Abstract

While `safety` has been a major concern in most engineering disciplines for a long time, it has been mostly emphasized in systems where lives can be endangered. Safety in terms of business and loss of money has not been incorporated to a noticeable extent in the software industry. The reasons may be many, but most likely it is thanks to a lack of well-known, easy to use methodologies. There has been many proposals, but most of these have been intended for research. In many cases, a custom made modeling language is part of the proposal, but lacking supportive tools, the take-up has been low. In addition, the only available literature is often the article containing the proposal.

Preface

This report concludes my Masters Thesis for the degree Master of Science in Technology, Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The problem was chosen based on my interest for Software Engineering principles, using an idea from professor Tor Stålhane.

Working on this thesis have given me insight into developing `business-critical` software, and hopefully this report will make developing such systems more structured and successful.

The thesis has been written under supervision of Professor Tor Stålhane at IDI.

The report has been written using the MiKTeX version of LaTeX 2_ε.

Conventions

Though we have tried to make this report as simple and understandable to read as we could, we find that some small hints may prove useful.

Reading the report from start till end will, hopefully, provide all information and knowledge necessary to grasp our ideas. Still, we have tried to include references to more specific information as much as feasible. This references looks like this: [7], and the corresponding literature is found in the bibliography. In some cases, we refer to information found elsewhere in the report. For the reader to easily find this information, we use a reference to the chapter and section number. For instance, the reference to the introductory chapter will look like this: 1.

We mostly use this regular font, sometimes **bold** or *italic* to mark out things we think should be noticed. In fact, the only time our font is really meaning something is when it is `teletyped`. A word written in teletype is marking out a word that can be found in our list of definitions in appendix B. These definitions are included to make the text more precise, and to make sure that the readers will understand what we mean with, e.g., `safety`

The first chapters of this report is meant to be an introduction to terms and techniques that we think is necessary to know before we present our proposal of how `business-safe` systems should be developed. In these chapters, our contribution mostly consists of collecting and organizing the information. Instead of using citations all through these chapters we are giving the references in the beginning of the sections. These chapters are chapters 2-6 and chapter 8. Using the knowledge of these chapters, we develop our proposal in chapters 7, 9 and 10.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem 2

1.2.1 Problem text 2

1.2.2 Problem description 2

1.3 Outline 4

2 BUCS **6**

3 Rational Unified Process **7**

3.1 Phases and milestone 7

3.1.1 The Inception Phase 9

3.1.2 The Elaboration Phase 10

3.1.3 The Construction Phase 10

3.1.4 The Transition Phase 11

4 Hazard Analysis Techniques **13**

4.1 Preliminary Hazard Analysis 14

4.2 Failure Modes, Effects, and Criticality Analysis 15

4.3 Hazard and Operability Analysis 18

4.4 Fault Tree Analysis 20

5 Documentation strategies **22**
5.1 Intent Specification 22
5.1.1 Writing Intent Specification 23
5.2 ASCE 27
5.2.1 Claims-Arguments-Evidence 28
5.2.2 Preliminary Safety Case element 31
5.2.3 Architectural Safety Case element 32
5.2.4 Implementation Safety Case element 36
5.2.5 Operation and Installation Safety Case element 37

6 Tools **39**
6.1 ASCEd 39
6.2 SpecTRM 40

7 Today's practice **41**
7.1 Requirements Engineering 42
7.2 Design 42
7.3 Implementation 43
7.4 Testing 43

8 RUP in BUsiness-Critical Software **44**
8.1 General 44
8.2 The Inception Phase 45
8.3 The Elaboration Phase 47
8.4 The Construction phase 49
8.5 The Transition phase 49

9 Barrier Conservation **51**

10 Conclusion **54**
10.1 Choice of tools 54
10.2 Using ASCE for barrier conservation 55
10.3 Further Work 56

A Demonstration of concept 58

A.1 Conceptual description 58

A.2 Inception phase 58

A.3 Architectural phase 61

A.4 Construction phase 71

A.5 Transition phase 73

B Definitions 75

List of Tables

4.1 Excerpts of a PHA table for a flight controller 15
4.2 Part of the HAZOP table for a database connection client . . 20

A.1 The PHA Table 59
A.2 The system test plan 62
A.3 The discount use-case 64
A.4 The HAZOP table 66
A.5 Testplan for Claim 3 68

List of Figures

1.1 The development process and tracing 3

3.1 The overall architecture of RUP 8

4.1 The FMECA table of appendix A 17

5.1 The Intent Specification document structure 25
5.2 The correspondence of RUP and Safety Case 29
5.3 The attribute table 33

A.1 Adding the first claim to the ASCE network 60
A.2 Our second claim, linked to external documents and the first claim 65
A.3 The subclaim connected to the previous claims 67
A.4 The arguments connected to the appropriate claim 69
A.5 The FMECA table 70
A.6 We have completed one fork of the ASCE network 72
A.7 The complete demonstration network 74

Chapter 1

Introduction

As computers find their ways into new environments, so do software. And like all other tools, computers and software must be customized to their environment to be utilized in the best way possible. What is often forgotten is that it is not only performance that decides how useful a tool is, but also how prone it is to errors. The largest and most powerful caterpillar in the world is of little use if it spends six months a year being repaired. Still, this is a small problem compared to an unstable caterpillar failing often. Imagine a caterpillar where the controls suddenly gets stuck, or start acting on their own.

But not all `failures` will lead to dangerous situations. 'Normal' software does not control big machines or poisonous fluid processes. Most software controls servers and personal computers that, contrary to popular myths in the early 90's, cannot be turned into a bomb by malicious programs. `Accidents` in these systems does not kill people, but in a worst case scenario they may kill a company. Software `safety` has so far mostly been implemented with physical safety in mind, while this report will look at how a business can avoid to loose money when a failure occurs.

1.1 Motivation

Working through a number of software engineering exercises and projects has shown me that this is a field where ad-hoc and temporary solutions are the rule more than the exceptions. This is especially true for new fields in software engineering, where old methods are used without much thought, if any methods are used at all. Whatever project I was taking on for my

Masters Thesis, I wanted to break new ground, and the description of this project promised a lot of ground-breaking work.

1.2 Problem

In this section we will present our problems and the goals of this report.

1.2.1 Problem text

BUCS Implementing safety - A proposal as to how to implement safety concerns

When using hazard analysis methods on requirements, one is aiming to come up with `barriers` for avoiding unwanted situations. The next step is to refine the requirements by including the results of the first hazard analysis, and also make sure that the test plan will cover the barriers. We then have to perform a new hazard analysis to see if the refined requirements have opened up for new `hazards`. When doing these iterations of hazard analysis, barrier development, and requirements and test refining, it is important than one do not loose the barriers nor the analysis results from earlier iterations. Our goal in this report is to investigate how to keep and test these barriers in order to make the systems more stable and safe.

1.2.2 Problem description

One of the goals of the BUCS project is to contribute *"A method for testing that the customers' `business-safety` concerns are adequately taken care of in the implementation"*. To reach this goal, we need some way to define, trace, and test these concerns throughout the development process. A graphical presentation of this process is shown in figure 1.1.

The dotted lines symbolize the documents and information flow through the process. The continuous lines symbolize `intentions`. As one can see from the figure, we start with the costumer's requirements. These requirements are transformed into implementation requirements, and are at the same time used to perform the first hazard analysis. As the implementation requirements are updated to include the results of this hazard analysis, it will be necessary to perform new hazard analyzes to detect if the updating has lead to new `hazard`. The hazard analyzes may also discover the need

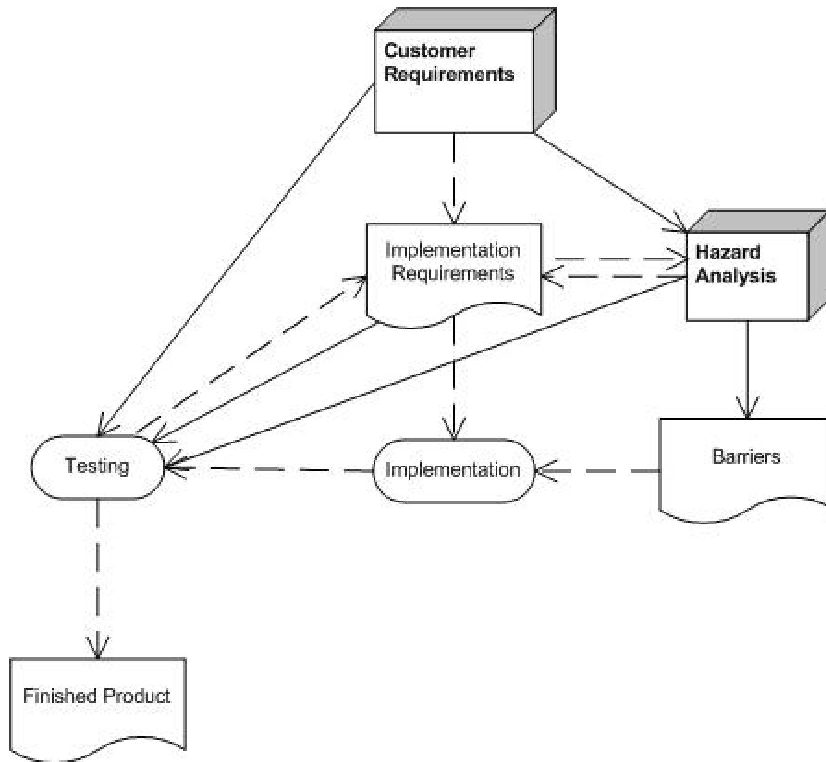

Figure 1.1: The development process and tracing

for barriers. Information about the barriers are also used in the implementation, and should also be included in the test plans. The implementation requirements are used for the implementation of the system. Testing the implementation may lead to changes in the implementation requirements, or, if all requirements are fulfilled, to a finished product.

The hazard analysis needs to know the intentions of the customer, in addition to the requirements, to discover possible hazards. When testing, both customer requirements and hazard analyzes must be considered in addition to the implementation requirements. In this way, the intentions and reasons for the implementation requirement can be taken into account when evaluating the test results. If the test results suggests that changes should be made to the implementation, knowing the intentions and reasons for the original implementation makes it easier to make the best decisions.

This report aims to analyze the problem shown in figure 1.1, and make a proposal as to how this can be solved. We will not develop new methods, but use familiar and well established techniques in new settings. This way we hope that just a minimum of training will be necessary to maximize the

effects.

1.3 Outline

The report is structured as follows:

1. Introduction. This chapter - presents the problem along with the BUCS project and the report.

2. BUCS. A short presentation of the BUCS project.

3. Rational Unified Process. We give a short introduction to the RUP development methodology.

4. Hazard Analysis Techniques and

5. Documentation Strategies. In these two chapters we will make a short presentation of the methods that are candidates for use in the report.

