance documentation (see Chapter 6 References) and has been kept out of this guide for the sake of readability. In addition to describing some model-based changes, this guide mainly focuses on providing a better explanation of applying the model, to make its application simpler and more unambiguous. The necessary reporting is also explicitly described, so that this too provides clarity to all parties.

On some points, we also provide practical guidelines in both technical and process approaches. These are deliberately not compulsory activities, but are intended to simplify the application of the model in a responsible manner.

1.3 What is TOPAAS?

The abbreviation TOPAAS stands for *Task Oriented Probability of Abnormalities Analysis for Software*. This incorporates the following essential features of TOPAAS:

- TOPAAS looks specifically at task execution by software. In a RAMS analysis, software is not an amorphous object where every error has equal consequences for reliability, availability or safety. In a system, software

performs a defined task (task execution) that may be essential for one or more RAMS requirements.
- TOPAAS provides a probability of failure (per question), a quantitative measure for reliability of the task execution, which can be added as basic events (Q) in a fault tree.
- TOPAAS considers all forms of abnormal behaviour that could compromise a RAMS objective, provided they come from the software. So not only behaviour that is absent, too early or too late, but also undesirable behaviour. The behaviour of the environment (systems exceeding specifications) or hardware failures are explicitly excluded here. This needs to be addressed in the RAMS analysis itself. So-called cascading failure will be discussed in more detail later.
- TOPAAS assumes a context in which the relevant failure modes are identified in the fault tree. The use of fault trees also implies that the software has been designed, built and tested in a thorough way.
- TOPAAS can be applied in all phases of system development: design, realisation, testing and production.
- This probability of failure is an a priori probability of failure that can be used after practical experiences to further Bayesian update with actual performance.
- TOPAAS provides a TOPAAS score by means of a parameter model. The TOPAAS score results in a probability of failure per question (Q).
- TOPAAS can be applied to every type of software system (including COTS).

In short, Topaas is a method for estimating the probability that a software result, decision, or control is absent, incorrect, too early or too late.

1.4 What is TOPAAS certainly not?

TOPAAS is a measuring instrument: it helps to estimate the probability of failure of the task execution in a structured way. It is certainly not a process engineering tool to determine the lower limit of the development effort. For this purpose, *best-practice* documents such as the IEC 61508 [1] are much better suited: just like a cookery book, this standard describes the required activities based on a desired RAMS level (probability of failure) via the corresponding SIL. This makes TOPAAS and IEC61508 complementary: the IEC61508 describes the *common accepted practice* to realise a probability of failure, while TOPAAS looks at the actual realised probability of failure.

TOPAAS is not a design tool, either. Although TOPAAS can help in comparing multiple design solutions in RAMS terms, it is certainly not a method that guarantees that a designer makes the right choices.

Nor is TOPAAS intended to specifically include underlying aspects in the requirements specification for tenders or contracts. A RAMS level to be achieved can be part of the question; a contractor can answer this by means of demonstrable expertise. Inclusion of individual aspects of TOPAAS that determine that level is therefore undesirable.

TOPAAS deals with the reliability according to the definition in [8]. It should be noted that this is not exactly the same as the common definition of *'reliability'* used in the IT sector, according to ISO/IEC 25010. TOPAAS does not contribute to other

quality aspects of ISO/IEC 25010, nor estimates of time to failure or time to repair, nor other maintenance characteristics.

1.5 Scope of application

The probability of software failure can be determined in two fundamental ways. The first way is based on the findings of operational software. It involves looking at how the software behaves when executed in different situations. These can be created test situations as well as situations in production. Examples are Reliability Growth Modelling (RGM) and Monte Carlo tests. The second way is based on the circumstances and external characteristics of the software such as the development process, the complexity and the scope of the code. TOPAAS is based on the second way, like its predecessor the TDT model.

Each failure probability analysis method has specific characteristics that make it easily or less applicable in certain situations. Specific examples of failure probability analysis methods are:

- Failure probability analysis during the design and realisation phase (i.e. before running software is available) is only possible with methods such as TOPAAS;
- Analysis on safety-critical functions already running where sufficient defect data are available, can preferably be done with Reliability Growth Modelling;
- Analyses on such functions without statistically significant numbers of defects can best be done with TOPAAS.

TOPAAS is designed for operational industrial systems that have a noticeable impact on RAMS performance of structures and industrial process automation, where reliability cannot be established or easily determined by other means. These can be newly developed systems as well as COTS products.

Often these are simple short chains of sensors, systems and actuators without function providers experiencing significant mutual interference. As soon as task executions have many interdependencies, because they are strongly intertwined, the simple analysis described in this guide is not sufficient and more extensive analyses need to be carried out. More extensive analyses are also necessary to verify extremely high reliability requirements (probability of failure 10^{-6} or lower).

TOPAAS is not designed to identify all IT risks. TOPAAS is designed to estimate the probability of intrinsic failure in the task execution of a well-designed and well-tested system. The probability of failure due to external IT threats, for instance cybercrime and viruses, cannot be estimated with TOPAAS.

In the decision table below, the different methods are linked to the following specific characteristics:

- development phase of the software
- frequency of use of the software
- number of findings from testing or production
- complexity of the software

The methods themselves are dealt with in Appendix A.

ontwikkelfase	ontwerp		realisatie		testen								productie							
frequentie van gebruik hoog	Y	N	Y	N	Y				N				Y				N			
aantal bevindingen significant	-	-	-	-	Y		N		Y		N		Y		N		Y		N	
complexiteit hoog	-	-	-	-	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
TOPAAS	X[1]	X	X[1]	X	X[1]	X[1]	X[1]	X[1]	X	X	X	X	X[1]	X[1]	X[1]	X[1]	X	X	X	X
RGM					X	X			X	X			X	X			X	X		
Monte Carlo						X		X		X		X		X		X		X		X
Duurtest						X		X		X		X		X		X		X		X

Figure 2: Decision table, X1: TOPAAS applied to applications with a high frequency of use produces a factor that via memo 'Continuous systems and the probability of software failure' results in a failure probability per period (λ).

The use of the table is explained in the following example:
Above the thick line on the left (first quadrant) are the different characteristics; for example, the *'number of significant findings'*. Above the thick line on the right (second quadrant) are the corresponding conditions; for example, the *'number of significant findings'* is not applicable ('-'), where there is ('Y') or not true ('N'). The indicated condition includes the action, or possible way of determining reliability, under the thick line in the same column.
If the project is in the development phase of *'testing',* the frequency of use of the system concerned is *'high'*, the number of significant findings is *'low'* and the complexity is *'high'*, then TOPAAS is the suitable method, noting that via the memo *'Continuous systems and the probability of software failure'* a probability of failure per period (*λ*) can be determined. However, this method is still under development and is not scientifically substantiated.

The use of TOPAAS in the design or realisation phase is necessarily limited to assessing the production process, the architecture and possibly plans with regard to traceability and test coverage. The availability or unavailability of field data can also be taken into account. This provides an initial indication of the probability of failure. The uncertainty concerning the various factors additionally provides an upper and lower limit within which the ultimate probability of failure is likely to move. The graph below shows an example of this phenomenon per TOPAAS dimension and for the total probability of failure. Such an approach therefore does not provide reliable quantification, but it can identify shortcomings or points for attention.

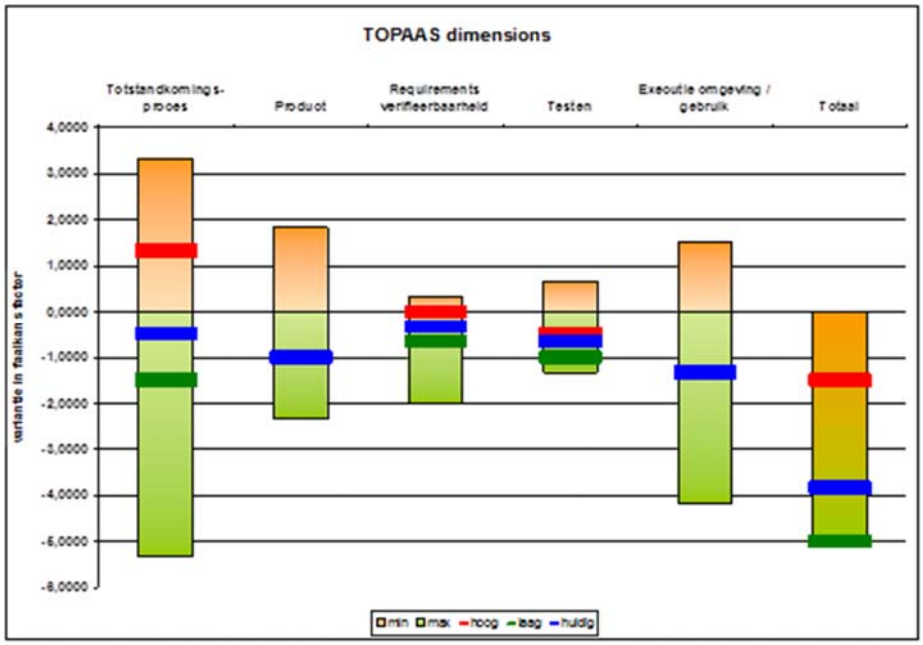


Figure 3: TOPAAS dimensions

The columns represent the theoretical maximum positive and negative contributions to the probability of failure. Within this, the blue bar shows the current TOPAAS estimate. The red and green bars around it are an estimate of the possible final values for the specific project. Of course, this is only one way of graphically displaying information.

Incidentally, a failure probability analysis using RGM can be a good addition to and verification of a failure probability analysis carried out earlier in the development process using TOPAAS. When the results of an RGM analysis based on running software deviate significantly from the TOPAAS analysis, further investigation is necessary.

It should be mentioned that IEC 61508 is explicitly not part of the alternative methods; IEC 61508 is not a method for failure probability analysis! The application of IEC 61508 enables a project or development team to build software that meets a certain probability of failure norm by following the *'recommended'* and *'highly recommended practices'* at the target Safety Integrity Level. However, it is by no means a guarantee that the software will achieve this probability of failure. Provisions, for instance using TOPAAS, must demonstrate the ultimately achieved probability of failure. However, the application of IEC 61508 is part of the assessment within TOPAAS.

1.6 Output and accuracy of results

The TOPAAS score is rounded to a whole number in the last step and is therefore between 0 and -5. A score higher than -2 does not provide useful quantification. The interpretation of the scores -2, -3, -4, -5 results in a probability of failure of (in fault tree analyses also called the Q) 10^{-2}, 10^{-3}, 10^{-4} or 10^{-5} per question. For dealing with frequently used systems, refer to the memo *'Continuous systems and the probability of software failure'*.

We see that the accuracy of the resulting Q is in orders of magnitude, so the outcome is always 10^{-2}, 10^{-3}, 10^{-4} or 10^{-5} per question. The main reason for this is that the expert estimates are not more accurate and 'pseudo-accuracies' should be avoided. In practice, there is not much difference at the top event between a task execution with a probability of failure of 10^{-3} or a probability of failure of $6{\cdot}10^{-4}$, and this inaccuracy often does not cause problems. TOPAAS is accurate enough to determine whether or not software is a hindrance to RAMS performance.

Scientific research shows that software failure probabilities estimates by experts take a conservative approach with a maximum deviation of a single order of magnitude. The TOPAAS results will generally provide a reasonable to conservative estimate of the software's probability of failure. It is always wise to investigate the influence of this conservatism on the analysis with a sensitivity analysis.

A probability of failure of one reflects the total lack of confidence in the task execution by experts and should be treated as a finding in the form of restructuring the solution.

1.7 Principles

This guide is concerned with quantifying the contribution of software failure to a solution that is based on a solid architecture and that has been professionally developed further into an effective solution. TOPAAS assumes at least a basic design and testing process, which also implies a minimum level of quality assurance.

TOPAAS cannot deal with systems where fundamental design flaws are present. The principle is that in the validation and verification approach, these faults are taken into account as risks. Faults at the architectural level (change of internal or external matters) can occur during the lifespan. The principle here is that the change process addresses these risks.

Before the TOPAAS analysis takes place, the analyst responsible must therefore determine whether the system is free of fundamental faults and also that the production process followed is suitable for software development.

> *Requirement:* A description or reference of the use of an appropriate production process must be part of the TOPAAS reporting.

1.8 Reading guide

In chapter 2, we give a broad outline of the process and distinguish the two process steps. In chapter 3, the qualitative analysis and in chapter 4, the quantitative analysis.

TOPAAS was drawn up in collaboration with:
Dr Wouter Geurts (CGI)
Jaap van Ekris (Delta-Pi)
Ed Brandt (Refis)
Dr Gerben Heslinga (Intermedion)
Dr Gea Kolk (Movares)
Prof. Jan-Friso Groote (TU/e)
Prof. Mariëlle Stoelinga (UT/Radboud University)

1.9 Document overview

TOPAAS consists of the following documents:

Part	Name	Type
1	Guide (this document)	Framework (binding)
2	Questionnaire	Informative
3	Model rationale	Informative
4	Reference and validation database	Informative
5	Maintenance process (draft)	Informative
6	Audit framework (draft)	Informative

To make a TOPAAS analysis, part 1 Guide and possibly part 2 Questionnaire are required. Anyone wishing to know more about the background to TOPAAS is referred to part 3 Model rationale. Part 4 Reference and validation database is the underlying database containing the reference projects, which were required for the establishment of the TOPAAS model. This part is intended for Rijkswaterstaat only. Part 5 Maintenance process describes how the documents in the TOPAAS framework must be managed and maintained by Rijkswaterstaat. This part is intended for Rijkswaterstaat only. An audit framework is described in section 6. It describes how an auditor should perform an audit on a TOPAAS analysis. This part is necessary for the person wishing to perform an audit.

1.10 General tips

Software tends to form a first order object in the fault tree. As a result, a TOPAAS estimate relatively quickly has serious consequences for the estimation of RAMS performance of structures. A thorough approach during the analysis, with the help of risk analysts who have extensive experience with software reliability, is therefore essential to achieve a software structure that can be readily assessed. In addition, a sensitivity analysis is useful as input for, for instance, a measurement programme, which can be used to supply Bayesian updates.

Information: Good documentation of both the decisions and the underlying 'evidence' is necessary for both modelling and quantification. Communicating the decisions and outcomes to stakeholders is also important to gain acceptance of the software.