6. Tools. This chapter gives a brief overview of tools that may be used to support a better way of developing `business-safe` systems.

7. Today's Practice. There is missing practices for developing `Business-critical` software. We will look at how things are done today, and also briefly discuss what is missing.

8. RUP in business-critical Software. This chapter is discussing how RUP can be altered and expanded to be more suitable for developing business-critical software.

9. Barrier Conservation. This chapter is specifically discussing how our candidate methods cope with barriers.

10. Conclusions and further work.

11. Appendix A, Demonstration of concept. This appendix walks through the development process, showing how we propose to develop business-safe software.

12. Appendix B, definitions. In this part we will collect the definitions used throughout this report to make them easy to look up.

In the first seven chapters we look at today's situation, along with a number of techniques and tools that we believe may be helpful for developing business-safe systems. What we have learned during these chapters are then discussed in chapter eight, looking at the pros and cons of the different techniques and their combination. As BUCS has already decided to use RUP [3], the discussion is mainly about what documentation strategies that are best combined with RUP. The last chapters are based on our discussion, aiming to take a closer look at the consequences of our decision. Also covered in these final chapters is a brief demonstration of how our proposal may be used in a real development project.

The most important chapters are chapters 5 to 10, assuming that most readers do not have much knowledge of Safety Case and Intent Specification. In the opposite case, chapter 5 and 6 may be omitted. Readers that are not familiar with RUP should include chapter 3, while chapter 4 provides the necessary knowledge of hazard analysis for using this report. We do not recommend our readers to look at Appendix A without having read chapters 8 and 9, as the example does not contain all of the information used in the discussions.

Chapter 2

BUCS

BUsiness-Critical Software (BUCS) [3] is a research project founded by the Norwegian Research Council (NFR) and conducted by IDI at NTNU. The project is aiming to develop methods that can be used in development of `business-critical` systems. These are systems intended for use in settings where `failures` may lead to financial losses rather than physical losses. BUCS are initially not concerned with the time and costs of development projects, and the goal is not to help developers complete their projects on budget. The BUCS project formulates their goal as *"Help developers, users, and customers to develop software that is safer to use"*. Safety in this goal is `Business-Safety`.

BUCS wants to tailor existing methods to handle business-safety concerns in developing, operating, supporting, and maintaining systems. The resulting methodology will follow the RUP (3.1) process management model. To accomplish this, BUCS is focusing on `hazard` prevention rather than hazard detection and reduction.

Along with the BUCS own website, information about both the BUCS project and the work conducted on business-safety can be found on [14; 6; 15].

Chapter 3

Rational Unified Process

The Rational Unified Process® (RUP®) is a framework for software development developed by Rational Software, today a division of IBM. RUP is based on 'best practice' from leaders in international industries. RUP is solely concerned about *project* `risks` and `hazards`, aiming to reduce project risks and extend the control of the process by discovering and attacking project hazard at an early stage of development. The hazards are then continuously attacked throughout the product's lifecycle. The same goes for changes to the product which should be accommodated as soon as possible after being deemed necessary. RUP is organized using four phases, each ending with a milestone and a review of the results from this phase. Each full pass through the four phases is called a development cycle, and produces a generation of the software.

Another feature of RUP is its orientation towards constructing code that is reusable in later iterations. When designing, RUP puts much effort in use-cases as the main tool. Finally, RUP is designed to be iterative, aiming to develop a stable (baseline) architecture as early as possible. More details about RUP can be found in [20; 18; 9].

3.1 Phases and milestone

As mentioned above, the RUP consists of four phases. These are:

- The inception phase
- The elaboration phase

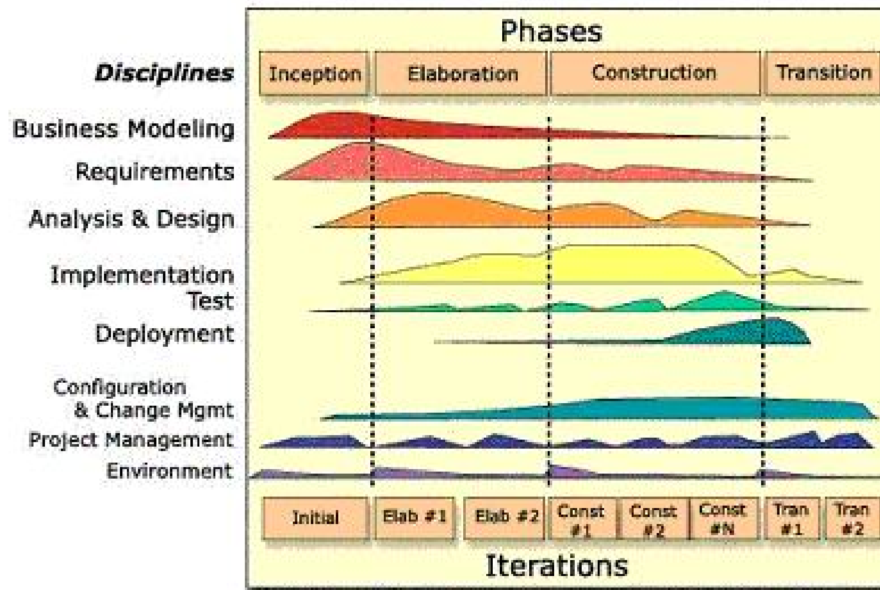

Figure 3.1: The overall architecture of RUP

- The construction phase
- The transition phase

Each of the phases are concerned with one aspect of the software development process; thus aiming to make sure all dimensions of the finished product has been thoroughly examined. The actual development takes place in the last three phases, whereas the first phase deals with project risks, budget, human resources, and other necessities for running the project. The amount of time and resources assigned will be different for different phases, as it depends upon the workload of the phase. The general architecture of RUP is shown in figure 3.1.

Time is represented on the horizontal axis, showing the lifecycle of the process. The disciplines on the vertical axis are grouping the activities logically by nature. The graphs illustrate how focus changes between the disciplines as the project proceeds.

The figure also shows that allthough the names of the four phases may suggest different, each phase makes an iteration of analysis, design, implementation, and testing. This is according to the best-practice principle that were used when developing RUP as most products are developed in several small iterations. The focus of each phase is different, e.g. the load of business

modeling is higher in the Inception phase than in the Construction phase, while the opposite is true for implementation.

In the following sections we will look at each of the phases and the corresponding milestones.

3.1.1 The Inception Phase

This phase is focused towards the business aspects of a software development project. The developers decide upon what to build and the key functionality of the product. The goal is to evaluate if the project should be run at all and, if so, whether it meets the business requirements of the stakeholders. The main document is a Business Case that includes business context, success factors and financial forecasts. The phase also produces plan documents and a basic use case model. It should also address business and requirements risks of the project.

To pass the milestone of the inception phase, the following has to be satisfied:

- Stakeholder concurrence on scope definition and cost/schedule estimates.
- Requirements understanding as evidenced by the fidelity of the primary use cases.
- Credibility of the cost/schedule estimates, priorities, risks, and development process.

The stakeholders should also look at the actual expenditures versus planned expenditures, to find if the project costs are acceptable and if the budgets needs to be updated.

Failing to meet the criteria of this milestone will often cause the project to be canceled.

On later iterations, the Inception phase will be shortened. It will still be used to decide whether a new generation of the product should be developed, but using the knowledge from previous iterations should make this decision easier than on the first run.

3.1.2 The Elaboration Phase

This phase may be compared to the design phase of the waterfall model. The main effort concentrates on designing a stable architecture and investigates critical items in the design. This includes planning for the rest of the project and handling of the project hazards. In many cases, an evolutionary prototype along with exploratory, throw-away prototypes will provide much needed information.

The most important documents evolving from this phase are:

- A use-case model in which the use-cases and the actors have been identified and all significant use-case descriptions are developed.
- A description of the software architecture in a software system development process.
- Architecture prototype, which can be executed.
- Revisited Business case and risk list.
- A development plan for the overall project.

Both during the phase, and especially upon failing to meet these criteria, refining and validation of architecture and vision is important if the project is to continue. The completion of the milestone of this phase is usually the last chance for cancelation before the project turns into a high cost, high risk operation.

3.1.3 The Construction Phase

As the name suggests, this is the phase were the system is implemented. In this phase, the requirements enter their final version and are implemented. The software may still be developed in a series of iterations to achieve working versions as soon as possible. This way, testing may begin even though the system is not fully implemented.

This is also the biggest phase, both in man-hours and costs. This means that managing and planning in this phase is important for the outcome of the project.

At the end of the construction phase, a first version of the product can be released. The phase will in most cases not end until this version is ready,

or the project is canceled if implementation turns out to be infeasible. In rare instances, the phase may end without a product and a new iteration is started. This can happen if there is evidence that the system concept is sound, but the design has flaws.

3.1.4 The Transition Phase

This phase is concerned with the delivery of the system to the consumers. Although the development may seem finished, there is still a lot of work to do. Large, custom-made systems may require a lot of installing and fine-tuning. Often, training of the end-users is also needed. For all kinds of systems, both tailor made and COTS, patching and debugging together with other kinds of maintenance is necessary.

For some projects, this is the first phase were extensive user feedback is available. This means that new issues may be found. If the system is not going to be released in a new generation as a result of a new iteration, the issues will have to be solved "on the fly". So even if new generations of a system are not planned, work on the system cannot be abandoned even if it has been delivered to the customer. It is necessary that a plan and an infrastructure for maintenance of the system is developed, and that resources are made available to make the necessary changes. This plan should be developed by the developers, customers and users in cooperation. This cooperation is important so that the maintenance and support is fitting the customers and users organization(s), while providing enough resources to keep the system up to date and running. There may also be need for a support team to help the users, at least until enough knowledge has been transferred to the customer for him to run his own support team. The resources needed for the maintenance team is smaller than what the developer team demands. In many cases, it is possible to have one team maintaining several products. The best support and maintenance plans are a results of cooperative work between the developers, support team, and the customer and users.

As the phase is ending and the product is delivered, an evaluation of the project and product should be performed. The main concerns of this evaluation are:

- Product generation. Should a new development cycle be started to improve the product?
- Customer and user satisfaction. Did the product satisfy the customers? Does the product fulfill the users needs?

- Project satisfaction. Did the project go according to the plan? Are the expenditures acceptable? Is there anything to be learned from this project, good or bad, that can be utilized in later projects?

Whatever conclusions are reached, the phase results should be stored as experiences to be used for later projects and products.

Chapter 4

Hazard Analysis Techniques

This chapter is an introduction to some of the most popular techniques and methods for hazard analysis. Different methods are fit for different settings, according to how much information that is available, what kind of system one is dealing with, what sort of `hazard` one is looking for, and so on. T Each technique will have a brief explanation of its function and target, prospective dangers and limitations, and which results to expect. We also suggest at what stage the method is usually applied, but when to perform an analysis is dependent upon several factors, including the manner of which the system is developed. This section is not meant to be a tutorial of how to perform the analyzes, as such are found in lots of literature together with the necessary details.

Even if some of the methods will cover the same areas and some hazards are likely to be discovered in every analysis, it can be dangerous to skip or haste through any analyzes or parts of them. Even if the hazard is the same, different views may bring forth new aspects. There is also an important point that some of the techniques are only looking at the hazard itself, while others include causes, consequences or both. In any cases there are two important factors that must be in place:

- Information about the system's environment.
- The stakeholders.

Without these pieces of information, the analysis will not be of any use.

To be able to handle hazards, it is also important to have knowledge of other aspects of the system. How much effort that should be put into eliminating a hazard depends upon how serious the consequences of the resulting

`accident` will be, but the probability of the hazard taking place must also be taken into consideration. Using risk tables help deciding which hazards are the most important to attack, as the `risk` is calculated from several factors influencing the graveness of hazards. Also, when introducing barriers, it is necessary to calculate the probability of the barriers not working. The last piece of every analysis is perhaps the most crucial to make sure that efforts are not wasted. Every solution must be connected to a test to make sure that it will be evaluated in the final system.

4.1 Preliminary Hazard Analysis

The Preliminary Hazard Analysis (PHA) [13] is normally the first analysis one will perform. It is simple, and does not require much information except for the concept of the system and its intended users. Usable for almost any kind of projects, the PHA aims to give the developer a head start on the `hazard` by forcing them to think about anything and all that may go wrong. As the PHA is usually conducted before the design starts (since it is used to help in making design decisions), most of the hazards discovered are also conceptual. They may, however, also be used to discover design solutions that can be dangerous.

How do you do PHA

The PHA is performed by the developers and domain experts in an informal way. A simple table which contains four fields is used : Hazard, Cause, Main effect, and Preventive action (table 4.1. Additionally, one may use a field for characterizing the graveness of the `failure`. The analysis itself is a brainstorming session, where possible hazards are stated as they are discovered. Rather than structuring the session too much, and thereby possibly forgetting an idea when the right time comes, one should plan on spending some time after the session to organize the results. It is important to remember that at this stage, the goal is to document all possible hazards that the analyst may come up with. In this way we get more information so that the design may be as good as possible from the start. One should be careful with dismissing any proposed hazards. If one of the analyzers is able to come up with a hazard, it means that this is a possible state for the system that should be avoided. If the hazard is not recorded, chances increase that this state may be present in the system design because the designer is not aware of the hazard.

Hazard	Cause	Main Effect	Preventive Action
...	...	...	...
No altitude measurement	Broken antenna	Pilots receive no altitude readings	Backup antenna
Altitude miscalculation	Weather conditions	Pilots receive erroneous altitude readings	Different technology backup system
...	...	...	...

Table 4.1: Excerpts of a PHA table for a flight controller

Results

As this analysis is performed very early in the project, the hazards will usually be quite general and based upon the participants experience. Most of the hazards will be stated as "could it be that"s, but they should be no easier dismissed than a fully stated hazard. They are all equally important when design decisions are made. In some cases, a vague hazard may even be more dangerous if its consequences are dire. The same way, a detailed failure leading to a minor lag in the execution should not lead to any major changes. Unfortunately, software developers tend to put more faith in anything concrete than loosely defined ideas.

As we are still early in the process, it is important to keep records of all hazards, even the ones that are omitted or dismissed in the first design. Later changes may lead the system back on a track that has been left earlier. A removed or deleted hazard may then suddenly be brought back into play.

4.2 Failure Modes, Effects, and Criticality Analysis

The Failure Modes, Effects, and Criticality Analysis (FMECA) [13; 16] is an old method for analyzing technical systems. The FMECA is an expansion of the Failure Modes and Effects Analysis (FMEA), adding a description or ranking of the `failure` modes. As the difference is often somewhat blurry, we will use FMECA to name both methods. When doing a FMECA analysis, one should know much more about the system than what is needed for the PHA. The analysis should take place during the design phase of the system.

Thanks to more information, the method is able to dig deeper into the system, and therefore providing more specific solutions.

In addition to finding and categorizing `hazards` and failure modes the FMECA tries to find the cause(s) behind a `Failure` and the consequences for the entire system. This expansion of the analysis means that there is also need for a more formal approach, in this case realized by a more detailed table.

How do you do FMECA

Before starting the analysis is it important to decide on the conditions for the system. What parts of the system is included in the analysis, which modes of operation will be taken into account, and what are the failure modes. What modes that are failure modes depends on the nature of the system, and it is not always possible to transfer the failure modes of one system to another.

The fields of the FMECA table must also be decided. Depending on which stage of development the project is in, and what goal is set for the analysis, different fields can be needed. In [16] it is suggested to run FMECA twice during a project. They suggest the first run to take place during the definition of the detailed requirements, and the second run should take place between the design and implementation. Table 4.1 shows an excerpt of an FMECA table for our demonstration system in appendix A.

If the system is too large or complex for a single analysis it must also be decided how to split up the system. After dividing the system into analyzable parts one should make a diagram to show the connections and dependencies between the parts. Finally, the components of each part are listed to make sure they are analyzed.

There are several ways to divide the system into components, mostly depending upon what kind of hazards and failures one is looking for. A common strategy is to look at the design and source code components - modules, classes, methods and so forth. Deciding upon how to divide the system into components is closely related to finding the failure modes of the system, as dividing a system into components that can not have any failure modes makes the analysis worthless.

After the components are listed, there is no formal procedures as to how they are analyzed. Like the PHA, the analysis is a brainstorming session, but this time it is structured by the table. For each component, the participants will ask "what can go wrong with this component?". The answers are then included in the FMECA table.

System: MTD Conducted by: Ole-Johan
Ref. drawing: Date: 27.09 Page: 1 of 1

Unit description			**Hazard description**			**Failure effect**	
Ref.nr.	*Function*	*Operational state*	*Failure mode*	*Failure cause*	*Failure detection*	*On other units*	*On main unit*
01	Order registration	Accepting orders	Fail to accept order	Erroneous order		None	None
02	Order registration	Registering orders	Fail to register order	No database connection	Catching exception	None	System halt
03	Discount registration	Registering discount and recalculating sum	Registers wrong discount	Algorithm fail	Operator	None	None

Ref. nr	Failure rate	Failure effect rank	Failure reduction	Comment
01	10%	Negligible	Order registration routines	
02	2%	Critical	Connection control procedures	Can stem from multiple synchronous fails
03	1%	Critical	Testing	Discovery should lead to code review

Figure 4.1: The FMECA table of appendix A

Ideas for other components may be noted by the participant, but should not be brought into the discussion till the actual component is in question. As mentioned for PHA, this may lead to some proposals being forgotten, but the size of the analysis forces this strategy to maintain the overview. On the other hand, the great increase of information will improve the quality of proposed hazard. The participants knowledge of the system also often means that they have already discovered possible hazard, and they have had time to think about the analysis while working on the system.

Results

A FMECA produces a lot of data. Every component has been analyzed, and every `Hazard` that the analyst team is able to imagine is documented. The construction of the FMECA table also helps getting all of the details recorded.

The massive amount of data is one of the FMECA's drawbacks. A lot of small and possibly negligible failure take up space in the table. This can make it difficult to maintain an overview of the analysis. Another problem is that FMECA is not taking into consideration chains of problems, though it evaluates the failure consequences on the entire system. Lastly, FMECA is not considering human errors. One way to cope with the amount of data is to organize the entries depending on the consequences of the hazard. The internal ordering will then build on how easy the hazard may be removed. This way, the designers may start right on getting rid of the most threatening hazard, while still having some time to think on hazard that are not so easily resolved. Irrespective of how the developer team organizes the data, every record should be kept for future use. hazard that are taken care of must still be kept for use during the testing of the systems. If resources are too small to handle all the hazard found at this time, they may be found in a later iteration or during maintenance.

4.3 Hazard and Operability Analysis

The Hazard and Operability Analysis (HAZOP)[7; 13] is a major analysis, attempting to guide the analyst team through the system while providing help to discover its `hazards`. Even if the HAZOP in some ways will recreate the work done in FMECA, the results will be a complement to the FMECA results. A HAZOP is usually conducted after the system design is more or less complete, as the need for complete and correct documentation is crucial

for the quality of the analysis. The need for a design is also based upon the HAZOP aim, namely to identify 'deviation from design intent' [7]. A HAZOP should always be done by a team of domain experts, which should be picked based on both knowledge and cooperating skills. It is crucial that the study leader is experienced in HAZOP. Studies have shown that HAZOP analyzes performed with inexperienced leaders tend to fail.

How do you do HAZOP

Strictly following the guidelines are absolute necessary to get the full benefits of a HAZOP analysis. While FMECA may be conducted by one person, the HAZOP is a team effort, and both the leader and the composition of the team is important for the results. The HAZOP is performed in a structured and formal way, using "Study Nodes" and "Guide-words", thus helping the analysts to cover every aspect of the system. As for PHA and FMECA, the analysis should be documented by using a predefined table. For HAZOP, this will contain the hazard as discovered when using the guide-words, consequences, reasons and suggested solutions. An example is shown in table 4.2. During a study, the guide-words often have to be interpreted to fit the situation, e.g. "more" is interpreted as "greater". It is important that these interpretations are documented, or the `Hazard` may not be resolved correctly. One of the proposed techniques for keeping track of the interpretations is a matrix with the generic guide-words on the top row, and the components to be analyzed in the first column. It is an ongoing discussion whether it is better to use new, customized guide-words for software systems, or stick to the originals. Supporters of the former argue that the guide-words were developed for process industries, and are thus not useful for software. Those who supports the latter usually claims that the guide-words, though not developed specifically for software, are general enough when interpreted in a sound manner. There are also some studies which results suggests that the variety of software systems is so great that the set of guide-words still will have to be interpreted, even if they are customized.

Results

The records of the HAZOP study should show all the hazard found. It is also important to include all questions that have risen, whether it is a request for more information, or if it is a possible hazard depending on design decisions. The last part of the record is recommendations for avoiding the hazard. As resolving problems is not the main aim of a HAZOP, the recommendations

Guide-word	Hazard	Consequences	Reason	Suggested solution
No	No connection	No data will be fetched	Erroneous address	Address syntax verification
Too much	Too many bits in result	Erroneous calculation	Signal distortion	Parity bits
...	...	...	...	...

Table 4.2: Part of the HAZOP table for a database connection client

should be focused on advising how to avoid the hazard, and not presenting specific changes to the design. The task of adjusting the design to cope with the hazard is the responsibility of the designers.