In this guide, we describe tools for both decision-making (modelling and quantification) and the minimum amount of documentation.

However, it is wise, also for future software maintenance, to include all the design solutions studied in the design file. In practice, such design rationales prove to be extremely valuable for the future maintenance of the software.

2 Working method

2.1 Broad outlines of the process

The principle of TOPAAS is that the contribution of software to the failure of an object is taken into account through the task execution of the software, where failure of task execution results in an undesirable or insecure situation. Software generally does more than the task execution included in the fault tree. Software reliability analysis focuses on the parts of the software that realise specific tasks in an object.

The basic concept for a TOPAAS analysis is the TUB (Task Execution Block/TaakUitvoeringsBlok), which should be seen as the software analogy of a hardware component. A hardware component (butterfly valve, pump, relay) has a very clear function and therefore a few failure modes. So it is already fairly clear what the basic event *'butterfly valve fails'* means. By analogy, basic events for software should be defined in the same way. These must be designated as not at all, not on time, or not correctly fulfilling a task (or tasks) assigned to the software. A basic event *'software crashes'* lacks the clear explanation of which task assigned to a piece of software is not being performed and is therefore compromising the mission. A better basic event is *'software is not controlling butterfly valve'*. The term TUB is defined in section 3.3.2.

In order to achieve a good software failure estimate, a TOPAAS analysis has two steps:

- A qualitative modelling of software failure, aimed at recognising task execution and dividing this up into concrete TUBs (see chapter 3);
- A quantitative determination of probability of failure, the purpose of which is to estimate a probability of failure for each TUB (see chapter 4).

The qualitative modelling is necessary in order to relate the results to an overarching fault tree and is crucial in order to arrive at a good technical demarcation of TUBs. Often this step also leads to a further elaboration of software failure in the fault tree. By paying close attention to this step, it is also easier to achieve a clearly delimited definition of the task execution of a TUB. This sharply delimited definition is necessary in the quantitative analysis in order to answer the questions.

The quantification of the probability of failure of a TUB is carried out in two steps. The first step is to determine the TOPAAS score on the basis of a questionnaire. The aim of the questionnaire is to work towards a substantiated estimate in a structural way. The second step is to interpret the TOPAAS score based on the TUB's trigger frequency.

Information: Not only the final answers are important, but also the underlying reasoning and evidence. This makes the scoring process transparent for the client.

2.2 General staffing requirements

A software reliability analysis requires a RAMS specialist with significant experience in software architecture. The RAMS specialist must be able to discuss in terms of content the division of the software into TUBs, with a realistically achievable quantification of these TUBs in mind. This requires a good understanding of architectural concepts and, on the other hand, a good understanding of how TOPAAS works.
It is highly desirable that the TOPAAS specialist is experienced in software-intensive systems and that the specialist is independent from the development team.

2.3 General requirements for the working method

Modelling and estimating software failures are partly subjective activities. The choices that are made influence the ultimately modelled probability of failure. The general approach proposed by the TOPAAS working group in this process is based on consensus and transparency. Reaching consensus on essential choices in modelling and aspects influencing quantification creates intersubjectivity: a result that is supported by multiple stakeholders and minimises individual flawed thinking. There should also be no more discussions about factual conclusions. Transparency is needed in order to be able to verify afterwards which choices were made and how these impacted on the further analysis process.

2.4 General preconditions

Documentation of the choices made concerning the efforts of the analysts and the working method used, as well as the reasons why the choices were made, is essential in order to make the analysis process transparent. Documentation limitations in the approach (e.g. limitations in the depth of analysis) are also part of this documentation.

2.5 General documentation requirements

This guide deliberately leaves open how the documentation is organised and how its quality is checked and guaranteed. It is conceivable that an accountability document (RAMS file) will be systematically built up, of which the TOPAAS analysis is a structural part. Other working models that create independent documentation by means of audits can also be a workable model. This is a choice that must be discussed with the client and the builder of the software.

The key issue is that all guiding decisions necessary for independent review must be documented.

> *Requirement:* The way in which the analysis process is organised and the qualifications of the analysts involved must be documented.

3 Qualitative analysis

3.1 Objective and overview

The aim of the qualitative analysis is to break down the top event so that the contributions from hardware, software and human actions are included separately. By introducing the concept of task execution, this connection is made: hardware *'controlling an engine'*, software *'taking a decision'*, human intervention *'carrying out an emergency repair according to work instruction'*.

This means that a basic event (failing task execution) is linked to collections of source code for which a probability of failure can be calculated at a later stage. These source-code collections are called TUBs (more specifically defined in section 3.3.2 *'TUB'*). It should be noted that the demarcation in TUBs may not correspond with the technical subdivision of source code into packages, modules or functions common in software engineering.

The qualitative analysis therefore leads to the identification of the TUB: a minimal software cross-section that does the task execution. The TUB must be sharply defined in order to reduce the risk of overestimating the probability of failure. More lines of code contain statistically more faults (defects), but not all of them lead to failure of the task execution. On the other hand, for practical reasons (measurability) it is often useful not to demarcate too sharply. This is an explicit subject in section 3.7 *'Choosing the TUBs'* and section 3.8 *'Practicalities during design'.*

The qualitative analysis has two inputs:

- the provisional fault tree;
- the structure diagrams of the software: its architecture.

These inputs are subject to substantive requirements, which are dealt with in more detail in section 3.4 *'Input documentation requirements'* .
The step from task execution to TUBs is often not an easy one to make. Because the relationship between task execution and software modules is recorded in the architecture, we start with a rough sketch: we identify the packages/modules/functions involved in a task execution. The boundaries are subsequently honed: the exact boundaries of the TUBs are determined by grouping the modules in terms of internal characteristics (process persistence, development team, etc.), and impact on a specific task execution, and then clustering.

Broadly speaking, identification of the TUBs consists of the following sub-steps:

- Identifying software events in the fault tree. For a specific RAMS performance, the relevant task executions of software (TUBs) are identified. This is elaborated in section 3.5 *'Recognition of failing task execution in the fault tree'.*
- Identifying the TUBs in the architecture. The modules involved in the execution of a task are identified from an overview diagram. The following selection should be borne in mind:

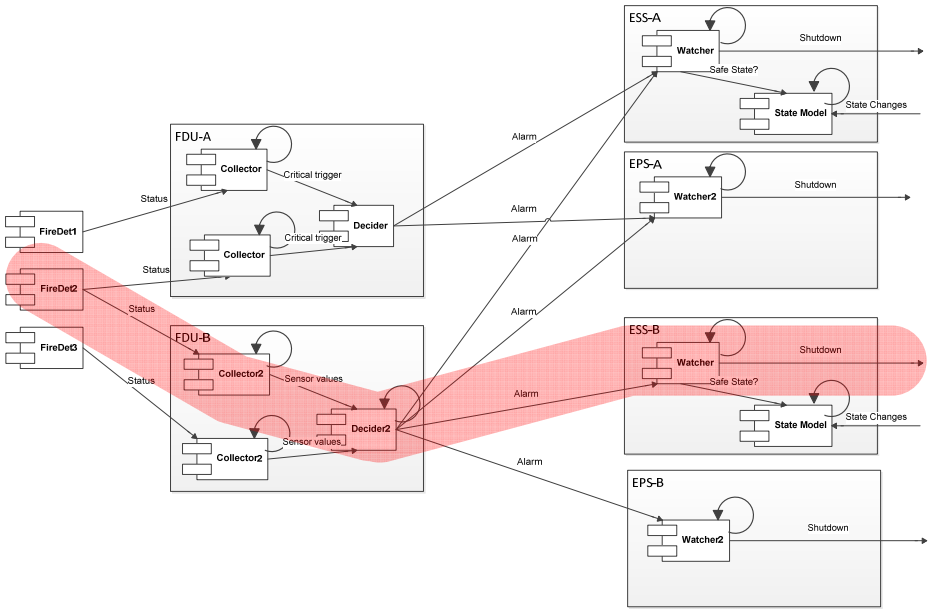


Figure 4 Identifying the TUB

Here, the modules are marked in architecture diagrams, as it were, if they are involved in TUB. This is further elaborated in section 3.6 *'Identifying failing task execution in the architecture'*.

- Choosing the right demarcation of the TUBs based on the TUB's internal characteristics (such as development organisation, process persistence, etc.) and externally perceptible effect of the TUB. This step also marks the parts of the modules that are considered and are left out of consideration for the specific task execution. Schematically shown (two TUBs marked in blue) as follows:

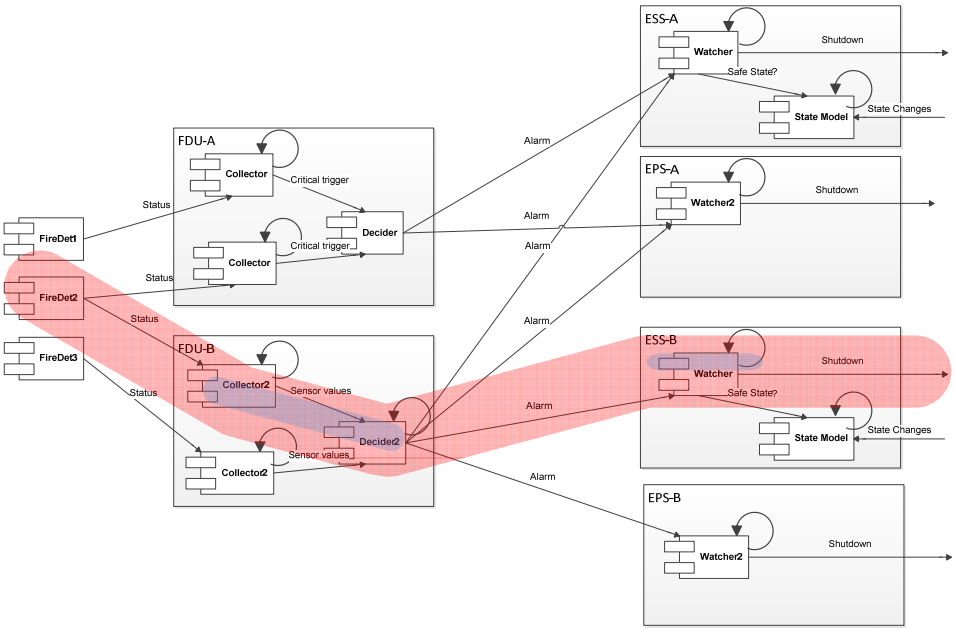


Figure 5: Note the blue shaded area at FDU-B and ESS-B.

This is elaborated in section 3.7 *'Selecting the TUBs'.*

3.2 Process and working method

3.2.1 Staffing

For qualitative modelling of software, the software architect and a RAMS specialist with significant experience in software architecture are required. The RAMS specialist must be able to discuss in terms of content the good division of the software into TUBs, with a realistically achievable quantification of these TUBs in mind. This requires a good understanding of architectural concepts and, on the other hand, a good understanding of how TOPAAS works.

3.2.2 Working method

The qualitative analysis is easiest to perform as a small workshop. During the workshop, the following two roles are distinguished: architect and RAMS specialist. The role of the architect is to design the solution well. The role of the RAMS specialist is to model the reliability of the described architecture and to document its substantiation. Identifying the task executions and TUBs should therefore be a joint action.

There must certainly be room for discussion as well: the demarcation of TUBs is not a deterministic process and several valid divisions are conceivable. The discussion about this, and what it means for the final modelling, is an essential part of the substantiation and should therefore also be documented.

3.2.3 Preconditions

There must be insight into the organisation of the overall system that delivers the RAMS performance, the presence of design knowledge (in written or human form) and the underlying choices.

3.2.4 Documentation

Information: The outcome of the qualitative modelling (fault tree model with task executions, division into TUBs) and all choices that have been made must be documented.

3.3 Definitions and concepts

3.3.1 Task execution

To make software behaviour in the fault tree tangible, the concept of task execution has been introduced. Task execution is defined as:

Requirement: Task execution is the execution of tasks by software where the execution can be observed externally and it is possible to determine whether it is carried out correctly or incorrectly.

Examples of task execution are:

- determining water levels based on working sensors;
- deciding to close a flood barrier based on a water level;
- activating the firefighting system based on a signal from a fire detector.

Task execution is the equivalent of human action or electrical switching. For the execution of tasks by people, one trained person is sufficient. For task execution by electrotechnics, a working electrical circuit is required, which is modelled loosely. Task execution by means of software and computer resources should therefore also be modelled.

TOPAAS assumes that software fails if the RAMS performance of the installation is compromised (as described in the top event) or that the result, decision or control of the software is absent, incorrect or untimely.

This refers explicitly to the top event in the fault tree: a fault in the software certainly does not have to mean a failure of the software. Software can still exhibit desired behaviour despite faults. This is certainly the case if there is fail-safe behaviour, where software faults always produce safe behaviour. These errors may then have a negative impact on the availability of the installation. TOPAAS does not consider a TUB with faults as a failure in its (safety) task if it displays fail-safe behaviour in the specific situation. It is an important part of the failure definitions in the fault tree to distinguish between the availability performance and safety performance of the installation.

3.3.2 TUB

In TOPAAS, a task execution block (TUB) is defined as follows:

Information: A TUB is a delimited set of logical lines of programme code (or graphical equivalent) that realises a task execution, where the behaviour is observable and the internal characteristics are the same.

TUBs are specifically testable cross-sections of the source code that are related to task execution, but may not correspond to the format commonly used in software engineering for breakdown into packages, modules or functions.

The following apply here:

- A clear demarcation can be identified with respect to other pieces of code;
- There is a recognised functional objective in the system as a whole;
- Verifiable quality characteristics are present, testable and observable;
- The internal characteristics (development team, production process, development tools used, etc.) are identical.

A TUB can therefore consist of part of a software module, but also of a collection of modules that are spread across various physical PLCs. The core issue is that a TUB is distinguishable and has some degree of independence (in its failure behaviour). In practice, a TUB is a cross-section of one or more software modules, where the task execution of that TUB is externally observable. The set of logical lines of code is considered in a very broad sense:

- a TUB can be part of a specific module (e.g. a specific function);
- a TUB can be all the software on a dedicated hardware component (PLC node);
- a TUB can be all the source code within a process or executable on a mainframe;
- a TUB can be a cross-section of the source code of a collection of collaborative processes in a system.

Examples of TUBs are:

- All the source code that determines the water level on the basis of a number of sensors;
- The source code which, on the basis of a certain water level, decides to close a flood barrier;
- A piece of source code that controls the fans based on the measured air quality;
- A piece of source code that puts a fan into emergency operation when the fire alarms become active.

3.4 Input documentation requirements

3.4.1 Provisional fault tree requirements

The provisional fault tree is important because the recognition of task execution starts there. At specific points in the fault tree, it is observed that software failure affects the RAMS performance of the structure. The definition of these basic events is subject to TOPAAS requirements to ensure that it also concerns failing task execution.

The core of this requirement is that it must explicitly concern the execution of tasks that fail. The main concern is that ambiguous definitions of failure are used, as a result of which the probability of failure is not delimited. For example, a single first-order basic event *'Software error'* is too general for analysis. The point is that the fault tree explicitly describes what the software is supposed to do (task execution), but the wrong handling of it causes the top event. Ideally, the failure mode is mentioned in the fault tree at the level of *'PLC does not control valve'*, after which it is further broken down in the PLC hardware and the task executions of the PLC software.

The fault tree should initially name software failure specifically, but not elaborate in detail on the underlying causes. It is up to the qualitative analysis to complete the fault tree further with identified TUBs as underlying cause. The most logical

abstraction point for the fault tree is when a sensor gives a signal and the software is supposed to control an actuator.

3.4.2 Requirements for software structure diagrams

If people want to make an analysis of the software, they must have a set of 'structure diagrams' to serve as a basis for the further development of the fault tree. As with mechanical engineering and electrical engineering structure diagrams, the architecture of the system must create a consistent picture of the cohesion of all the objects involved in each mission. Important questions are, for example, whether there are single points of failure and which critical chains determine the specific behaviour of a structure.

In the absence of such structure diagrams, they should be drawn up on the basis of interviews or experiments with the system. It goes without saying that such reverse-engineering activities must be qualitatively guaranteed and described in terms of content.

The structure diagram must therefore provide insight into how the software is structured. A specific basic event is identified in the fault tree. This is the failing task execution that is further investigated on the basis of the architecture. In order to make this possible, the structural plans must provide insight into the following issues via three types of overviews:

- Which software components make up the running system: processes, handlers?
- How is a task execution built up from processes in terms of software? In particular:
    - How do the processes work together to achieve the intended result?
    - How do processes communicate with each other?
    - Are there parallel or redundant processes?
    - Are there watchdogs?
    - Are there on-demand processes?
- Which source code runs in which processes?
- What is the failure behaviour of components? Is that functionality or realised at a 'higher' level?
    - fail-safe behaviour is present in a component,
    - in the event of failure, a component stops working: redundancy or a watchdog give rise to alternative success scenarios.
- What hardware does the task execution depend on?
- What does a software module depend on in terms of resources (CPU, memory, disk) and with which other processes does it share them?

With this information, the task execution can be divided into TUBs.

Contrary to other technical disciplines, there is not yet a uniform drawing technique that automatically covers all these aspects. However, there is a growing consensus in the ICT sector that architectural descriptions consist of multiple views. A view describes a system in a specific way with a particular purpose in mind. Because multiple parties often need different types of information, this means that different views of the same system also need to be created, with the specific characteristics explicitly highlighted.

Information: The ISO/IEC/IEEE 42010 [2] specifies the generic preconditions for system designs based on different views, as well as the rationales for documentation of design decisions.

To give more substance to this, the 4+1 Kruchten method [3] can be used, which is further elaborated in Appendix B: Use of the 4+1 model (informative).

3.5 Recognising failing task execution in the fault tree

Identifying failing task execution in the fault tree is simple, provided that the requirements set for failing task execution are clearly described by the basic events. If a single first-order basic event is included with the name 'software fault', then of course it needs to be further elaborated to the specific failure forms that are important for the top event of the fault tree.

A clear distinction has to be made between the different task executions in which software is involved. Controlling a valve, for example, is a task execution (the happy flow), but dealing with a valve failure is explicitly considered by TOPAAS as a different task execution (the unhappy flow). Even if the valve control is a single piece of software. The reason for this is that the often complex unhappy flow is only important if the valve fails, while the much simpler happy flow is used much more frequently and is naturally many times more reliable as a result. Error handling by means of software is further elaborated in section 3.8.3.

Requirement: The identified forms of failing task execution must be explicitly documented.

These failing task executions are further elaborated via TOPAAS and it is important for external reviewers (e.g. Rijkswaterstaat) to have access to the decisions that underlie them.

3.6 Identifying failing task execution in the architecture

In order to identify the failing task execution in the architecture, it is first necessary to define what is meant by the architecture. We will examine below how to identify the task execution in such a description.

Here a method is worked out that seems to be most widely applicable. Specific organisations of processes by suppliers may give rise to a different form of architecture description that is not easy to match up with. For TOPAAS this does not matter, as long as the division of failing task execution to TUBs remains possible and the TUBs can be linked to chunks of software (modules) with internal and external metrics.

By combining all the views of an architecture, an image can be constructed of how the system works, similar to a hydraulic, electrical or mechanical construction diagram. These construction diagrams must therefore either be provided by suppliers on delivery or must be determined by a software analyst in retrospect on the basis of all the other documentation available.
Once the architecture is described, linking the failing task execution to the architecture is fairly simple. By following the task execution through the architecture and identifying which parts of modules/source code are affected, the TUBs can be identified. Various scenarios can also be identified that lead to success.

For example: if the failing task execution indicates 'no locking command given at high tide', one should look in the scenarios for all the ways in which the locking command can be triggered.

There may of course be multiple scenarios that lead to success. Redundancy and fail-safe behaviour are typical examples of multiple scenarios that can lead to success. In the final fault tree, this will lead to an AND gate in which all the redundant scenarios must fail.

Because TUBs have been identified, it is also possible to determine via the physical view on which hardware the specific scenarios depend, so that the hardware can also be specifically included in the fault tree.

A note must be made concerning the inclusion of hardware components in the fault tree. Hardware in IT, such as firewalls, routers, SANs and NASs, often contains firmware. In theory, firmware can have a significant influence on the performance of a system. TOPAAS implicitly assumes that for such hardware, the probability of failure includes the software, unless the task execution directly controls such hardware (for example, opening or closing a port on the firewall). If the task executions only use such hardware as part of the solution, then as a rule only the hardware+failure is modelled in the fault tree and the software on this hardware does not need to be modelled in TOPAAS.

Requirement: The following items must be documented:

- the description of the system architecture;
- the links between the failing task executions and the scenarios, including the relationships between them (whether scenarios are redundant);
- the identified source code per task execution;
- the dependence of the specific scenarios on hardware (if not already included in the fault tree).

3.7 Selecting the TUBs

It is wise to define the TUB at a good level of abstraction by taking the task execution of software as the basis.

Software modules and TUBs cannot be related to each other one to one. After all, the code must be related to the task execution, where that task execution is part of the fault tree. In many cases, the module is internally coherent (e.g. concerns control of a valve) but the different parts of the module are involved in different task executions. For example, one part may be involved in controlling a valve and another part involved in handling error messages from the same valve. Software that cannot be related to the task execution and cannot disrupt it is not part of the TUB. In many cases, a TUB will therefore contain parts of one or more software modules. In practice, a TUB is therefore a cross-section of one or more software modules that can be related to a specific task execution.

TOPAAS deliberately looks at software in a relatively abstract way: it is concerned with the software and not the underlying hardware or firmware/operating system. The TUBs used abstract from the operating system, internal software structure, libraries, drivers and network connections. This is because an OS, driver or library does not usually fail spontaneously, but more likely fails because the call from the application/module is not correct. With the current development environments, a

Remote input/output or a Remote Procedure Call is also so transparent that communication over networks, except for the already modelled hardware failure, is actually no longer a reason for failure. The failure is more likely to lie in the poorly designed communication between modules than in the fact that communication takes place via a network.
For each failing task execution, the relevant source code is identified. In this step, the intention is to create a clustering and refining of this source code so that the resulting TUBs can also be used in the quantitative analysis.

Information: As a rule of thumb, a TUB should include as much code as possible, unless the code is incompatible.

When clustering, the requirements for TUBs continue to apply, the most important being that the behaviour must be externally observable. This means that it must be possible to relate input and output to each other and determine whether this is correct behaviour.

The source code is incompatible in a TUB if:

- the software was created by another development team, because then many (process) parameters in the quantitative analysis are different, such as the experience of the developers and the culture in the organisation;
- the nature of the process becomes different, for example because one process is a continuous active process and the other is an on-demand process that is only active in specific situations.

One point of attention is that the source code is sometimes only a partial component of a TUB and must therefore be administered in all subsequent steps. This means, for example, that control of a valve must be separated into several TUBs:

- One TUB (cross-section source code) that controls the valve under normal conditions;
- One TUB (cross-section source code) that deals with valve failure, which is part of another part of the fault tree (i.e. dealing incorrectly with a valve failure);
- One piece of code that sends status reports to the user interface and does not participate in the failure probability analysis;
- It is wise to consider a clustering of modules as a separate TUB if it is shared with another task execution (see section 3.8.2).

Requirement: The following items must be documented:
•The division of task execution into TUBs, and the way the TUBs are constructed from source code;
•The rationale behind this division.

3.8 Practical ideas during design

3.8.1 Clever dividing up of complex task execution

One of the things that can quickly lead to an unfavourable probability of failure is the combination of a complex task execution that is rarely used. Safety systems that act on a complexly calculated limit value suffer in particular from this. If one

divides these in the modelling into two TUBs (one complex TUB that performs and reports a measurement continuously, and one simple TUB that only acts when the limit value is exceeded), the probability of failure becomes considerably more realistic. Especially if the complex calculation is subject to good monitoring, extremely good reliability can be achieved here.

3.8.2 Dealing well with shared modules

Some systems have several modes: regular mode (actions based on a limit value), an emergency mode and/or a maintenance state (where actions are not performed fully automatically). Good modelling of these modes is pretty essential: a task execution that is rarely triggered naturally has a worse probability of failure.

Take, for example, a ventilation system that is used both for climate control in a tunnel and for emergency ventilation in the event of a fire. Now you can apply the input 'emergency ventilation' to a ventilation system, which will probably never be used and hardly ever tested, or in the event of a fire you can simply give the ventilation a setpoint as well. The latter case has the advantage that the knowledge about the 'why' of the setpoint is limited to a single module. The core issue is that the knowledge about such a state must have a minimal spread over software modules. Because the handling of a setpoint by a ventilation system is monitored and tested a great deal, this TUB becomes much more reliable than via a dedicated input with the spread of this substantive knowledge of the 'why' a setpoint is set. By designing and modelling this correctly, a demonstrably more robust solution is created.

Here, however, the control must be looked at very explicitly to determine whether exactly the same parts of the software module are used; if the control is different, one must actually assume that the TUB must be divided up into smaller pieces. So it makes a difference whether a fire alarm system puts the ventilation into emergency operation or simply sets it to full power.

3.8.3 Handling hardware failures (cascading failure)

Many task executions are not only about software, but also concern the interaction between hardware and software. This is where the proper handling of hardware failures also becomes important. As a rule, this handling is large and complex. A situation arises that is unexpectedly different and must first be 'understood' by the software before it can be handled correctly.

TOPAAS assumes that handling hardware failures is a different task from initially controlling the same hardware, and that this is also modelled in the fault tree. For example, the failure of a valve fails in two ways:

- The control software does not control a valve, either too late or incorrectly, based on a valid request to open the valve;
- The valve does not open due to mechanical failure and the control software handles this too late or incorrectly and, for example, does not control the redundant valve properly. This is called cascading failure.

In the fault tree, this should therefore give rise to two separate basic events for the software. In practice, modelling in two separate basic events gives a sharper picture of the software's probability of failure:

- The primary control without errors (happy flow) is often thoroughly tested and in practice often has very limited scope and complexity. As a result, this software is often reliable.
- The handling of hardware errors (unhappy flow) is often less thoroughly tested and, in addition, is usually large and complex due to the many conditions and exceptions. As a result, this software is often noticeably more unreliable than the primary control of the same hardware. In most fault trees this is not a problem in practice: hardware has a high reliability (often in the order of magnitude of 10^{-4} to 10^{-5} on request). As a result, handling hardware failure badly no longer has a noticeable impact on the ultimate reliability of the task execution.

4 Quantifying failure

4.1 Quantification process

4.1.1 Staffing

For the quantification step based on TOPAAS, the following persons are preferably present:

- a preferably independent RAMS specialist with lengthy experience in software architecture;
- the software architect;
- the developers;
- the project leader;
- the project's quality manager.

4.1.2 Working method

Section 4.2 contains the TOPAAS questionnaire. The working method for scoring the TOPAAS questions depends on the approach:

- If it concerns an internally driven TOPAAS analysis, a workshop where the questions are answered and discussed together is sufficient. Consensus on the answers and their consequences may then be more important than the TOPAAS estimate. Documentation of the matters discussed is then useful, also to provide the rationale behind decisions.
- If the analysis is part of the approval of an installation, an audit-like approach is more likely to be adopted. In that case, there is a strong emphasis on providing evidence and written substantiation of the choices.

In conclusion, section 4.3 contains practical tips.

4.1.3 Preconditions for quantification

A precondition for good results is honesty and openness about the facts. TOPAAS is not primarily intended as an audit tool with *'checks and balances'*. TOPAAS forces a supplier to provide substantial transparency concerning their process. This process therefore relies greatly on the trust one has in the independent RAMS specialist. Independence from the ultimate supplier and from the client is desirable to prevent the supplier from feeling that everything is passed on to the ultimate client one to one.

An integrated audit framework, which measures what documentary evidence is present, has value for the process of TOPAAS analyses. The presence of documentary evidence, which is also covered by regular *Quality Assurance or Product Assurance* provides more external confidence in a report. Because TOPAAS focuses on the quality of a product in terms of content, the documentary evidence should be assessed on content; however, the absence of documentary evidence does not lead to a lower score. Each score must be supported by documentation. This may be documentation drawn up in the context of the TOPAAS investigation.

4.2 TOPAAS questionnaire

The TOPAAS score is determined by answering one question per aspect. The answer determines the aspect score. In total, there are 15 aspects, i.e. 15 questions, that

have to be answered per TUB[1]. This chapter describes the aspects and also describes the minimum level of evidence and documenting.