4.4 Fault Tree Analysis

Though, strictly speaking, The Fault Tree Analysis (FTA) [13] is not a hazard analysis, but a technique for `reliability` analysis we include an introduction in this chapter. The reasons for performing an FTA in software engineering is much the same as for doing hazard analyzes, and knowing more about the causes and events connected to a `hazard` can provide valuable information when deciding how to the hazard.

FTA is widely used in the space industry and on nuclear plants. It builds a tree that logically connects unwanted events and their causes. The FTA can cover most kinds of causes, including human errors and environment factors. Depending on the reason for conducting a FTA, the results can be expressed in different forms. As the FTA starts with the unwanted event, it will be natural to do this after e.g. a HAZOP, to further investigate the `Hazards` discovered.

In [1], System Hazard Analysis, Nancy Leveson is proposing using quantitative FTAs for refining system design constraints, and also for verifying code.

How do you do FTA

The most important part of FTA is to define the top-event, which in our case may be a hazard. An incorrect or shady definition will make the analysis

hard, and may leave it invaluable. As a rule of thumb, the definition of the event should answer what, where, and when. One also has to define the system's borders, including, but not constrained to, the physical borders, initial settings, external events, and the level of detail.

When the definitions are in place, is it time to build the fault tree. This is done from the event and down, by asking for the causes for this event. The rules for building the tree is quite simple, but makes it easy to keep control of the tree in addition to helping the tree becoming consistent and complete. A finished fault tree will give two important answers; the "cut-set", which are the events that by happening at the same time will lead to the top-event, and the "path-set", which are the events that by not happening at the same time will ensure that the top-event does not happen. For small trees finding these sets is done manually, but for larger trees there are tools available as the calculations quickly grows to large to handle by hand. The most popular method is the MOCUS algorithm[11]

Lastly, one can analyze the tree, both qualitative and quantitative. The qualitative analysis is ordering the events after the types of `failure` that can occur, while the quantitative is used to find the probabilities of the top-hazard.

Results

A FTA will provide a picture of the combinations of events that may lead to hazard. It is simple to understand and much used. Creating a correct and complete FTA will make the analyst a virtual system expert, which can be of great advantage later in the development process. Working with the fault tree may also lead to the discovery of errors and problems even before the analysis begins.

The FTA's drawback is that it is static and does not cope well with dynamic systems. Dynamic systems will often result in incomplete fault trees, which are mostly a waste of resources. Periodical events like testing and maintenance are also handled badly by fault trees.

Chapter 5

Documentation strategies

As we have pointed out some times already, documentation is crucial when developing safe software. We wanted to look at some strategies for organizing this work, as the amount of information grows throughout the development process.

We have chosen two methods for documenting and organizing the safety work to be further evaluated. These are Intent Specification from Safeware [5] and ASCE from Adelard [1]. Our reasons for picking these two methods included availability of evaluation copies, and that the BUCS project have earlier looked at these methods, giving us access to people who knew the methods. After a short prestudy of the methods, we found that their different approaches to the documentation task would make us able to discuss different ways of implementing `business-safe` systems.

While RUP is focusing on the project `hazards`, the Intent Specification and ASCE can be used for both project and product hazards. We find this property to be an advantage as it makes us able to keep all hazards together, though we are still going to analyze the product and the project separately and keep the records of the analysis's in different documents.

5.1 Intent Specification

The Intent Specification is developed by Safeware [5]. They present it as a tool for writing specifications that covers the entire lifecycle of the software development process. The structure of the Intent Specification is developed using theory from several fields, including software engineering and cognitive psychology. The underlying vision is that *'Specifications should help solve*

problems and support the basic system engineering process'. In opposition to common specifications that are based upon 'what' components do, and 'how' it is done, Intent Specifications build their hierarchies based on 'why'. Each level in the abstraction is linked to levels above and below, providing traceability all the way from requirements to implementation. There are five levels altogether, supporting different views of the system, and having different levels of detail.

The following sections presents the main ideas of the Intent Specification. More information, including a large example implementation can be found at the Safeware homesite [5]. Most of what is used in sections stems from Nancy G. Levesons paper [10].

5.1.1 Writing Intent Specification

The Intent Specification is designed for up-front planning and inclusion of system-essential properties. Much emphasis is laid on recording intents throughout the development process. The author believes that many errors in a software system happens because intent has been omitted or guessed upon. The Intent Specification methodology is also focusing on recording other kinds of implicit information, as research points out that even common knowledge tend to be ignored or overlooked if not explicitly stated in the specification.

The Intent Specification differs from regular specifications in using `Means-Ends Hierarchies`. While regular hierarchies have a 'what' level over a 'how' level, the Means-Ends extend the hierarchy with a 'why' level above the 'what'. In this way, tracing the hierarchy top-down allows for browsing in reasons, while bottom-up looks at the causes. The importance of the top-down ability is grounded in cognitive psychology, which shows that most expert solutions starts with reasoning about high-level functional structures and then moving down the hierarchy.

An Intent Specification document is organized as showed in figure 5.1. It is structured as a matrix, covering each aspect of the specification in different several levels. The horizontal axis decompositions the system along the part-whole dimension, having in the following fields:

1. Characteristics of the system environment.

2. Human operators and users.

3. The system "itself". The fourth column is the internal system components.

The vertical axis of the figure is the intent dimension. These levels provide the intents for the level below, preparing the ground for traceability and reasoning about the system. The five levels are:

1. System level; goals, constraints, priorities, and trade-offs.
2. Physical principles and laws.
3. Logical design, interaction between components.
4. Design.
5. Implementation.

Requirements for each level are also important parts, and level 2-5 includes validation and verification (V&V) requirements. There is no V&V for level 1, as research has not yet provided any clear answers as to what validation would be appropriate at this level.

Every level will also have to include assumptions. These are used to explain decisions made, which makes it clear to the developers why a particular design are chosen. If an assumption can be showed to be wrong, it should be updated, and the system may be analyzed to change the design according to the new information. The assumptions also provide the designers with fundamental information about the system. Even if an assumption does not lead to a decision, it will provide information that may be important at a later stage of the development cycle.

In the following sections, each level of the intent dimension will be more closely evaluated to give the readers a better understanding of the method before a discussion of how the BUCS goal can be reached.

Level 1: System Purpose

This section defines the goals of the system, and translates them into requirements. The goals provides the intent behind the requirements, and can be used for validating design choices and test plans. The goals also includes operators, human-computer interfaces and the system environments. Along with the requirements comes the design constraints. These can be divided

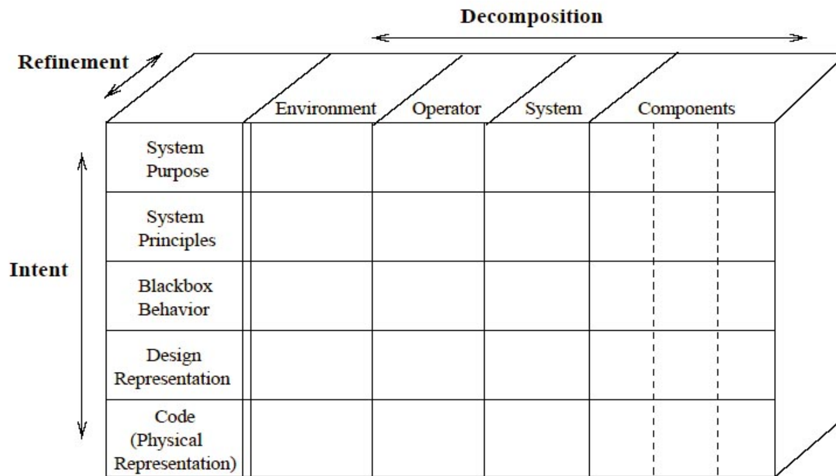

Figure 5.1: The Intent Specification document structure

into normal and `safety`-related constraints. The latter should be linked to the system hazard records and the hazard analysis which results made the constraint necessary. Requirements and design constraints are also made with respect to the environment under which the system will operate.

`hazard` and `reliability` analyzes, e.g. FTA and HAZOP, may point to system limitations. These limitations should be included in level 1 along with pointers to their origin. This could be FTA boxes, PHA tables etc. It is important to discover these limitations early on to be able to set the system boundaries.

The final parts of level 1 are evaluation criteria, priorities and system analysis results. The former two are used to resolve conflicts and guide lower level design choices. Having clear priorities for requirements and design constraints leads to faster development as the designers may not have to use time to investigate trade-offs. The latter is used to document the results of hazard and other analyzes. If the system design is changed, these analyzes may have to be redone.

Level 2: System design principles

This level contains the basic design for the system, and the necessary scientific and engineering principles. These are linked to the upper level, providing traceability. In this way it is possible to see that all the upper level results are covered, and that all principles are actually needed.

The level also contains pointers to lower levels like the black box requirements specification, where the design principles are embodied. Finally, the level will reflect trade-offs that were made for the basic design.

Level 3: Blackbox behavior

Much of the information used to fill in this level is found in the system engineering specifications, making it more of an organizational effort to complete large parts of it. The level specifies the system components and their interfaces, including human users, operators, and environments.

The level does not attempt to describe how the components do their task, merely what the tasks are and how they communicate. This includes assumed behavior of external components and the systems interfaces to its environment.

For modeling purposes Safeware suggests using SpecTRM-RL for describing the components. SpecTRM-RL is a modeling languege developed by Safeware as a successor to RSML and is made to be closely integrated with Intent Specification documents.

Level 4: Physical and logical function

At this level, the component design is described. This is the first level where information about the physical and logical implementation is recorded. Not all of this information may be traceable or linked to the upper levels, as some implementation decisions are not dependent upon the intents. One such decision may be the choice of graphics library classes.

There is still research going on for how the information from this level may be utilized. One proposal is to keep copies of un-optimized code for use in review, as this tends to be more understandable.

Depending on the system in question, this level may also include hardware design specifications, manuals or other information regarding the environment that may influence the design.

Level 5: Physical representation

This level contains a description of the implementation of the system, be it hardware assembly, software and other components.

5.2 ASCE

ASCE is is developed by Adelard[1] to support implementation of `safety` critical systems. ASCE is used as short for both 'The Assurance and Safety Case Environment' and 'Adelard Safety Case Editor'. The latter is a software tool used for developing safety cases[1]. To distinguish the tool from the method, we have adopted the abbreviation ASCEd for the software tool. ASCE is thus the methodology and environment for developing safety cases. Adelard defines a `safety case` as *A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.* A safety case provides a framework for recording and using the results of hazard analyzes. Requirements and decisions regarding the system safety are also collected. While e.g. a FTA is performed on a system in a phase of development (though it may be performed several times on different stages), the safety case evolves along with the system.

This chapter provides a breakdown of the manual on Safety Case and ASCAD that is provided by Adelard. Readers that want to use ASCE or learn more are encouraged to read the manual, which may be found at [4]. The manual provides an extensive explanation of the ASCE methodology that implements ASCAD, along with examples. A safety case is divided into four phases[2], which covers different stages of the development process. As can be seen in figure 5.2 the four elements corresponds closely to the four stages of RUP, making ASCE a good add-on for making RUP more safety related. The elements are:

Preliminary Establishes the system and safety context. Corresponds to the Inception phase of RUP.

Architectural Provides the first level of detail. This element finishes the RUP Inception phase and Elaboration phase.

Implementation Provides arguments and evidence. Corresponds to the Construction phase of RUP, including some parts of the Transition phase.

[1]As 'safety case' is the name of the method as well as the document resulting from applying the method, we separate these by using upper case leading letters for the method and all lower case for the document

[2]Although Adelard calls the parts elements, we use the term phase to make it easier to see the correspondence between ASCE and RUP.

Operation and installation Defines safety related operational, installation and maintenance procedures and requirements. Completes and extends the RUP Transition phase.

A safety case is developed in the context of plants and machinery, e.g. nuclear plants, medical devises, and air traffic control. These systems are usually more complex than the systems that BUCS has in mind, and without doubt more comprehensive. The only problem we get from this is to identify what parts of the safety cases we should omit, while we gain a number of benefits. One of the benefits is that the Safety Case includes what Adelard calls Programmable Electronic System (PES), which in the BUCS context will be the computer hardware. When feasible, inclusion of the hardware in the safety case will lead to more `business-safe` systems as we can take into account `failures` from these parts too.

The first section of this chapter will explain the principles of ASCAD. The rest of the chapter will focus on Safety Case. This part will be divided according to the elements, providing a brief introduction to each.

5.2.1 Claims-Arguments-Evidence

The Claims-Arguments-Evidence (ASCAD) is developed by Adelard [1] for `Safety Case`. ASCAD is not a hazard analysis method, but a notation technique. It helps structuring the `safety` requirements and their implementation status. It also keeps track of the `intentions` behind decisions and requirements, which may be helpful when testing and evaluating a system.

How to do ASCAD

ASCAD is developed in a top-down manner, starting with claims. For each claim, one or more arguments are defined which supports the claim. In the end, evidence shows that the arguments are correct, and therefore the claims hold.

There are several types of claims, depending on the system and intention. The claims are connected to attributes for the system, and whether an attribute is considered to be relevant for the system depends on the developers and customers of the system. Some example attributes are:

- `Reliability` (e.g. MTTF)

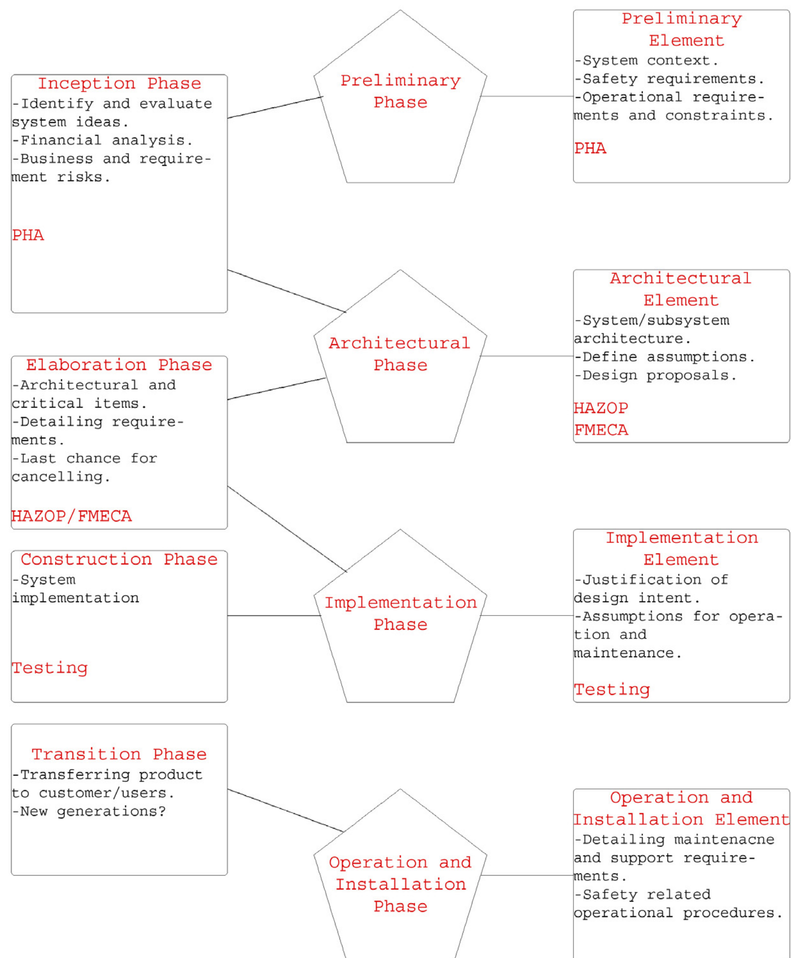

Figure 5.2: The correspondence of RUP and Safety Case

- Response to internal failures
- Fail-safety
- Functional correctness
- Accuracy

When defining claims, the corresponding system attributes should also be identified and quantified. Some attributes of a system are not safety-related, and some safety issues may be resolved outside the software system. The former may usually be noticed and left out of the safety case, while the latter should be recorded in the same way as if they were resolved in the system. This is needed to hold on to the practice of recording all safety decisions and intents, in case something has to be changed later on.

ASCAD usually operates with three forms of arguments, depending on consequences of the claim failing and the effort needed to avoid the `hazard`. The three forms are:

`Fault` **elimination** Get rid of the hazard altogether.

`Failure` **containment** Lessen the impact of the failure, or at least make it happen more seldom.

`Failure` **rate estimation** Estimate the failure rate and reduce it to acceptable level.

The form of argument used will influence the further work on the model. If failure containment arguments are used, the system must be analyzed with this in mind. That means taking into accord not only the probabilities of failure happening, but also the probability of the failure breaking through the containment.

Though the arguments are mentioned before the evidence in this overview, they are not necessarily identified in this order. When stating the arguments, it is helpful to have at least some idea of what evidence may be available, and where it is expected to come from. Evidence is found in a variety of sources, with the following being the most common:

- The design.
- The development processes.

- Experience from simulations.
- Prior field experience.

To save time and effort, evidence from design and prior experience are the most valuable since this makes the developers able to make changes early in the process.

Depending on the level of the safety case, and the available information, evidence takes on one of three forms. **Facts** will support the claim as valid as long as the corresponding argument is used. At the system level, claims and arguments should not change much, while there may be changes to the chain at lower levels depending on design choises. An argument should be a fact, but an **assumption** may be necessary if no facts can be found, but the argument is believed to hold. Ideally, all assumptions should be backed by another piece of evidence to the argument, or be replaced by facts as the process moves forward. The last type of evidence is **sub-claims**. Sub-claims are used to build hierarchies of detail levels. When using a sub-claim for evidence, it is crucial that the chain is linked to the next level. If not, it is a risk that top-level claims are believed to be valid, though the lower level corresponding arguments are false.

Results

The result of using ASCAD will be a chain of claims, arguments, and evidence, providing control of safety issues. When used in a Safety Case, you will have a complete collection of documents with requirements, intentions, hazard analysis results, design decisions, and so forth. According to Adelard, an ASCAD document is not finished before the system is made obsolete. It should be growing and evolving together with the system, reflecting later changes or new versions.

5.2.2 Preliminary Safety Case element

This element defines the structure and range of the safety case. This includes the phase of the project lifecycle and the safety requirements and attributes. Depending on the project, the preliminary element may vary from organizing existing documentation to requiring new analysis. In all cases, the following should be done:

1. Define the system and equipment that a safety case is being developed for and assess existing information about the project.

2. Select relevant attributes and define safety requirements as claims from the attributes.
3. Provide traceability to system and other sub-system safety cases.
4. Establish project constraints on design options and availability of evidence.
5. Assess potential long term changes to the safety case context.

In many ways, this corresponds to the targets of the inception phase of RUP. The preliminary element is mostly consisting of defining and establishing the boundaries for the further work on the safety case. By starting to define the requirements and claims it also sets the path which the safety implementation of the project will follow.

5.2.3 Architectural Safety Case element

This element adds the first level of detail to the safety case. The aim of the element is to establish safety requirements and evaluate design options based on the information found in the preliminary element. The element also uses the results of earlier hazard analyzes to decide on design strategies. When the element is completed, the following tasks should have been performed, and the results added to the safety case:

1. Establishment of the safety requirements either by importing the Preliminary safety case and/or repeating it for changes that have occurred.
2. Evaluation of design options or existing features to assess their relevance to the safety case's claims and attributes.
3. Adoption of a design for the assessment approach to develop a solution for each safety attribute claim.
4. Elaboration of the evidence to show that the claim is met and definition of the evidence that is required to be collected.
5. Identification of the requirements that will be passed onto subsystems to implement the architectural requirements.
6. Undertaking of a `hazard` assessment to identify any additional `Hazard` arising from: random failure, systematic `faults` or human errors in operations and maintenance.

Attribute: Functional Behaviour			
Claim *From the Preliminary element*	Design Features	Assumption/Evidence *Needed evidence (assumption), or present evidence used to substantiate the claim*	Subsystem Requirements *For documentation and tracing of evidence*

Figure 5.3: The attribute table

7. Assessment of any additional hazard introduced by the subsystem to ensure that they are acceptable in the context of the overall safety case.

To help in the development of the element, and the entire safety case, a table should be used for each attribute. The development of the Architectural element may then be seen as the progressive completion of this table. Figure 5.3 shows an example outline for the table. The next sections will provide possible strategies for completing the fields of this table.

Claim

The information of this field is mostly found in the Preliminary Safety Case element, but claims may also be added as the Architectural element is developed. In the latter case, a new iteration of the Architectural element is necessary to complete all fields for the new entry.

Design Features

The general design approach advocated in Safety Case is design for assessment. This implies that the safety system and Safety Case arguments are designed in parallel, which should us to help exclude unsuitable designs and contribute to more realistic trade-offs. In addition to software errors, the design should take into account hardware failure, design flaws and human errors.

Since not all systems benefit from the same design strategies, it is hard to decide upon one design strategy to be followed for all systems using Safety Case. The ASCE manual therefore suggest three general design strategies:

KISS (Keep It Simple Stupid) Making a system simple is usually much harder than making it complicated, but a simple system has many benefits, including reduced costs, improved understanding of the system and the safety case, and reduced risks of delay. Focusing on simplicity early in the process makes the task easier and gives more opportunities to make design choices emphasizing simplicity.