TOPAAS is designed to deal with uncertainty: if nothing is known and additional investigation is practically impossible, the option *'Unknown'* should be chosen. As a rule, however, this does result in a lower probability of failure for the TUB. If nothing at all is known about a TUB, it concludes in a probability of failure of one. This is extremely conservative, but does correspond with the perception of the expert group. The more is known, the better the probability of failure of a TUB becomes.

If there is the possibility to do reasonable research, this should also be done. In practice, one could only score *'Unknown'* if the TUB consists of COTS software components to which a supplier refuses to give access, or with old legacy systems where the project documentation can no longer be traced. Scoring *'Unknown'* for a question does oblige the person filling in the score to explain why he opted for this score:

- why it was not reasonably possible to perform scoring;
- why it is unlikely that there can be a worse (numerical) score than *'Unknown'*.

4.2.1 Development process

The use of IEC 61508 has a significant impact on the quality of the final product. However, it is deliberately estimated more conservatively than suggested by the IEC: based on this measurement, a SIL-4 system will have a probability of failure of 10^{-3}. This is because the causal relationship between the SIL and the reliability of the system has not been scientifically demonstrated.

1 The development process complies with one of the SILs in IEC 61508		
1	Unknown, the development process does not demonstrably comply with a SIL.	0
2	Development process demonstrably does not comply with a SIL due to the use of Not Recommended Practices.	½
3	Development process demonstrably complies with SIL-1.	-½
4	Development process demonstrably complies with SIL-2.	-1
5	Development process demonstrably complies with SIL-3.	-2
6	Development process demonstrably complies with SIL-4.	-3

Examples of *Not Recommended Practices* under IEC 61508-3, table A.2 [1] are:

- using artificial intelligence;
- dynamic reconfiguration of the process allocation based on available resources. N.B.: this also happens in practice with solutions in virtualised server environments and cluster environments.

If one does not demonstrably comply with a SIL, one automatically falls back to the level *'Unknown, the development process does not demonstrably comply with a SIL'*. Demonstrable compliance with the requirements of a specific SIL for a TUB must be determined independently, using the relevant requirements set out in IEC 61508. This can be part of a TOPAAS assessment, but it can also be the result of an

[1] Some analyses may show that many tasks are performed by the same (impossible to divide further) software entity. If there are multiple TUBs, the score per TUB will be the same.

audit. This assessment must have been carried out by an independent party on the specific project that has generated the TUB.

On the basis of the TOPAAS preconditions, at a minimum, a defined quality process must be employed. If there is no basic quality assurance in the process that is appropriate for software development, then TOPAAS is not applicable and the entire questionnaire may not be used.

It should be noted that IEC 61508 has a certain status and is not non-binding. It is a *common-practice* description established by the industry itself of how suppliers are expected to operate. In fact, it has become a bar that lawyers compare suppliers to, in order to determine whether they are culpably negligent. Not following it is therefore extremely unwise because the supplier would not follow the industry's established common practice. When building safety-critical systems, it is therefore wise to follow this standard, or its specific implementations.

With this aspect, it is very explicit that no 'arithmetical optimisation' is allowed: fill in the SILs that have been realised. Following a SIL-3 process but filling in a SIL-2 process administratively in the TOPAAS questionnaire in order to achieve a better result arithmetically is seen as not filling in the list truthfully.

Requirement: The documentation must consist of a reference to the reports of an independent party (another organisation), where the specific TUBs are explicitly identified and examined in the report. For COTS TUBs, the certificate must be requested, explicitly stating that the certificate must have been issued with the same version number and all 'installation conditions' must be explicitly addressed.

4.2.2 *Use of inspections*

Inspections of documents and code have a very strong influence on the quality of a TUB. In practice, structural reviews are often more effective (and cost-effective) than testing. Because IEC 61508 already requires these reviews at SIL-3 and SIL-4 levels, and recommends Fagan inspections, they are scored differently at those SILs.

2 Use of Inspections			
		Normal	**SIL-3 / SIL-4**
1	Unknown	0	N/A
2	No inspections carried out	⅓	N/A
3	Demonstrable inspections/reviews carried out on designs and code	0	⅓
4	Demonstrable Fagan inspections carried out on all designs, code and test documents	-½	0

Demonstrability of the (Fagan) inspections is evident from several elements:

- The intention to carry out inspections/reviews must be evident from the quality plan, project plan or Software Development Plan (SDP), further substantiated with a process description of the inspection/review process.
- Correct execution must be evidenced by one of the following supporting documents for each object to be reviewed:
    - review forms/minutes of review sessions;

- signing of documents as a reviewer;
- approval for release by a reviewer by means of electronic documentation, such as a version management system, document management system or Wiki.

For a description of Fagan inspections, see Appendix D.

Requirement: The expectation is that (a reference to) the quality plan, project plan or SDP will be documented. The process description of the inspection/review process is also part of the documentation. Other objects are accessible for sampling.

4.2.3 Volume of TUB changes

A design that is regularly exposed to large-scale changes is more likely to contain faults. This is because changes are not fully thought through or incompletely implemented. The chance increases that changes in design documents are implemented, but not (completely) correctly implemented in underlying documents.

3 Volume of changes compared to original TUB design / requirements package		
1	Unknown	0
2	Very frequent or a few fundamental changes	⅔
3	Few changes, with extremely limited impact	0
4	No changes	-⅓

At the heart of this aspect is the collective scope of the changes and the impact of the changes on all the resulting design documents. The aim of this aspect is to determine the dynamics of significant changes.
Fundamental changes are understood to mean:

- Architectural changes, such as shifting responsibilities from one module to another;
- Change in basic design assumptions, such as the inherent safety of subsystems, etc.;
- Change in basic design concepts, such as the safety philosophy;
- Changes to the preconditions under which the system must operate.

The indication *'very frequent or a few fundamental changes'* means that more changes are being made than the organisation is set up to implement (the change cadence). For this purpose, the following three topics have to be considered:

- The process organisation: the *'Agile'* approach can implement a large number of changes in a much more controlled way than a *'Waterfall'* process organisation;
- The agility of the organisation: a small close-knit team can deal with changes more easily than a large team;
- The effectiveness of the change management process and related controls. In an organisation where there are very strong controls on the content of work packages, it is easier to implement adjustments properly than in an organisation where controls are absent.

Based on the above points, the normal change cadence can be determined. This typically varies from once a year to once every two weeks.

It should be noted that when you start all over again with the design and the resulting detailed designs, etc., this change cadence starts all over again. If

principles and concepts change and the development team decides that the design of the product should be completely redone, this is seen by TOPAAS as a new design without changes.

The volume of changes can be demonstrated in different ways:

- By categorising the impact assessments of the change requests;
- By means of registrations from any change management process that may be present;
- By making a difference between the first version of a design/implementation and the final version.

Requirement: In the reporting, the choice must be substantiated by a count of the changes. However, the changes must be transparent for possible inspection.

4.2.4 *Culture and collaboration*

Culture is a not-to-be-underestimated part of collaboration and the effectiveness of that collaboration. In particular, the effectiveness of the collaboration has a significant impact on the probability of failure of a TUB. To score this aspect, the culture in the specific development project organisation of the software is important.

4 Culture and collaboration		
1	Unknown	0
2	Rules-based organisation	⅓
3	Goal-oriented organisation	0
4	Self-learning organisation	-½

This characterisation is based on the definition of culture and collaboration as defined in the IAEA TECDOC-1329 [5]:

Type of organisation Features	**Rules-based organisation**	**Goal-oriented organisation**	**Self-learning organisation**
View of mistakes	Blaming staff instead of listening and learning	Mistakes lead to more checks and training	Mistakes are seen as opportunities to learn and improve
Time focus	Short term is the most important	People are rewarded for exceeding goals, regardless of the longer-term consequences	Short-term performance is analysed to improve over the longer term
Role of managers	Managers set rules and press employees to achieve the set goals	Managers use techniques such as 'Management by Objectives'	Managers coach people to improve their performance and support collaboration

Type of organisation Features	Rules-based organisation	Goal-oriented organisation	Self-learning organisation
Dealing with conflicts	Conflicts are rarely resolved and group competition remains present	Conflicts are discouraged in the name of teamwork	Conflicts are resolved through mutually acceptable solutions
View of people	People are components in a system	Realisation that people's behaviour affects their performance	People are respected and valued for their contribution

The hard evidence for this is a tricky subject. One has to rely on a self-assessment of the organisation, possibly supplemented with anecdotal evidence.

Requirement: The report should make use of the above framework in its justification.

4.2.5 *Education level and experience of development team*

Experience in the domain is an important prerequisite for understanding the requirements and preventing common mistakes in a specific domain.

5 Education level and experience of development team		
1	Unknown	0
2	No knowledge of system development for the specific domain (unconscious incompetence)	1
3	Insufficient knowledge of system development for the specific domain (conscious incompetence)	½
4	Desired knowledge of system development for the specific domain (conscious competence)	0
5	Excellent knowledge of system development for the specific domain (unconscious competence) and extensive experience	-½

The desired experience focuses mainly on the knowledge of the specific domain of people involved in the requirements analysis, architecture, design and test plans. Domain explicitly means an object domain according to [6] and [7], or a specific domain such as a system for a storm surge barrier, lock, moveable bridge or tunnel. The following requirements apply to the knowledge:

- The domain knowledge must be up to date: if projects are more than five years old, knowledge of the domain may have diminished too much or even become outdated;
- The domain knowledge must be at the right level of abstraction: someone who has done programming work for a lock is not necessarily suitable for a requirements analysis of a lock;
- System development knowledge must be appropriate to the nature of the system: this includes necessary knowledge of formal methods, testing methodologies, test coverage and documentation standards;
- The knowledge must also actually be used for implementation (requirements analysis, design, testing, etc.) or for quality assurance, for example by means of reviews of these deliverables.

For excellent knowledge and considerable experience, one should think of requirements analysts, designers or testers who have demonstrably worked for years on different projects in the domain at a comparable level of abstraction.

The knowledge and experience can easily be demonstrated by identifying the specialists with domain knowledge and recording the following:

- The CV of the employee(s) showing their domain knowledge;
- The documents for which the employee(s) have been responsible, both in a creating role and a reviewing role.

Requirement: A score in options 4 or 5 requires explicit documentation of the required knowledge.

4.2.6 Collaboration with the client

The purpose of this aspect is to prevent sub-optimal solutions resulting from:

- a client not understanding what to order from the software developer;
- poor design decisions being made because the interests of software development are considered subordinate to other disciplines.

In more complex projects, such as building combinations, the client is often a complex group of actors. From the perspective of TOPAAS, we think of the direct client of the software development team, which in a combination is the combination itself, or a main contractor if only software is being developed.

6 Collaboration with the client		
1	Unknown.	0
2	Not closely involved client with limited IT knowledge, highly contractually/financially driven client.	½
3	Indirectly involved client with moderate IT knowledge.	0
4	Heavily involved client with sufficient knowledge and open dialogue where the client is prepared to change the overall architecture if this improves the reliability of the software. There is a systems-engineering approach for the entire development of the system.	-½

There are necessary preconditions for larger objects (such as bridges, locks and tunnels) that make a real systems-engineering approach possible:

- During the global design of the installation, designers with IT knowledge are involved to ensure that no unmanageable physical design is created;
- Software engineers are involved in the trade-off analyses for the hardware to be controlled;
- Design/purchase of hardware is only completed when the designs of software are also ready and demonstrate that the hardware is also controllable;
- Changes are considered to be multidisciplinary, and are not dealt with in a monodisciplinary way;
- Problems in products/designs are considered to be multidisciplinary.
- The demonstrability of this point consists of:
- CVs of employees of the client, who are involved in the global design;
- The presence of an explicit systems-engineering approach, for example by documentation of the trade-off analyses;
- The multidisciplinary handling of changes and problems in the design, for example in accordance with the SE guideline.

Requirement: The documentation consists of the role of the designers and client with their CVs and some example changes/problems if present.

4.2.7 Complexity decision logic of the TUB

Complex software is extremely difficult to make reliable. Complexity has three important effects that adversely affect the reliability of the TUB:

- Complex software contains many paths. Due to a lack of overview of and insight into the exact operation of the TUB, mistakes can easily be made;
- It's difficult to make a good overview of complex software, making modification and bug fixing much more likely to result in adverse side effects;
- Complex software is extremely difficult to test, which significantly reduces the effectiveness of testing.

7 Complexity decision logic of the specific TUB		
1	Unknown.	0
2	Decision logic is extremely complex (contains many branches and exceptions), McCabe index greater than 60.	½
3	Decision logic moderately complex (contains some exceptions), McCabe index between 30 and 60.	0
4	Decision logic is fairly simple (contains some very isolated exceptions), McCabe index between 10 and 30.	-⅓
5	Decision logic and fault recognition are very simple, McCabe Index less than 10.	-½

Formally, the approach is that it is purely about the TUB. As mentioned with the qualitative analysis, this means that a distinction must be made between the often simple happy flow and the generally more complex handling of errors. After all, they are different parts of the source code. This separation often lies in the middle of the source code of a module, which means that simply having tools calculate it does not work. This does not matter for simple modules: if the McCabe index for the module is already less than 10, it is also less for the specific TUB.

With larger modules, where several TUBs come together, this is more difficult. Counting a large number of paths manually, in particular, is extremely difficult. So the most pragmatic approach is to subtract the McCabe index of the easy-to-determine TUBs from the McCabe index of the complete module (provided they are completely disjunctive). It is also possible to remove non-critical software from the analysis in this way. This approach is fairly simple and can even be partially automated.

It should be noted that complexity is not always avoidable. Domain complexity (the complexity of the problem that needs to be solved) can sometimes translate quite directly into a complex solution. In such situations, it would be striking if the software-based modelling of a very complex problem resulted in a very simple solution.

A complex solution to a problem with limited domain complexity may be a sign of poor design. The implementation of *defensive programming* and fail-safe behaviour may have some negative effects on complexity, but it is certainly a bad sign if the

solution is much more complex than the domain. As a rule, this results from a design which does not comply with the principle of *'low coupling, high cohesion'*.