Partitioning according to criticality Dividing the system based on the criticality of the components is used to minimize design complexity. It also makes it easier to allocate resources during development.

Avoidance of novelty Reusing components and/or using established components (including COTS) can reduce the need for providing new evidence to support arguments.

Assumptions and evidence

If there is evidence already available, this is used to support a preliminary argument which shows why the candidate designs satisfy the safety related requirements. Available evidence is usually found in:

- Hazard analysis results.
- Human Error Analysis results. ([13])
- Probabilistic design assessments.
- Qualitative design assessment studies.
- Resource estimates for the implementation and the associated safety case.
- Prior evidence about specific design techniques.
- Independent certification.
- Experience from existing systems in field operations.

Subsystem Requirements

This early in the development stage, evidence will almost inevitably be lacking for some of the arguments. This leads to assumptions being made about the design. These assumptions will have to be verified at later stages. In addition is it necessary to start planning for the life of the system after the development is finished. To complete these two tasks, is it necessary to identify the following requirements:

- Requirements for additional Safety Case evidence to be produced during the development project.
- Requirements for the long-term maintenance and operation of the equipment.
- Requirements for the long-term safety case maintenance.

Other parts of the Architectural element

In addition to the information used to fill in the table, the Architectural element contains a few more parts. The process of choosing a suitable system architecture and safety case should begin. There can be many possible arguments and architectures that meet the safety requirements, and the choices will affect subsystem requirements. To help in choosing an architecture, suitable design options should be identified and preliminary safety cases be developed for each option.

When a design option is identified and a preliminary safety case is drafted, both are reviewed to see if they satisfy the design properties. Among the properties to be evaluated are:

- Do the design implement the safety functions and attributes?
- Are the design criteria of the preliminary element satisfied?
- Is the design feasible?
- Are the associated safety arguments credible?
- Is the approach cost-effective?

Long term `risks` to safety and other life time properties should also be considered.

Finally, after choosing a design, the design and Safety Case argument should be reviewed and a hazard assessment should be performed. Using methods like HAZOP, the developers should identify hazards arising from random failure, systematic faults and human errors. It is also important to assess hazard introduced by subsystems to confirm that they do not contradict the safety requirements of the overall safety case. For final approval of the design and safety case, all stakeholders should be involved.

5.2.4 Implementation Safety Case element

This element completes the arguments and evidence to support the safety claims. The main concerns of the previous elements are to identify the evidence needed, even though they may also provide some evidence in the process. The main effort of gathering the evidence needed is located to the Implementation element, during both implementation and testing. The task of developing the Implementation element aims to:

1. Establish the component safety requirements, either by importing them from a Preliminary safety case and/or by activities specific to this case.
2. Elaborate the evidence to show that the claims are met.
3. Document the results and provide traceability to the appropriate Preliminary and Architectural Safety Case elements.

Evidence takes on many forms, and what sort of evidence is used depends on the system, the claim and arguments in question, design strategies and choices, together with other factors. Evidence is mostly found in validation and verification, testing, and analysis activities. The evidence itself can be a combination of:

- Design features and supporting analyzes.
- Process features and results of the process.
- Experience from previous projects or simulations.

As each Implementation element is completed, supportive evidence for all the components safety claim shall be documented. The subsystem safety cases are then integrated to an overall system safety case. During this integration, evidence may have to be found to support multi-component claims, and claims regarding the cooperation of components. The complete system safety case should contain evidence that:

- The design features, V&V and safety analysis demonstrate that the required attributes were implemented.
- All sub-contracted components have been implemented according to specification and implement their required attributes.
- All deviations are documented, and their impact has been analyzed and justified.

As the Implementation element nears completion, so does the attribute-claim-evidence table. It is important to remember that evidence contradicting or undermining a claim should also be recorded. Such evidence may lead to changes in the design, after which the record is revisited and updated.

As for the previous elements, a `Hazard` assessment and review must take place before concluding the element. The safety case is at this point usually reviewed bottom up, starting with acceptance of the subsystems implementation and safety cases, and then their consistency to the systems safety case. The system safety case is then reviewed to make sure that it is complete and consistent and that all identified hazard have been tracked and resolved. Finally, the system support infrastructure is identified, including:

- A supporting document set.
- System operations and maintenance requirements.
- Technical resources.
- Safety case maintenance infrastructure.

5.2.5 Operation and Installation Safety Case element

Much of this element is addressed in the earlier elements, as they all look into operational, maintenance and installation aspects of the system. The main task of this element is to complete the plans for these aspects by collecting

and organizing the information found in previous elements, and add any missing or changed information. It is also necessary to define requirements and instructions regarding aspects that the developers are not competent to define. This may include training requirements for operators, user permits and other issues demanding knowledge of the wider system and environment.

It is also part of the element to confirm the support infrastructure identified in the Implementation element. As the safety case must remain alive for the entire lifespan of the system, this element is formally not concluded until the system is discarded. Changes to the system, environments and requirements should be reflected in the safety case and acted upon by performing the corresponding activities.

Chapter 6

Tools

This section will discuss two software tools that we consider suitable to help us in our study. These are ASCEd[1] from Adelard [1], and SpecTRM from Safeware Engineering Corporation [5]. As this discussion is based mostly on the presentation of the tools from their makers, we need to take a closer look at each to decide which, if any, should be used. The section will first present each of the tools. We will then do a comparison and based on this we will decide if we will use any of the tools.

6.1 ASCEd

ASCEd has recently been released in version 3.0. It is a plug-in architecture supporting a variety of notations used in hazard analyzes and safety planning. The default notations are Goal Structuring Notation (GSN) and Claims-Arguments-Evidence (ASCAD). Other notations include Why-Because Analysis and Goal Question Metrics (GQM). The extra functionality comes in one of two forms; notations are usually based on XML and called schemas, while plug-ins enhances the functionality in other ways. The additional plug-ins and schemas are available for downloading from [5].

The ASCEd provides a graphical structure combined with narrative hypertext, making it easy to link nodes to additional information. For ASCAD the tool will link the Evidence to the Arguments, giving a simple and understandable picture of the analysis. For printing and sharing, ASCEd includes

[1]The correct name is ASCE, but as we explained in 5.2 we use the term ASCEd to avoid confusions.

several report generation tools that presents the analysis in doc or html format. ASCEd also includes a tool that can be used to compare two networks for structural and content differences.

The interface of ASCEd is simple, and provides a good overview of your networks. To improve workability and control, ASCEd provides the opportunity to work with your network in several views. This gives you the choice of only seeing some nodes, and not the entire network. This functionality is also used for subnetworks. You may also watch your network as a table.

6.2 SpecTRM

SpecTRM (Specification Toolkit and Requirements Methodology) is a tool from Safeware Engineering Corporation (Safeware) [5] that implements their software safety method, the Intent Specification. Using SpecTRM is supposed to help discovering `safety` issues early in the development cycle so that costs and efforts of resolving these are minimized.

The SpecTRM tool is built using the Eclipse framework, which provides a familiar interface. Using the Eclipse architecture also opens up for writing more plug-ins that can enhance the SpecTRM and customize it for other needs. Nancy G. Leveson's paper on Intent Specifications [10] suggests a number of possible enhances to the Intent Specification. Some of these may lead to the need for additional functionality in SpecTRM. By using Eclipse, such functionality may be added without having to change the original tool.

Along with the Intent Specification, Safeware has developed some techniques. This includes the SpecTRM-RL, which is a modeling language made for use with Intent Specification, and that is supported in the SpecTRM tool.

At first glance, SpecTRM provides only a template for Intent Specification documents consisting of an outline of the specification and functionality like the ability to create links between the levels. It should be possible to simply work your way through the outline, just copying the sections when you use more components, and in the end you have a complete Intent Specification document. At this point, SpecTRM enables you to visualize your system in several views that is intended to improve your understanding of the system. There are also functions for running simulations of the system to evaluate whether all safety requirements are taken care of. Finally, SpecTRM provides tools for robustness and determinism analysis of your specification.

Chapter 7

Today's practice

This chapter will look at how `business-critical` systems are developed today, starting with whether it is any established practice or not. In the latter case we will look at the basic RUP methodology and what it is missing for the business-critical needs to be fulfilled. We will base our discussion upon the results from [17], in which a selection of Norwegian companies explains their standings when developing business-critical software.

As the answers from different companies on most questions are similar, the survey seems to provide good insight into what the problems are of today's methodologies when developing business-critical software.

Most of the companies use RUP or parts thereof when developing business-critical software. As using RUP in many cases is a requirement from the customers, developing our method based on RUP seems to be a good decision. In this way, many of the users will already be familiar with the terms of the method. Of the developer companies that do make software that is not business-critical, most use the same methods for all kinds of software, the difference, if any, being that they are more careful when developing business-critical systems. Again, basing our work on RUP will hopefully provide an environment suitable for all kinds of software, while at the same time taking care of any special needs for business-criticalsystems.

According to the interviews, communication is a common problem along with integration. Some of the communication problems stem from changes in the customers organization and that the developers need to work along with different groups from the customer. Differences in methodology and knowledge are also leading to troubled communication in addition to being a problem in its own right.

To structure our discussion and making it easy to use in our further work, we will divide this chapter according to the phases of developing software as depicted in [19]. The translation from this model to RUP is straightforward, as the activities are more or less the same, but have different names.

7.1 Requirements Engineering

For some projects, this may actually be the second phase. Projects developing new software for the market will have a pre-study phase to gather and analyze information. The results may be used in deciding whether to go on or cancel the project, or as a basis when developing the requirements.

For business-critical software, this is an important phase. It takes structured and good communication between customer and developer to get the requirements right. According to the answers in [17] this phase is conducted in various ways. Some of the developers use some method or another, but not consistently and often just parts thereof.

This phase should also include hazard analyzes - at least a PHA should be conducted to bring forth as many `hazards` as possible early in the development process. Whether the requirement is coming from the customer or from a hazard analysis result; the intent and reason for the requirement should be documented. If this is not done, requirements may be incorrectly implemented. This is especially a danger when it comes to requirements that are only indirectly implemented, such as .

To ensure that all the requirements made are being implemented, test planning should start already in this phase. It is usually easier to write the test while the backgrounds for the requirements are still fresh in mind, but the test plan also contains useful information for later phases.

7.2 Design

When the requirements are finished, it is time to start working on building the system. There is a lot of design methods available, with different pros and cons. Most of the companies in [17] use elements from UML, often use-cases. Use-cases is also the preferred method in RUP. From the interviews, only one of the companies is using any hazard analysis methods. All of the companies are mostly concerned with the functionality of the product, as is many of the customers.

Again, an important part of this phase is to do a hazard analysis. This is the last phase were changes can be introduced at a low cost, while at the same time the system is developed enough to get a clear picture of both functionality and solutions.

7.3 Implementation

Ideally, all hazards should be discovered and resolved by the start of this phase. While not giving any guarantees for the results, at least this increases the possibility of the final system being safe and valid. The interviews in [17] does not put any emphasis on how the developers organize this phase. On the other hand, most development methods are more concerned with the final structure of the code, and do not put much emphasis on how it is made. Different programming languages also demand different approaches.

7.4 Testing

For business-critical software, this is another crucial phase. As mentioned before, some requirements may be only indirectly implemented, and these are easily overlooked if the testers are mostly concerned with making sure that the application runs smoothly. It is not clear from [17] if the developers have any routines for how to test, but according to the methods they are using, we assume that there are no special precautions taken for testing business-critical software compared to other software.

Chapter 8

RUP in BUsiness-Critical Software

In this chapter we will discuss how the presented analysis and documentation methods may contribute to making RUP more suited for `business-critical` software, particulary with regard to the use of barriers. The discussion will lead to a suggestion of which methods to use, and how to use them.

We will begin the discussion with some general issues, after which we will organize the chapter according to the RUP phases. If a method span several phases it will be divided so that the relevant parts are covered within each phase. In case such a dividing is unfeasible, the method will be fully discussed when first encountered.

8.1 General

This discussion does not attempt to cover all possible methods that could be used for customizing a development strategy for business-critical software. The reasons for this are many, the main reasons being time constraints and knowledge, along with the numbers of more or less well known methods to choose from.

When it comes to using RUP, this was not a choice for us to make, as RUP was already decided to be the method on which to build BUCS. According to the interviews made early in the BUCS project, most developers are already using some version of RUP. This is not only a decision made by the developers, but also in some cases a requirement from the customers.

The use of RUP also puts some strong suggestions to what hazard analysis methods we are going to include.

The hazard analysis used for RUP are well known methods, and complement each other well. Using the same methods for both project and product `hazards` lowers the costs and effort needed for schooling analysts. As we later discovered, our techniques for `safety` development also advocated methods that are present in our selection. Based on this, we find that our selection will satisfy the needs of BUCS.

For recording and control purposes we chose to consider Intent Specification/SpecTRM and ASCAD based upon recommendations and availability. These methods cover the entire process, so we will discuss the relevant parts in each phase.

8.2 The Inception Phase

As we explained in the RUP overview (3.1), the Inception phase (3.1.1) is aiming at deciding whether the project is worth running, while the product is kept at a conceptual level. RUP uses this phase to establish a business case, evaluating if the proposed product and project meets the stakeholders' business requirements. The phase should also establish the project's scope and boundaries, and begin to develop system use-cases.

Though we only have a conceptual design of the product, it is already time to start looking at the `safety` aspects. In [2] the following artifacts are identified as safety related:

- Requirements, leading to a System Test Plan
- Identification of key functionality
- Proposals for possible solutions

At this stage of development, PHA (4.1) is the advocated hazard analysis method. The information available is not sufficient to run a complete HAZOP or FMEA, but a PHA will be able to provide valuable input to the hazard records. Neither the analyzers nor other stakeholders should make decisions on how to avoid the discovered `hazards` at this point. Without any designs there is no way to know the best strategies, except in general ways. The results should, however, be used when making and chosing between design

options, as many hazard are depending on specific designs or manners of operation.

The Intent Specification is not contributing much in this phase. The uppermost level, System Purposes (5.1.1), is not concerned with the process attributes, but some of the concept information can be filled in. Except from this, the work on the Intent Specification document will begin as we move on to the Elaboration phase.

Using ASCE, we will work on the Preliminary phase of ASCE (5.2.2). Like the Inception phase of RUP, the Preliminary element is concerned with concepts. We start defining system limitations and boundaries, along with constraints on design options. In [2], it is proposed that BUCS should combine the Preliminary element and the Inception phase. They state that one of the characteristics of the Inception phase is to "Establish safety requirements". While this implies the need for hazard analyzes to complete the element, we do not think that these analyzes should take place in the Inception phase. Rather, we think that the Preliminary element should span both the Inception phase and the Elaboration phase to maximize the synergies for the hazard analyzes. Using the candidate architecture developed in the Inception phase of RUP we will be able to identify the safety attributes of the system. We should then develop the system safety requirements at the start of the Elaboration phase as we are baselining the architecture. In A.2 we show a PHA table, and demonstrates how the first claims are added to the safety case.

Since RUP aims at developing at least one candidate architecture in the Inception phase, completing the Preliminary element in this phase requires near to complete requirements specifications for each candidate. We think this is a waste of time that is better spent in the Elaboration phase. We also want to make a point of that HAZOP and FMECA are usually performed in the Elaboration phase for RUP. By letting the project and product hazard analyzes run together, we believe that the analyzes will be better able to learn from each other and discovering interconnected hazard.

If there is available information on the PES on which the system will run, this is a good time to start looking into what requirements may be needed for this. The PES may also add constraints to the software system.

8.3 The Elaboration Phase

According to our presentation in 3.1.2, this is the design phase of RUP. Though it is still possible to cancel the project, by now we should have control of most aspects except design problems. As we start developing design ideas, we will get more information about both the product and the process. This information can be used to classify the `hazards` from the PHA during the Inception phase. Though some hazard may not be present in some designs, they should not be removed from the records, as later design changes may reintroduce the `Hazards`. It is also a possibility that the hazard may return as a result of maintenance or upgrading of the system.

For both project `safety` and product safety, this phase is important. As we begin developing the details of the system, we also identify the information needed for running more concrete hazard analyzes. Using HAZOP (4.3) and FMECA (4.2) provides complementary results that cover almost every `failure` that can be discovered using the information available, and also gives much information about the causes. Information from the hazard analyzes, including the PHA from the Inception phase, should be used to decide what design options to explore. The hazard analysis results are also used when we make decisions and trade-offs between the designs. hazards discovered at this time are normally handled at much lower costs now than during implementation. Part A.3 of the appendix demonstrates the use of HAZOP and FMECA as complementary analyzes, and also the use of the analyzes after the hazards discovered during the first iteration are included in the design.

After changing the design according to the hazard analysis results, a new iteration of hazard analysis should be performed to make sure that no additional hazards has been introduced. If several design options are explored, hazard analyzes may be performed at different levels of detail to help choose between the options. A FTA (4.4) is also useful, as this is more aware of the causes behind hazards and failures. According to the hazard handling strategy of BUCS, the design options should aim to avoid the hazards. If barriers are necessary, this is the best place in the development cycle to include them. As always, documentation is crucial.

Even though the test phase is a long way ahead, a system test plan should already be in place. When decisions are made to avoid hazards, suitable tests for the hazard should be developed at the same time. This is important for hazards that are believed to be avoided, and crucial if a barrier is used. Along with the test description, we should record the reason for the test. Linking to the `Hazard` in question can also prove helpful if the test uncovers any

problems. An example testplan is developed throughout the demonstration process of appendix A, starting out with little detail and then filling in the test as we design the system.

The Elaboration phase fills level 2-4 in the Intent Specification document. By getting gradually more specific, we are in control of the system and it is easy to make necessary changes. In RUP, use-cases are used to describe the system. Intent Specification uses SpecTRM-RL, a language developed by Safeware. While this language fits neatly into the Intent Specification, and also is supported in SpecTRM, it is a drawback that it is not commonly used in the industry. Mixing modeling languages may lead to errors when information is to be exchanged between models using different languages.

When linking tests, hazard handling and the hazard, the intent idea is useful. One of the main ideas behind the Intent Specification is to create a connection from the intent to the solution. Safeware believes that this connection leads to better understanding of the system and the decisions made as the system is developed. For BUCS, Intent Specification is thus helping to ensure that all hazards that can be avoided really are avoided.

ASCE does not put much emphasis into one modeling language or another. A model is not part of the Claims-Arguments-Evidence chain, but is linked to provide information. This gives the developers more freedom to use whatever modeling languages fits the system best, and that they are most familiar with. As RUP is advocating use-cases, it is feasible that use-cases may be easily fitted into the recording scheme.

The Elaboration phase will finish the Preliminary Safety Case element and the Architectural Safety Case element. The Preliminary element will then contain the system properties and attributes that lies behind the system claims. The claims are then transformed into requirements that are used to develop design options in the Architectural element. The Architectural element aims to identify arguments that supports the system claims, along with the evidence needed. Using the hazard analyzes results and the requirements, ASCE helps both exploring design options and creating test plans. Developing claims and requirements for subsystems also takes place in the Architectural element. This because the subsystems are not identified till exploration of design options has started.

8.4 The Construction phase

As we explain in the RUP chapter (3.1.3), this is the largest phase in RUP. By this time all `hazards` should be discovered and a plan for handling them has been developed. Discovering new hazards at this time will usually lead to a costly task of resolving them.

Neither RUP nor we put much emphasis into how the implementation is conducted. In the Intent Specification document, the implementation makes up the fifth level, Physical Representation (5.1.1). As we explained in our chapter on Intent Specification (5.1.1), the level consists merely of the implementation.

The Implementation Safety Case element adds completion of the Claim-Arguments-Evidence table to the tasks of the phase. Some of the evidence needed to support the arguments are found in the implementation itself, but for `safety` issues, most evidence will stem from the testing. Developers should be cautioned that most hazard handling strategies are not completed until the testing has proven that the hazard is handled according to the requirements. To complete the element, and thereby the Construction phase in our proposal, all claims about the system must have been provided with evidence. The only arguments which may still be unresolved are concerning lifecycle and maintenance of the system, these will be addressed in the next phase.

Though the element does not directly address how the system is implemented, the organization of the safety cases makes some suggestions. As the safety cases are recommended to be completed bottom-up, the same strategy should be followed for the implementation to keep the work on both in parallel, meaning that the subsystems and their components are to be implemented first.

8.5 The Transition phase

While RUP mainly uses this phase to deliver the product to the customers/users (3.1.4), this is also the right place for the Operation and Installation Safety Case element (5.2.5). Planning the installation and maintenance of a system requires close contact with the customers and the maintenance team. The Safety Case element also includes planning for user training and other aspects demanding a broader knowledge of the system than what the developer team

possesses. In the Intent Specification, this phase is mentioned, but it does not contain any specifics for how it is to be performed.

Whether a new generation of the system is planned or not, it will usually be necessary to develop upgrades and patches. Feedback from customers and users may also lead to changes in the system. When designing and implementing these changes it is crucial to consult the hazard records. In many cases, new hazard analyzes are also necessary to ensure that the changes does not introduce new `hazards`. In [10] these challenges are addressed as areas that needs new methodologies, proposing Intent Specification as a framework for including the required information.