It is possible that the software's source code itself may not be available. One can then consider counting the number of paths through the software in the decision logic, as an approximation. This can be done by looking at the externally observable behaviour of the TUB (or an available specification of this) and making an estimate on this basis.

Although there are several formal mathematical definitions of the McCabe index, the simplest one is:

McCabe Index = 1 + D

Where D is the number of decision points (if statements, while statements, switch/case statements).

This definition can be applied to both source code and described decision logic of a TUB. It should be noted that structural monitoring during design and construction of the complexity of software is reasonably easy to set up by means of tooling.

Requirement: The documentation of complexity consists of three parts:
•An overview (generated by a tool) of the complexity of all modules;
•An overview of the complexity of each identified TUB, subdivided into the (parts of) software modules it consists of;
•An explanation of the distribution of complexity across the TUBs, if a module is part of several TUBs. Parts of the code, which are completely disregarded, must also be explained.

4.2.8 *Size of TUB*

The size of the TUB determines the probability of failure. Statistically, the chance of faults increases as the code becomes larger and the clarity and comprehensibility decreases, resulting in a greater likelihood of faults.

8 Size of TUB (Lines of code)		
1	Unknown	0
2	More than 50,000	½
3	Between 10,000 and 50,000	⅓
4	Between 5,000 and 10,000	0
5	Between 1,000 and 5,000	-⅓
6	Less than 1,000	-½

If software is made for a specific purpose, it is fairly easy to count the lines involved in realising a TUB. At the design stage or in the case of COTS, it is often impossible to determine the number of lines of code. However, this is often still possible using function point analysis combined with the key figures of the IFPUG (International Function Point Users Group).

Programming language	**Average lines of code per function point**
Basic Assembly	320
Macro Assembly	213

C	128
C++	53
FORTRAN	107
Pascal	91
PL/I	80
Ada83	71
Ada95	49
Visual Basic	32

When working with a graphical programming environment, one graphical element (a block or a connecting line) is considered to be one line of code. If code is generated from a graphical programming environment by a reliable generator (preferably certified for IEC 61508, ISO 26262 or DO-178B/C), then the graphical environment determines the scoring of this item and not the generated code. Examples of certified graphical environments that can generate code are Scade and SimuLink.

The documentation of the size of TUBs consists of four parts:

- An overview (generated by a tool) of the size of all modules in numbers of lines of code;
- An overview of the size of each identified TUB in numbers of lines of code, subdivided into the individual (parts of) software modules it consists of;
- An explanation of the distribution of the scope across the TUB, if a module is part of several TUBs. In addition, parts of the code, which are completely disregarded, must also be explained;
- In the case of code generation: a substantiation of the reliability of the code generator, preferably by means of certificates.

4.2.9 *Clarity of architectural concepts used*

A good architecture of the total solution is an important prerequisite for making software easy to understand, buildable, testable and maintainable. The other extreme is poorly organised code, sometimes also known as spaghetti code, which does not name clearly defined responsibilities. This makes the result difficult to understand and extremely difficult to test.

9 Clarity of architectural concepts used		
1	Unknown.	0
2	No clear demarcation of tasks and responsibilities named for modules in design.	½
3	Although tasks and responsibilities are defined in broad terms, they are not being followed up in development.	⅓
4	A separation of tasks and responsibilities between modules has been described, which respects the principle of 'low coupling, high cohesion', but this has been passively monitored during the development process.	0
5	A clear separation of tasks and responsibilities between modules has been described based on valid documents, which respects the principle of 'low coupling, high cohesion', and this has been demonstrably actively monitored during the development process.	-½

low coupling, high cohesion

Crucial ingredients are the explicit definition of the architecture and the essential requirement of 'low coupling, high cohesion'. *'Low coupling, high cohesion'* falls under *separation of concerns*. This design principle requires that each software

module forms an internally highly coherent whole with a clearly demarcated function in relation to the rest. The modules communicate with each other but extremely sparingly and via an agreed interface.

Passive and active monitoring
Another point is the monitoring of the architecture. Passive monitoring means that an architecture has been described and used as the basis for designs, but that there are no explicit checks as to whether modules have actually been built according to that architecture. Active monitoring means that design and test documents are explicitly related to the architecture and that reviews and inspections also take place on these documents as to whether they have actually realised the architecture. This also includes demonstrably active supervision of the development team by the architect.

The following must be demonstrable:

- The description of the architecture, including the underlying decisions and considerations (rationale) as to why certain architectural choices have been made (necessary for options 3, 4 and 5);
- The explicit reference of the architecture in the underlying design documents (necessary for options 4 and 5);
- Checks on the correct interpretation of the architecture in those underlying design documents, for example by making it an explicit part of reviews and inspections or by making it a separate activity (necessary for option 5).

Normally, the architecture should largely be included in the documentation already (part of the identification of TUBs). In order to demonstrate that the architecture has actually been applied, it is sufficient to refer to the design documents in the substantiation of the scoring. The documentation of the monitoring depends on the form of monitoring:

- If the monitoring is anchored in the review process, it is sufficient to demonstrate that the architecture function is structurally involved in the reviews;
- If the monitoring is separate, the resulting registrations must be available as evidence;
- If the architect plays an active monitoring role in the building process, this should be evident from reports of discussions or design documentation.

4.2.10 *Using a certified compiler*

Compilers have an important influence on the quality of software. The most obvious requirement for a compiler is that it reliably converts programme code into executable machine code. But compilers also have qualitative properties, such as explicit limitations in the constructions used in programme code.

10 Using a certified compiler			
		Normal	**SIL-3 / SIL-4**
1	Unknown.	0	N/A
2	Using a random compiler.	⅓	N/A
3	Using a compiler that the developer has lengthy experience with.	0	⅓

4	Using a certified compiler in combination with a validated safe subset.	-½	0
5	Using a certified compiler in combination with a validated safe subset with corresponding calibration sets and test protocol to calibrate/test the compiler, which happens systematically for each new version of the compiler.	-⅔	-⅓

The safe subset is a subset of the programming language and is partly determined by:

- the programming language, where unsafe features are excluded from the language;
- the specific (version of the) compiler, because the scope of certification of the specific brand, type and version of the compiler determines which constructions of the source code are converted in a certified reliable manner into executable programme code.

The safe subset is therefore both language-dependent and compiler-dependent and must be explicitly determined for each installed (version of the) compiler.

The calibration tests and the corresponding test protocols are mechanisms to check whether a compiler is actually properly configured and is being applied well in its working environment. In practice, certain programme code optimisations must be switched off. The compiler manufacturer's calibration kit checks this by compiling a piece of source code which should generate a predefined output.

There are very few certified compilers. This is also recognised by IEC 61508 by allowing compilers with *'increased confidence from use'* as an option for SIL-3/SIL-4 systems. When developing at SIL-3/4, a certified compiler is described as an option in IEC 61508. IEC 61508 part 7 writes specifically about being certified:

'The certification of a tool will generally be carried out by an independent, often national, body, against independently set criteria, typically national or international standards. (…). To date, only compilers (translators) are regularly subject to certification procedures; these are laid down by national certification bodies and the exercise compilers (translators) against international standards such as those for Ada and Pascal'.

It should be noted that TÜV in particular is actively certifying compilers and that C and C++ compilers are also certified.

Many PLC suppliers have a SIL-3 or Safety mode, in which only certain constructions may be used. Often these are certified by TÜV as well. This is considered a certified compiler.

Requirement: The following must be demonstrated:
•Demonstrable use of the compiler in other projects (option 3);
•The certification data of the compiler together with the corresponding subset, where the safe subset must be explicitly defined in *coding guidelines* for the project (options 4 and 5);
•Calibration reports of the compiler (option 5).
The references, certification data, subset, coding guidelines and calibration reports must be included in the RAMS report as underlying substantiation.

4.2.11 Traceability of requirements

The traceability of requirements ensures that requirements are also explicitly implemented and tested. This demonstrates that a TUB does what it has to do.

11 Traceability of requirements throughout the process			
		Normal	**SIL-3 / SIL-4**
1	Unknown	0	N/A
2	No traceability	-⅓	N/A
3	Demonstrably traceable to test scripts	0	N/A
4	Demonstrably traceable to architecture and testing	-⅓	⅓
5	Demonstrably traceable from safety-critical requirements to the code and individual tests	-⅔	0
6	Demonstrable full traceability	-1	-⅓
7	Demonstrably mathematically/logically proven correct traceability	-2	-½

The easiest way to achieve traceability is by gradually translating user requirements into test scripts and technical system requirements in the software development process. Subsequently, software modules are identified that will meet the system requirements. In this way, the user requirements can be traced back to the code. Standard techniques called 'Verification Cross Reference Index' also exist in the testing process. Here, each user requirement is explicitly linked to test scripts.

Mathematically correct proven traceability (option 7) means that model checking has been applied as to whether a behavioural equivalent has been demonstrated between the implementation and a correct (validated) specification. For the time being, this can only be realised for small-scale systems. If the links between all user requirements, tests, underlying system requirements and execution code are clear, then the traceability in the software module is complete (option 6). This may apply only to safety-critical requirements (option 5) or only partially (option 4) or to testing (option 3).

Requirement: The option with substantiation must be documented in the RAMS analysis. Part of the substantiation must be a description of the method of keeping requirements traceable.

4.2.12 Test techniques and coverage

Testing has a major influence on the ultimate confidence in the TUB. The depth and coverage of the tests are particularly important. It should be noted that this explicitly concerns the coverage in relation to the TUB's task execution.

12 Test techniques and coverage of the TUB			
		Normal	**SIL-3 / SIL-4**
1	Unknown	0	N/A
2	No documented test executed	0	N/A
3	Documented test executed, no formal test techniques used. Unknown coverage	-⅓	N/A
4	Formal test technique(s) used with low coverage	-½	⅔
5	Formal test technique(s) used with medium coverage	-⅔	½

6	Formal test technique(s) used with high coverage	-1	0
7	Formal test technique(s) used with demonstrable (measured) high coverage	-1⅓	-⅓

In formal testing, techniques from mathematics and logic are applied. The starting point is a verified model of the software design or the TUB itself if it is annotated with formal specifications. This automatically generates tests that test a desired requirement with a controlled coverage. The formal testing process makes the link between requirement and test case very clear and is therefore of great value in demonstrating software reliability. A few tools that support model-based test generation are T-VEC, Conformiq, Reactis and Unitesk. An academic tool that has been used intensively in the telecommunications world is TGV with TorX as its successor.

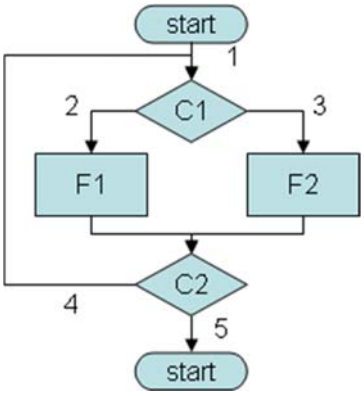
The coverage of test cases can be expressed in different ways:

- Dependent on the specification technique used:
  - decision tables based on statement, condition or multiple condition coverage;
  - process cycles (based on decision points) on the basis of 'test level 1', '-2' or '-n'' (see explanation relating to the test level);
  - combination of equivalence classes based on pairwise, triplewise, etc.
- As a percentage of paths followed during testing compared to theoretically possible paths using tools such as McCabe. This requires the use of tools during the testing process and also the availability of source code. The coverage is in this case:
  - high: 90% of the possible paths that can be followed;
  - medium: 50% of the possible paths that can be followed;
  - low: 10% of the possible paths that can be followed.
- As a percentage of the number of completed combinations of input variables compared to theoretically possible combinations of input variables. The coverage is in this case:
  - high: 90% of the theoretically possible input;
  - medium: 50% of the theoretically possible input;
  - low: 10% of the theoretically possible input.

Explanation relating to the test level

The test level is based on the principle that there are a number of outputs at decision points in the decision logic. The combination of different inputs and outputs at successive decision points therefore describe different 'functional paths' in a programme. The higher the test level, the larger the number of combinations and the higher the coverage. In addition, the application of a test level is a measurable justification for the identified test cases.

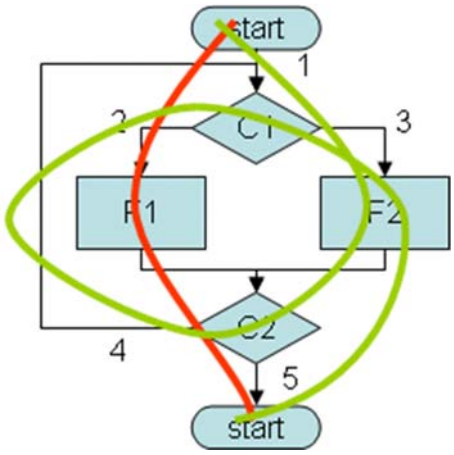
To illustrate this, a simple TUB with examples of identified test scenarios according to test level 1 and test level 2 is shown below. The example is based on the following diagram:



The following conditions apply:

- The outputs of the conditions (C1 and C2) are independent of each other;
- The operations performed (F1 and F2) can be repeated;
- Input for the TUB determines the handling in C1 and C2;
- The output of operations in F1 and F2 is measurable. This means that the behaviour of the TUB is observable.

Test level 1:



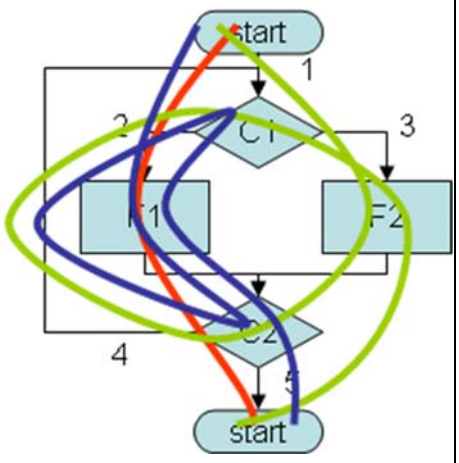
Run through each combination of outputs of one (1) condition.
Combinations:

1; 2; 3; 4; 5

As a matter of fact, these are not combinations but individual outputs. Test level 1 is the simplest form. Two scenarios are sufficient, for example:

A: 1, 2, 5
and
B: 1, 3, 4, 3, 5

Test level 2:



Run through each combination of outputs of two (2) conditions. Combinations:

1-2; 1-3; 4-2; 4-3; 2-4; 2-5; 3-4; 3-5

The scenarios from the example of test level 1 alone will not suffice:

A: 1, 2, 5 (covers 1-2 and 2-5)
B: 1, 3, 4, 3, 5 (covers 1-3, 3-4, 4-3 and 3-5)

Combinations 4-2 and 2-4 are still missing

Additional scenario required, for example:

C: 1, 2, 4, 2, 5

Test level 2 generally leads to more test cases than test level 1. It should be noted that there may be duplications, but on the other hand there are also path combinations that are not tested:

- 1, 3, 4, 2, 5
- 1, 2, 4, 3, 5 or
- 1, 2, 4, 3, 4, 2, 5.

Each subsequent test level (combinations of 3, 4, 5, etc. consecutive conditions) gives an even higher coverage.

In this way, the coverage can be formalised. The predetermination of a test level makes it possible to determine afterwards to what extent the requirements set have been met. Assessment of the test scenarios takes place on the basis of compliance with the output forecast, linked to the TUB input. This demonstrates the importance of the relationship with the functional specifications in determining the coverage. Specifications are necessary to identify the required paths and as a basis for the output forecast.

The above examples have shown that 'code coverage' does not provide any insight into the functional coverage achieved.

Requirement: Per TUB, the following must be documented and demonstrated:
•the coverage achieved;
•how the coverage has been demonstrated.
Designs, test plans and test reports must be accessible for random sampling by the client.

4.2.13 Multiprocess environment

Multiprocess environments result in processes competing for the same resources. As a result, timeliness in particular, but sometimes also stability, can be much less strongly guaranteed.

13 Multiprocess environment		
1	Unknown.	0
2	Multiple TUBs in a virtualised environment.	½
3	Multiple TUBs running independently on one piece of hardware.	⅓
4	Maximum of one TUB on a dedicated OS with a dedicated CPU.	0
5	One TUB on a dedicated CPU and memory on no OS or a trivial OS.	-⅓

Scores 2 and 3 are obtained when multiple TUBs active on one CPU are not naturally aligned with each other. TUBs are aligned with each other if they:

- control the same hardware (i.e. controlling a barrier is typically part of 2 or more TUBs: closing the barrier and opening it);
- control hardware and deal with the failure of the same hardware (i.e. controlling a valve and responding to the failure of the same valve).

Score 4 also includes: Multiple TUBS on one piece of hardware that are executed sequentially and in a well-defined way with a priori limited resource usage. In the case of varying resource usage of the TUB, score 3 continues to apply. Varying resource use already includes joint use of stack and heap space.

A trivial OS is an operating system with an explicit hard time constraint per module, for example a real-time operating system based on round-robin scheduling. This makes time behaviour deterministic, which improves reliability.

Requirement: The specific characteristics, such as the timeboxing characteristics of the OS scheduler, must be demonstrated and recorded in the architecture description. The architecture description is already part of the qualitative analysis.

4.2.14 *Presence of representative field data of the task execution*

Field data are data from the application of the TUB in other solutions. For instance, a ventilation control system that has been implemented hundreds of times in tunnels around the world and is now applied to a specific tunnel. This field data shows that the TUB has a proven track record. It may be that there is also limited practical experience in the field with the actual use of the TUB, for example in the case of safety systems. But when sufficient representative tests have been carried out, for example in Site Acceptance Testing of other applications, the data can contribute to that insight. The idea is that an intensively utilised TUB has also been thoroughly tested and used, making it naturally more reliable. Here it must be properly determined whether the current application also corresponds to the application used to obtain the field data, otherwise the reliability of the current use has not been demonstrated.

14 Presence of representative field data during task execution			
		Normal	**SIL-3 / SIL-4**
1	Unknown.	0	N/A
2	No field data or test data available, not even from (shadow) running of the TUB.	⅓	⅓
3	Limited data present and analysed during task execution by the TUB.	0	0
4	Significant amount of data present during task execution by the TUB.	-1	-⅓

5	Considerable representative field data/test data present from identical or very similar applications with the same TUB.	-2	-½

Test data and field data are experience figures gained from tests or production of other TUB applications. For example, they provide insight into the number of error-free running hours or the number of error-free test scenarios of a TUB. To be able to make statements about the TUB's reliability on the basis of field data, significant amounts of data must be available (from the supplier). In addition, it is essential that they have been obtained in situations that are representative of the way in which the TUB is used in practice.

If tests have been carried out in the real production environment for a significant period of time (shadow running), this may be regarded as field data. Important here is the realistic production environment. It may not contain synthetic or simulated test data, but may have been tested only in the actual final situation.

The significance of the amounts is shown in the possible answers to this aspect:

- In option 2, *'no data'* means that the test data do not cover realistic operational scenarios and field data are not covered by option 3.
- In option 3, *'limited data'* means that the TUB is used very little in comparable applications, less than in the order of magnitude of five times a month worldwide, or in the case of test data that the number of usable test results is low.
- In option 4, *'significant amount of data'* means that the TUB is used to some extent in comparable applications, at least in the order of magnitude of five times a month worldwide, or in the case of test data that the number of usable test results is significant.
- In option 5, *'considerable representative field data'* means that the TUB is used a great deal in comparable applications, at least in the order of magnitude of five times a day worldwide, or in the case of test data that the number of usable test results is similarly large.

Whether or not the data are representative depends on the following factors:

- The relevant task execution must actually take place on a regular basis in order to qualify as field data. A security system that has been installed in many places, but has never actually intervened, therefore has no representative field data.
- Test results must have been obtained with scenarios comparable to production situations and must also be distributed in such numbers over the relevant task execution.
- The technical environment in which the TUB is operational or tested is representative of the environment of the TUB under review. It is explicitly about representative technical environments: the TUB has to control very similar objects in the environment in order to deliver representative results.

In order to demonstrate this, the following is necessary:

- In the case of test data: there must have been a structured testing process with adequate logging of test scenarios and results or,
- in the case of field data, the data must have been obtained through a structured feedback process indicating TUB success or failure.

Requirement: The documentation should indicate:
•on which underlying data the choice is based;
•how the data collection was obtained by the supplier of the TUB;
•the justification for why the field data/test data are representative of the applications of the TUB.

4.2.15 Monitoring

Monitoring keeps track of the performance of the TUB during the task execution in the ultimate production environment. In a monitored system, good behaviour is determined, faults are detected and documented when they occur, while in a non-monitored system, faults may remain unnoticed. Essential for monitoring is that the total solution is monitored and that the correct specific task execution of the specific TU is seen and registered.

15 Monitoring of the TUB		
1	Unknown	0
2	None present	⅓
3	Limited/brief monitoring during task execution	0
4	Long-term monitoring, but infrequent task execution	-⅓
5	Long-term/frequent monitoring during task execution	-½

The monitoring focuses specifically on the TUB in question. It allows the confidence one has in longer-existing TUBs with a demonstrable track record to be reflected in the TOPAAS score as well. One can think of TUBs that have been functioning well for several years in the same object (e.g. tunnel or bridge). This factor also refers very explicitly to the operational application of the TUB in the production environment. Test data and field data are explicitly excluded from the definition of monitoring.

A more detailed explanation of the options:

- In option 3, *'Limited/brief monitoring'* means that the TUB is addressed in a limited way, at most in the order of magnitude of once a month.
- In option 4, *'Long-term monitoring but infrequent task execution'* means that the TUB is called upon to carry out its task execution at least in the order of magnitude of once a month but not more than once a year.
- In option 5, *'Long-term/frequent monitoring during task execution'* means that the TUB is used at least in the order of magnitude of five times a day, and is called upon to carry out its task execution at least in the order of magnitude of five times a year.

Requirement: The following should be documented:
•how the monitoring is organised;
•the TUB's trigger frequency.

4.3 Practical tips

4.3.1 Dealing with large groups of TUBs from the same supplier

It is possible to carry out a completely new quantitative analysis for each TUB. However, it is practical to distinguish between questions that will provide an identical picture across all TUBs and other questions that will differ from TUB to TUB.

The data that are probably the same for all TUBs within the same project are:

- Aspect 1: Compliance of the development process with one of the SILs in IEC 61508.
- Aspect 2: Use of inspections.
- Aspect 4: Culture and collaboration.
- Aspect 6: Collaboration with the client.
- Aspect 9: Clarity of architectural concepts used.
- Aspect 10: Using a certified compiler.
- Aspect 11: Traceability of requirements throughout the process.

The data that are probably different for each TUB are:

- Aspect 3: Volume of changes compared to original design/requirements package.
- Aspect 5: Education level and experience of development team.
- Aspect 7: Complexity of decision logic.
- Aspect 8: Size of TUB (Lines of code).
- Aspect 12: Test techniques and coverage.
- Aspect 13: Multiprocess environment.
- Aspect 14: Presence of representative field data during task execution.
- Aspect 15: Monitoring.

4.3.2 *Practical dealings with many different TUBs*

Some data are specific to a TUB. But TUBs are parts of one or more modules. In the substantiation of the qualitative analysis, the relationship is made between TUBs and source code. It is wise to always put the following information there in one large overview (per module with a subdivision of the TUBs):

- Aspect 7: Complexity decision logic
- Aspect 8: Size of TUB (Lines of code)
- Aspect 12: Test techniques and coverage

These are highly interrelated data that relate specifically to the source code and its distribution. This makes it easier to manage and analyse.

4.3.3 *Dealing with large monolithic subsystems*

In some designs, parts of code are created where it is difficult or impossible to isolate the different task executions. This is of course greatly at odds with the architectural principle of 'low coupling, high cohesion', but in some situations this is unavoidable.

From the analysis, such a monolithic design results in a collection of TUBs that all relate to one piece of source code. From the RAMS analysis, it is necessary to score these TUBs individually, which translates into many TUBs with identical scoring and identical substantiation.

In that situation, it is wise to remain focused on aspects that may still differ from one TUB to another:

- Aspect 7: complexity decision logic (perhaps to be determined on the basis of the established decision structure and not on the basis of the source code).
- Aspect 12: test techniques and coverage.
- Aspect 14: presence of representative field data during task execution.

As a result, the score may still differ from TUB to TUB, even though the same monolithic software achieves it.

4.3.4 *Dealing with knowledge and quality*

The four questions below are indirectly related to each other:

- Aspect 2: use of inspections.
- Aspect 5: education level and experience of development team.
- Aspect 9: clarity of architectural concepts used.
- Aspect 11: traceability of requirements throughout the process.

Aspect 2 concerns the peer reviews carried out, among other matters. Aspect 5 may place requirements for the application of domain experience in the application of reviews. Aspect 9 may place requirements on the monitoring of architectural principles in performing reviews. And aspect 11 is possibly qualitatively assured by means of reviews.

A generic underlying theme is that if reviews/inspections of documents check both the buildability/plausibility and the compliance with the requirements set in the above documents and the project has this carried out by the right people, many of the requirements mentioned are relatively easy to comply with.

5 Reporting

In the previous chapters, much has been said about the documentation in a RAMS file. The following items are expected to be included:

Requirement: A description of the architecture of the solution (the output from section 3.6).

Requirement: The identification of all basic events where software plays a role (the output as described in section 3.5), a distribution of these failing task executions over TUBs and software modules (the output from 3.7), together with a rationale behind these divisions.

Requirement: The substantiation per TUB for the probability of failure (the output from sections 4.1 and 4.2).

6 References

[1]IEC61508 Functional safety of electrical/electronic/programmable electronic safety-related systems, CEI/IEC, 1998
[2]ISO/IEC/ IEEE 42010 Systems and software engineering — Architecture description, ISO / IEC / IEEE, 2011
[3]P. Kruchten, Architectural Blueprints—The "4+1" View Model of Software Architecture, IEEE Software 12 (6), November 1995, pp. 42-50
[4]E.T.H. Brandt en R. P. Henzen, Handboeken Betrouwbaarheidsanalyse, Refis Reliability Engineering, 2005
[5]IAEA, Safety Culture in Nuclear Installations: Guidance for Use in the Enhancement of Safety Culture, IAEA-TECDOC-1329, ISBN:92-0-119102-2, http://www-ub.iaea.org/MTCD/Publications/PDF/te_1329_web.pdf
[6]ANSI/ISA-88.*.*-2010 Batch Control
[7]ANSI/ISA-95.*.*-2010 (IEC 62264) Enterprise-Control System Integration)
[8]Handreiking prestatiegestuurde risicoanalyses (PRA), Rijkswaterstaat, Oktober 2016
[9]Handreiking Bayesiaanse update, Rijkswaterstaat, December 2016
[10] Rijkswaterstaat memo: DOC-0055 Continue systemen en de faalkans van software, 28 maart 2018, J. Horstman

7 Abbreviations and terms

Abbreviation or term	Meaning[2]
Availability	This has two definitions: • the expected fraction of the total time that a system will function under given circumstances; • the probability that a system, under given circumstances, will function when triggered at any time.
Reliability	The chance of a system fulfilling its function without failure, for a certain period of time and under given circumstances.
COTS	Commercial off-the-shelf. TUBs, for instance turbine protection, which are purchased as complete products.
Failure	An event or set of events that causes a system to lose some or all of its functionality.
Fault tree	Graphical representation of the relationship between the failure of system elements and the failure of the system, expressed in the *Undesirable Top Event* (OTG). This graphical representation is often facilitated by software, which also calculates the probability or unavailability of the OTG.
IFPUG	International Function Point Users Group
IT	Information Technology
λ (lambda)	Failure frequency (dimension 1/hour)
OS	Operating System
ProBO	Probabilistic Management and Maintenance
Q	Probability of failure per question
RAMS	Reliability, Availability, Maintainability, Safety
Rationale	Description and substantiation of design decisions, in accordance with ISO/IEC/ IEEE 42010
SDP	Software Development Plan
SOBEK	Software for predicting expected local water levels
System	Coherent whole of physical components intended to fulfil a certain function. In other words: a collection of interrelated elements, which depending on the established aim are distinguishable within the total reality. Rijkswaterstaat's networks are systems, but so are their components. Each system is part of a larger whole and is therefore in fact a subsystem. So it depends on the context where the boundaries of the system are drawn.
TOPAAS	Task Oriented Probability of Abnormalities Analysis for Software
(Undesirable) top event	(partial) loss of function of a system. The undesirable top event (OTG) is the event whose probability is calculated in a quantitative risk analysis. Usually this is the failure of the main functions of the system, such as holding back high tide, allowing shipping to pass, allowing road traffic to pass, draining water, etc.
TUB	Task Execution Block
UML	Unified Modeling Language

[2] Terminology comes primarily from the Performance Based Risk Analysis (PRA) Guide [8].

Bijlage A : Alternative failure probability analysis methods

A.1 Reliability Growth Modelling

Reliability Growth Modelling (RGM) is based on the principle that the reliability of TUBs increases with the correction of faults found. The increase in reliability that emerges as a result is not actually random. The growth takes place according to a certain curve depending on the nature of the system, the way it is used, the types of faults found and the way they are solved.

When sufficient information is collected about the TUB and its failure, it is possible to derive a possible reliability growth curve. The 'possible' here lies in the fact that the models describing the growth curves are not necessarily exhaustive. Even when a TUB meets all the conditions set for the model, there is a theoretical chance that as-yet-unknown variables play a role.

In addition, the models need to be fed with statistically significant numbers of faults in order to be somewhat reliable.

And finally, the application of reliability growth models requires the necessary expertise in statistics, also when using specially developed tools.

A.2 Monte Carlo

By means of the Monte Carlo method, where the TUB is tested with random input values and where the results can be judged to be correct or in error, the probability of failure of a task execution can be determined after a given number of tests. The number of tests to be performed depends on the required probability of failure.

Based on the binomial distribution, the following table can be calculated for x fault-free tests:

	Number of faultless tests in typical use											
Probability of failure	1	10	30	100	300	1,000	3,000	10,000	30,000	100,000	300,000	1,000,000
10^{0}	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
10^{-1}	10.00	65.13	95.76	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
10^{-2}	1.00	9.56	26.03	63.40	95.10	100.00	100.00	100.00	100.00	100.00	100.00	100.00
10^{-3}	0.10	1.00	2.96	9.52	25.93	63.23	95.03	100.00	100.00	100.00	100.00	100.00
10^{-4}	0.01	0.10	0.30	1.00	2.96	9.52	25.92	63.21	95.01	99.99	100.00	100.00
10^{-5}	0.00	0.01	0.03	0.10	0.30	1.00	2.96	9.52	25.91	63.21	95.01	99.99

Table 1: Confidence level (%) at given probability of failure and number of fault-free tests/use

In this table, the required probability of failure is linked to the size of the sample/test required and the resulting confidence in this response. As a rule, a confidence level of 95% is assumed before the required probability of failure can be included in a fault tree. So, for example, to be able to demonstrate a probability of failure of 10^{-2}, at least 300 fault-free tests are needed. In this example, it should be

noted that it concerns a very simple task execution by a TUB. In setting up Monte Carlo testing for real software, determining the sample size and organising the testing are no trivial matters. It is therefore important when designing the testing to involve experienced advisers (e.g. from specialised consultancy companies or universities) to ensure that reliable pronouncements can be made.

What this example of very simple software already shows is that in the case of low probabilities of failure, i.e. reliable software, the number of tests to be performed quickly becomes considerable. If a test is fully automated, it is 'only' possible to test 15,000 cases a day with a test time of five seconds, which means that testing takes a few days to weeks. The additional difficulty with these volumes lies, on the one hand, in the turnaround time of these tests and, on the other hand, in checking whether the task execution by the TUB during the test was fault-free. When performing large numbers of tests, this check has to be automated. It requires a piece of verification software that has been developed independently of the tested TUB and needs to be validated itself. Although none of this is impossible, the total turnaround time for these tests and checks may therefore become too long in practical terms, even if the task performed by the TUB can be tested very easily and quickly.

A.3 Endurance tests

For some simple building blocks, especially the purely reactive ones without complex decision logic, it is an option to repeat the same test very often, with small variations, to show that the probability of failure is actually achieved. This inevitably raises the question of how large the sample (number of repetitions) should be in order to make a pronouncement with an accepted degree of certainty about the reliability of a TUB, which is also reflected in Table 2 *'Confidence level at given probability of failure and number of fault-free tests/use'*. Here, too, the numbers used in Monte Carlo tests with the associated practical problems are rapidly reached, and once again specialised knowledge is required for the design of the tests.

Bijlage B : Use of the 4+1 model (informative)

In order to give more depth to the architectural component, this appendix explains the 4+1 Kruchten method [3]. This method is explicitly not mandatory for TOPAAS, but is one of the most widely used implementations of the ISO/IEC/IEEE 42010 and is used for safety-critical applications. The advantage of the 4+1 method is also that it is easy to implement in UML, which can count on extremely broad tool support. In the remainder of this chapter we will illustrate the views of the 4+1 method with the points of attention for the link between TOPAAS and the fault tree concept. In so doing, we are trying to create greater understanding of the relationship between a fault tree, the architecture and TOPAAS TUBs.

A software architecture according to the 4+1 method contains the following views:

- **Logical view**, which describes the functional components of a system and its functionality for the end user. This view also provides the preconditions for the proper functioning of the software and its components.
- **Development view**, which describes the system from the developer's perspective. This view describes the organisation of the software during development in terms of source-code packages, standard code libraries used, compilers, links and the underlying interdependencies.
- **Process view**, which describes all active processes during execution, including all necessary mutual communication, synchronisation, redundancy and concurrency control.
- **Physical view**, which describes the hardware infrastructure (PLCs, servers and networks) and the allocation of the run-time components over these, from the system engineer's perspective.
- **Use-case view / Scenarios**, which, by means of a small set of examples (use cases) or scenarios, provides insight into the relationship between the various modules. This explicitly describes not only the primary scenario, but also the alternative scenarios that arise as a result of redundancy or monitoring with restart function (watchdog resets). As a minimum, this view should show all tasks to be performed that are important for the fault tree. The scenarios will serve as a basis for the test protocol or the prototype.

There is always a very large overlap between the views. For example:

- the process view describes how modules are interrelated;
- the physical view describes how the same modules are distributed over hardware and network connections;
- the scenarios view describes how specific modules work together to achieve task execution.

With all these views together, an image can be constructed of how the system works, similar to a hydraulic, electrical or mechanical construction diagram. These construction diagrams must therefore either be provided by suppliers on delivery or must be determined by a software analyst in retrospect on the basis of all the other documentation available.

B.1 Example of a 4+1 architecture

To make a 4+1 architecture a little more tangible, we describe one at a rather abstract level.

Logical View
The logical view looks like this:

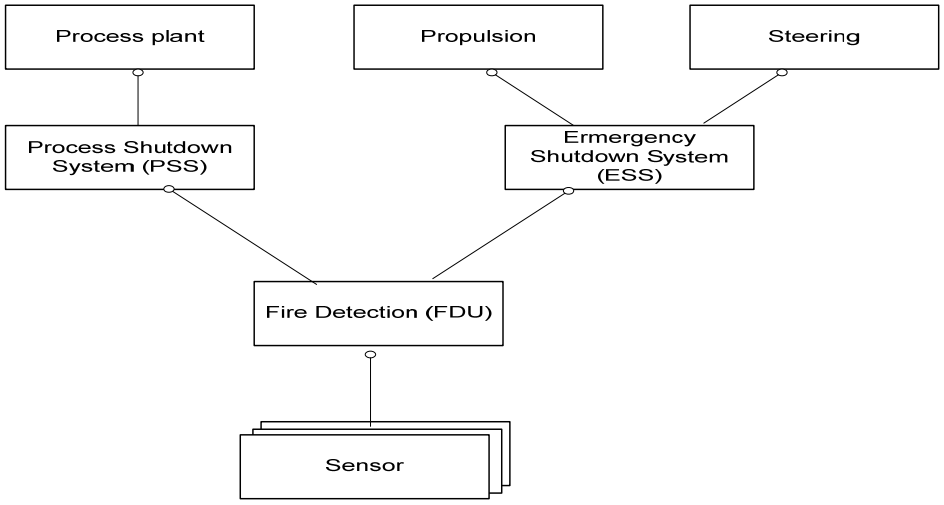


Figure 6: Logical view

This is a structure of a ship's fire-alarm system, where all mechanical processes (both primary and secondary) must be shut down if a fire is detected. It shows that the fire-alarm system (FDU) shuts down the ship's emergency stop system (ESS) and the industrial process via an emergency stop (PSS).

Process View
The process view looks like this:

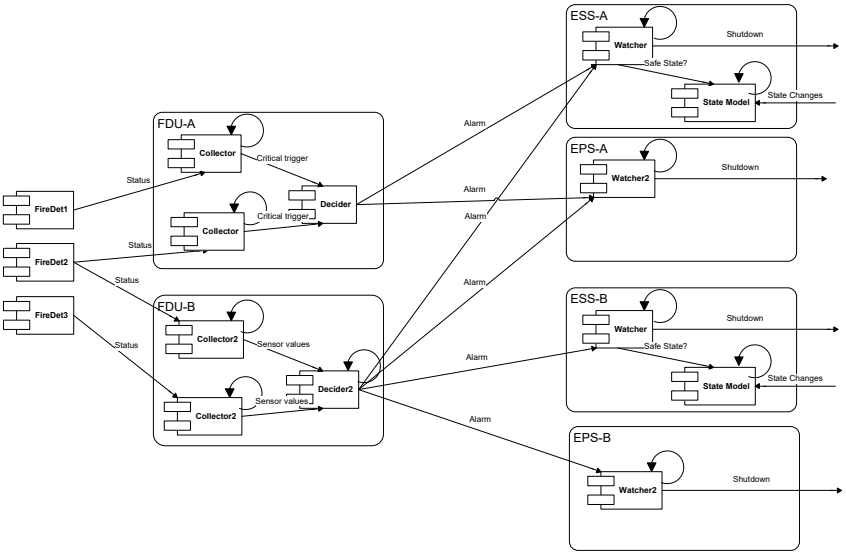


Figure 7: Process View

This view concerns the cohesion of the processes/modules identified, regardless of the physical demand of these processes on hardware, or hardware/software redundancy.

Development view
The development view is a view for software developers, and tells which libraries are used and shared:

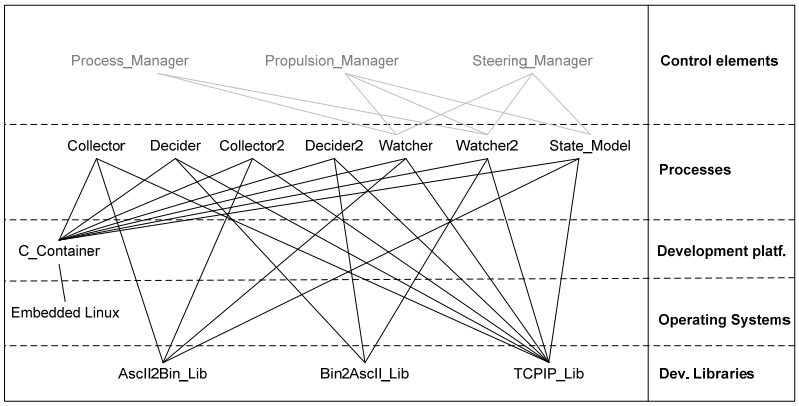


Figure 8: Development view

As the above example shows, the processes from the process view are related here to underlying libraries. One application could be to detect common mode failures due to library errors.

Physical view
The physical layout can be as follows:

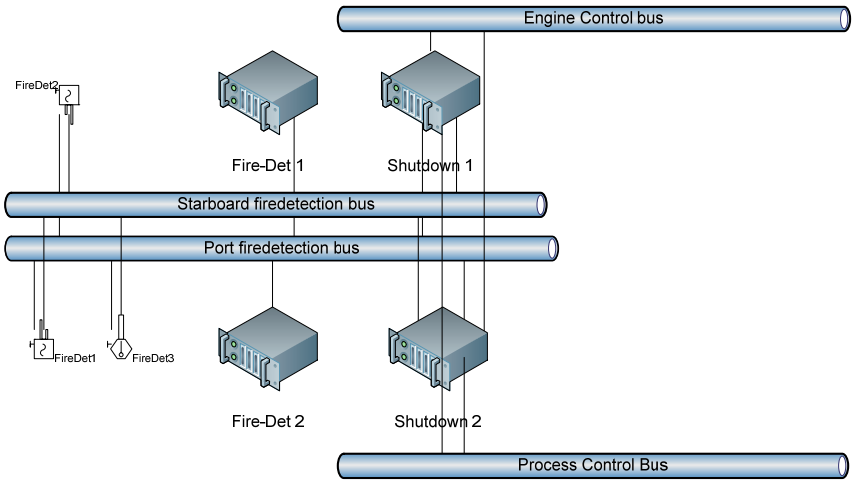


Figure 9: Physical View

Where the FDU-A and FDU-B processes are placed as redundant on the Fire-Det 1 and 2 servers, and the ESS and EPS processes are redundant on the shutdown servers.

Use-case view / Scenarios

The question now remains as to how the fire-detection scenario actually works. This is as follows:

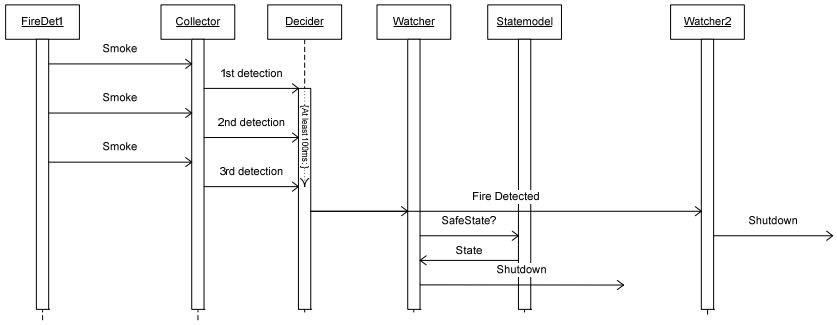


Figure 10: Use-case View

This shows how processes work together to realise functionality.

B.2 Points of attention for the use of a 4+1 model

To achieve a good usable architectural description, there are some practical points of attention, which focus mainly on the minimum level of detail used for the different views:

- Physical view, because physical components from this view also end up in the fault tree, it is wise to describe all individual hardware components at the same level of abstraction as the rest of the fault tree.
- The Process view should align with the detail level of the scenarios. This implies that each named process/module in the Scenario view must also be named in the Process view.
- Use-case view/Scenarios must be able to identify all task executions relevant to the fault tree. They must therefore match the level of detail used in the fault tree.

Bijlage C : Example (informative)

The BOS is the Decision and Support System that determines whether the Maeslant storm surge barrier should be closed. It also determines all the moments that are important in the closing procedure, for example the warnings sent to the Port of Rotterdam Authority, the opening of the dock doors, putting out to sea, etcetera. In the initial fault tree, as a basic event and failing task execution, the 'Wrongly not exceeded' object is included, which must be elaborated and quantified.

It goes too far here to treat the architecture in detail, but on – very simplified – main lines, the BOS looks like this:

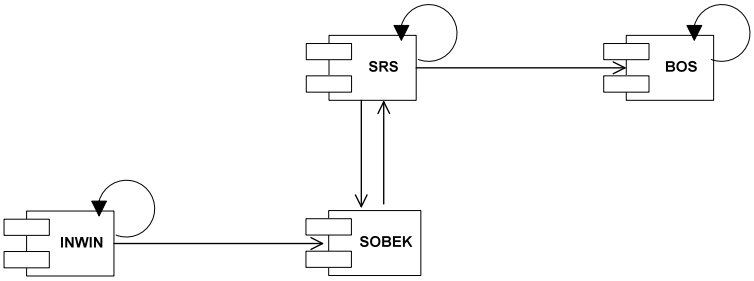


Figure 11: Process view of INWIN, SRS, SOBEK and BOS

Here INWIN delivers raw water levels from the monitoring network to SOBEK. SOBEK is periodically questioned about the expected water level at Rotterdam, after which BOS determines whether this is above the maximum water level.

In the determination in first qualitative modelling, INWIN, SOBEK, SRS and BOS are seen as part of the analysis.

Now there are two scenarios involved in determining a level exceeded:

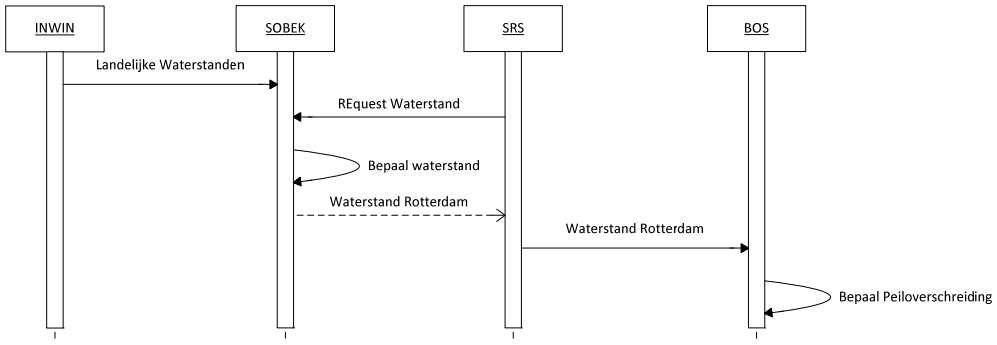


Figure 12: Regular Scenario Level Exceeded

And the alternative scenario, for 'broken' water levels, is as follows:

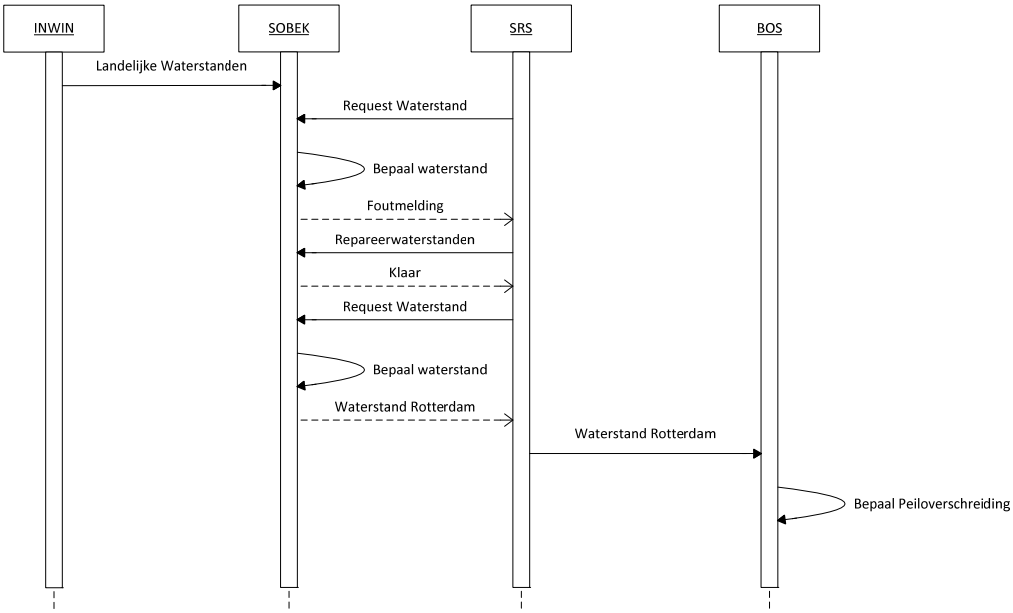


Figure 13: Alternative Scenario Level Exceeded

Now that the rough contours are clear, the specific TUBs need to be determined. The exceeded level is determined on the basis of a set of decision rules and an expected water level at Rotterdam. The expected water level is produced in the SRS module by software functions *SRS:ValideerModelInput* and *SOBEK:BepaalWaterstandRotterdam*.
Depending on the incompleteness of input data for SOBEK, the function *SRS:RepareerInput* is executed. Conditions for *SRS:RepareerInput* for this function are:

- C1: a recent (>-3 hours) expectation of the water level at Hook of Holland or a reasonably recent expectation (>-12 hours) of the water level at Hook of Holland is available.
- C2: the astronomical tide at Hook of Holland and a measured water level at Hook of Holland is available.
- In the case of C1, *SRS:RepareerInput* is not executed.
- In the case of C2 (and not C1), *SRS:RepareerInput* is executed.
- In other cases, *SRS:RepareerInput* fails completely (stops).

If one looks at the design of the architecture, we see the following TUBs:

- The tasks *SRS:ValideerModelInput*, *SOBEK:BepaalWaterstandRotterdam* and BPO logic are executed every 10 minutes, where only the execution of BPO logic depends on the task execution.
- The tasks *SRS:ValideerModelInput* and *SRS:RepareerInput* are implemented via the module SRS, running on the Stratus ftServer, making use of BOS database access. Implemented in C/C++ with associated libraries.

- The task *SOBEK:BepaalWaterstandRotterdam* is implemented via the module SOBEK, running on the Stratus, implemented in Fortran90 using Fortran libraries.
- The software function BPO logic is implemented in the BOS module PSI.

The elaborated fault tree could look like this for this part. The absence of measurement data has also been further elaborated in the current tree. That is required here as well. As an illustration, it has been included as a basic event.

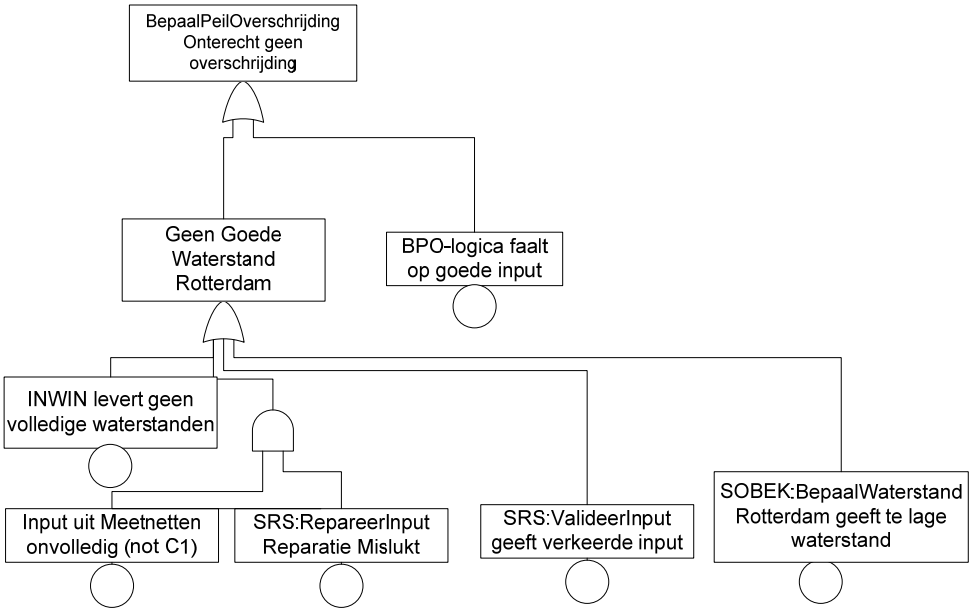


Figure 14: Fault tree BepaalPeilOverschrijding fails

In order to be able to quantify, details information about the software is required. An important analysis is the amount of source code that the task occupies and the complexity of this code. The software function *SRS:RepareerInput* is implemented by the module *SRS*. The task *SRS:ValideerInput* is only 10% of the module *SRS*.

The code for *BPO logic* is implemented in the BOS module *PSI* in methods:

- *psbepmodelevaluatie* with a cyclomatic complexity 14 and number of code statements 116 LOC);
- *psbeppeiloverschr* (complexity=10 code 162).

The module *PSI* is approximately 40 kLOC, of which specialist tasks (such as *beppeiloverschr*) are 10 kLOC. For the quantification, 30 kLOC is taken with a complexity 15 (being the maximum complexity in the general part).

The TUB INWIN, with the failing task execution 'INWIN does not give water levels' is estimated with the help of a statistical analysis; the frequency of this task execution (1 per 10 min) means that the last 10 years 144*365*10 = 50,000 runs can be compared with the actual water levels.

Looking at the *SOBEK: Bepaalwaterstanden* TUB, with the failing task execution *'SOBEK Bepaalwaterstanden gives too low water level'*, the following scoring is realised:

Development process		
1	The development process complies with one of the SILs in the IEC 2 Development process demonstrably meets SIL-1. *Built by small group of good SW engineers with 'moderate' specific process documentation within ISO9001 process.*	-½
2	Use of Inspections 2 Inspections carried out on designs and code *Reviews carried out, no formal inspections*	0
3	Volume of changes compared to original design/requirements package 3 No changes *Here the behaviour for SRS varies with respect to behavioural exceptions. The main stream is constant*	-⅓
4	Culture and collaboration of development organisation 2 Goal-oriented organisation *Organisation with a solid track record and a well-organised internal work process*	0
5	Education level and experience of developers 4 Demonstrable excellent knowledge and extensive experience of the specific domain *Built by small group of good SW engineers. Good in the subject matter, no process documentation.*	-½
6	Collaboration with Client 3 Highly involved client with sufficient knowledge, open dialogue where the client is prepared to implement changes at system level in order to avoid suboptimal behaviour. There is a systems-engineering approach for the entire development of the system. *Project organisation and its management left room for redesign, if necessary. All parties involved were capable and involved.*	-½

Product		
7	Complexity decision logic of the TUB 3 Decision logic is fairly simple (contains some very isolated exceptions), McCabe index between 10 and 30.	-⅓
8	Size of TUB (Lines of code) 2 Between 10,000 and 50,000	⅓
9	Clarity of architectural concepts used 3 A separation of tasks and responsibilities between modules has been described, which respects the principle of 'low coupling, high cohesion', but this has been passively monitored during the development process.	-⅓
10	Using a certified compiler 2 Using a compiler with which there is already lengthy experience	0

Traceability of requirements and verifiability		
11	Traceability of requirements throughout the process 2 Traceable to architecture/testing	-⅓

Testing		
12	Test techniques and coverage 4 Formal test technique(s) used with medium coverage	-⅔

Execution environment / use		
13	Multiprocess environment 2 Multiple processes on one piece of hardware	½
14	Presence of representative field data during task execution 4 Considerable representative field data present from identical or very similar applications	-2
15	Monitoring 2 Limited/brief monitoring during task execution	0
Total		-4⅔

This gives rise to an estimated probability of failure per question of $10^{-4⅔}$, rounded to 10^{-4}, which is fully in line with the estimate made by experts closely involved in the project.

Bijlage D : The Fagan inspection process (informative)

The Fagan inspection (once conceived by Michael Fagan for IBM) is a software review method in which a product (or an intermediate product) such as a design or code, is thoroughly examined with the aim of correcting faults before the project moves on to the next stage of development. The inspection is carried out after each development phase by three to six people with roles as moderator, inspector, recorder or author. The process consists of six steps:

- **Planning.** The moderator plans the inspection. He assembles the inspection team, plans the meetings and distributes the necessary material.
- **Kick-off [optional].** If necessary, the product to be inspected will be explained by the author.
- **Preparation.** The inspectors prepare individually for the inspection meeting. They study the product from their own expertise and search for faults, using the 'higher documents' and checklists. Afterwards, they fill in the faults found and the time needed on the individual findings form. The moderator collects all the individual findings forms.
- **Inspection.** The inspectors jointly search for, discuss and classify the faults in the product (or intermediate product). The moderator leads the meeting and makes sure that faults are looked for – not solutions – and that the author is not targeted. The recorder notes the faults found on a registration form.
- **Rework.** The author restores all serious faults found and keeps a record of which faults have been corrected and how much time it has taken.
- **Follow-up.** The moderator checks whether the author has done the reworking properly and whether the product meets the final criteria. In addition, the moderator makes a summary of the inspection in which, among other things, the number of participants, the time spent per step, the number of faults, the reworking time and whether the product has been accepted.

Typical for Fagan inspections is the use of start and end criteria that condition the start and conclusion of a development phase. These arise from the preconditions described in the 'higher documents', i.e. the documentation of the higher level of abstraction. For example, a detailed design is the 'higher document' of code. Practical experience shows that the inspection result strongly depends on the accuracy with which the process is carried out. That is why it is important that the people who start it are trained in performing Fagan inspections and have a minimum infrastructure at their disposal: standard inspection forms, checklists and a database to store the data.

Studies at various companies (such as AT&T, HP, NASA, IBM, ICL, BNR) have shown that Fagan inspections are 5 to 20 times more effective than testing, increasing productivity by 14 to 23 percent and making the correction of faults 10 times more efficient.