The Safety Case element is, as the name suggests, addressing both the transition and the follow-up of the system after delivery. While much of the information and parts of the Claims-Arguments-Evidence chain for the systems life span is provided from the previous `Safety Case` elements, there are still the need for organizing and completing this work. The element provides requirements for the support infrastructure, both for the product and the `safety case`s. Without neglecting the importance of this phase, our demonstration does not provide much details for this phase as this is very dependent upon the system, the developer - customer - user relationship, and the manner in which the system is developed. We still mention the phase in our appendix, A.5.

Chapter 9

Barrier Conservation

Even if it might seem to be forgotten earlier in this report, the primary concern of our work has been how to make sure that the barriers needed in a system are implemented. However, we would not be able to propose any solutions to this without having a process to start from. Without a process, our recommendation would have sounded something like *document and test all barrier requirements*. By outlining a process and suggesting analysis methods, we are able to extend and deepen this recommendation into something useful.

When introducing barriers, the reasons for the barriers may vary. The most common source is hazard analysis results. Still, the decision of using barriers to prevent a `hazard` should lie in the hands of the designers. We do not recommend that the analyzers include recommendations of how to handle the hazards except on special occasions. As for what hazard analysis methods to use, we do not find big differences between the methods in the context of barriers. To discover as many hazards as possible, we recommend to use complementary methods, like HAZOP and FMECA.

As we have stated several times, one of the most important tasks when dealing with `safety` is to record everything. This also goes for barriers. Each decision of using a barrier should be linked to the hazard it aims to prevent, the reason for using a barrier, and for using this particular barrier, to avoid the hazard, the design used for the barrier, and the test plan for the barrier. This documentation and linking should be easy to read and understand, in addition to include all necessary information on each part.

In this report we have looked into two proposals for recording our safety concerns, the Intent Specification and ASCE. Both of these methodologies are developed with safety in mind, but primarily in the classic meaning of

the term. However, there is little customization needed for using them for `business-safety`. Whether one is using Intent Specification or Safety Case, it is mostly a matter of omitting some parts which are not needed. This includes environmental dangers, which are handled in ordinary safety development.

The difference between the Intent Specification and ASCE is more than only a choice between two toolkits, it is a choice between two different methods. The Intent Specification is more or less built from scratch based on research and techniques that has not been much used in software engineering. According to [10], Intent Specification is tailored to how humans think and reason to a greater extent than traditional methods. Using principles of cognitive psychology along with system engineering theory, Intent Specification is designed *to enhance human processing and problem solving, to integrate formal and informal aspects of software development, and to enhance our ability to engineer for quality and to build evolvable and changeable systems.* To document and conserve a barrier, the Intent Specification will focus on recording the intent behind the barrier and link this to the design containing the barrier.

The Intent Specification does not contain dedicated sections for test results, which can be a problem. Even the most perfectly programmed barrier is no good if it does not avoid the relevant hazard. Documenting the results of testing the barrier and linking this to the other records concerning this barrier is necessary to be able to validate that all hazards are handled as planned.

In ASCE, a hazard or, more correctly, a hazard avoidance requirement will be introduced as a claim. Connected to this claim should be a record of the hazard. This record may be just a pointer to the appropriate hazard analysis table, or it may be more detailed if such details exists. If the stakeholders decide upon the use of a barrier to avoid this hazard, that decision will be an argument for the claim. The argument should link to a document explaining why it was decided to use a barrier to avoid this hazard, and how this barrier is intended to work. Finally, the testing will provide the evidence that the barrier has been implemented and is working properly.

If, on later iterations or during maintenance, changes are made to the implementation of the barrier, or the design containing the barrier, the claim-argument chain has to be updated. Changes to the design will require new hazard analyzes, and eventually introduction of new barriers. Every time a decision leads to a change of the claim or argument, it is also necessary to reconstruct the other parts of the chain to make sure that the claim still

holds. We believe that the ASCE contains the necessary tools to do this in a simple, but powerful manner, thereby providing the means for developing "business-safer" software.

Chapter 10

Conclusion

10.1 Choice of tools

Like all other hazard decisions, it is important to record information and decisions concerning the use of barriers to avoid `hazards`. Based on our previous work, we find that ASCE provides the best means of making sure that any needed barriers are properly implemented. While the Intent Specification seems to be a more complete tool than ASCE, it relies heavily on using its own methods. This leads to a demand for more training and learning of new techniques than what is necessary for ASCE.

We want our proposal to fit into a developers practice with as little ado as possible. ASCE mostly provides a new way of recording and organizing information, which in many cases is the most efficient way of ensuring barrier conservation. ASCE also gives more attention to the maintenance work of a project.

When choosing between ASCE and Intent Specification it is also necessary to consider the application tools used for the methods. The Intent Specification is bundled with SpecTRM, which is a template for the Intent Specification document. It comes with some nice functionality, including abilities for simulating the Intent Specification . Once again, the problem is that SpecTRM is made for Intent Specification alone and provides little support for other methods.

ASCEd, which is used to support ASCE is a supporting tool for organizing and controlling the development. Instead of containing the entire safety case it is used to make the Safety Case chains, and hyperlinking to the Safety Case documents. ASCEd is also supporting other notations, among them

GQM and Goal Structuring Notation. ASCEd's ability to generate reports is also important.

The Intent Specification 's use of links to enable tracing the levels in an efficient manner is making the specifications easier to read, but a complete Intent Specification document is nonetheless a large document. An example is mentioned in [10] and consists of more than 600 pages. The Safety Case is containing the same amount of documentation as the Intent Specification , but does not organize it within one document. As systems grow large and more information is needed, the safety cases linking strategy is better than the organization of an Intent Specification document.

That BUCS is using RUP as their process framework is also making ASCE more suitable than Intent Specification . The four elements of Safety Case fits nicely together with the four phases of RUP, making the result more streamlined. ASCE can make use of several modeling languages for a system. This is also an advantage since RUP is so associated with use-cases.

Last, but not least, we think that ASCE is better at handling the use of barriers for our purpose. One of the most important aspects of barrier implementation in our proposal is to make sure that the barriers are preventing the right hazard. The best way to validate this is through testing, and we find that ASCE includes the test results better than the Intent Specification .

10.2 Using ASCE for barrier conservation

We have previously discussed how ASCE and Intent Specification can be used to help us ensuring that barriers are implemented as intended, see chapter 9, in which we concluded that ASCE is the best approach to barrier conservation in BUCS. Though we find that a complete manual for using ASCE in BUCS will be far beyond the scope of this report, we will give a summary of our work and discussions to show how ASCE can be included in RUP.

In our section on integrating ASCE in RUP (chapter 8) we state that we disagree with the authors of [2] when it comes to where the Safety Case elements are developed. When working with barriers, it is possible to choose either of the proposals. A development team should never decide to use a barrier till their system is designed. A barrier should be the last resort for avoiding a hazardous state. It is far better to remove the state from the system, so that it cannot be reached at all. Only in those cases where this is

impossible or highly unfeasible should barriers be used, and then only with outmost care.

The first step towards implementing a barrier is taken before the decision of using a barrier is made. After a hazard analysis, the hazards should be converted into claims for the system. Or, to be precise, the avoidance of the hazard should be introduced as a claim. As the claims turn into requirements for the system, the designers will have to develop solutions for satisfying the requirements. As mentioned above, many solutions are better than introducing barriers, but in cases where a barrier is needed, this decision is linked to the relevant claim(s) as an argument. The argument should clearly state why proving this claim will satisfy the requirement. Along with the argument we should also decide what evidence is necessary. When using barriers, this evidence will usually have to be one or more tests.

The test should, at least, be outlined at the same time as the barrier is designed, and it is crucial that a passed test leads to the requirement being satisfied. The best strategy is to have the test designed before the barrier, at the same time as the decision on using a barrier is made. This will ensure that the test fits the requirement, and not the barrier design. If the test is designed to make sure that the design of the barrier is safe, a faulty barrier may be considered valid.

A completed test will provide evidence that the barrier functions properly. The documentation should include a description of the test, preferably also why the test was designed the way it was, the test results and any comments on the test. This makes it easier to replicate the test, in addition to providing valuable information for fixing the barrier if the test fails. If the barrier design is about to change, it can be valuable to know if the test discovered properties of the barrier that may cause problems to a change.

10.3 Further Work

We hope that this report is providing a framework for the use of barriers in BUCS, and in addition a platform for the fusion of RUP and ASCE. In our work we have focused on the principles behind BUCS, RUP, and our suggested methods, and there is much work remaining before BUCS becomes a fully useable methodology.

It is our intention that the information upon which we have built our discussions and drawn our conclusions provides enough details so that it will be possible to propose enhancements and improvements to our suggestions.

Anyway, it is necessary to develop the methodology in more details and fine tune the somewhat raw outline that we have developed.

Appendix A

Demonstration of concept

This chapter will provide an example of the concepts described in our report, using the methodology from chapter 8. To do this, we select a few parts of a system that we follow throughout the development process, showing how we think the ASCE methods should be merged with RUP. Our example system is not safety critical in the traditional meaning of `safety`, except for the possibility of people suffering from depressions due to huge budget overruns. On the other hand, the `business-safety` aspects of the system are clear. Any `fails` in the development of the system may lead to huge losses for the customer.

A.1 Conceptual description

Our system is used for handling orders in a mail-order store. The store has a number of expensive items, leading to some orders being very profitable. To encourage customers and gaining an advantage in the market, large orders and regular customers receive a discount on their order. Depending on the size of the discount, an additional confirmation may be needed before the discount is subtracted from the invoice.

A.2 Inception phase

During the Inception phase, the developers meets with the representees of the store to evaluate whether this system can be developed within the proposed budget, according to the rules of RUP.

Hazard	Cause	Main Effect	Preventive Action
...	...	...	...
Erroneous punching	Misreading/ punching failure	The order or bill may be wrong	Double checking routines
Wrong discount inserted	Wrong calculation	The customer receives a too high or too low discount	High discounts are to be approved by supervisor
...	...	...	...

Table A.1: The PHA Table

The developers and store representees are also performing a PHA based on the concept description to uncover possible `hazard`s that needs to be taken care of in the system. Table A.1 shows an excerpt of the PHA result table, with some possible hazards. Even though we only have a conceptual description it is already possible to list a number of hazards.

One of the hazards that is likely to show up during the PHA for this system is the possibility of entering a wrong value when entering orders into the system. This may be too many or too few items, wrong prices or a wrong discount. Many of these errors are difficult to avoid using software, though the order may be checked for reasonability. Mostly this has to be dealt with by creating good manual routines.

Using the results of the PHA, we start to define the first requirements for the system. At this stage, many requirements will be qualitative and stating wanted or unwanted attributes of the system. These requirements are included in the ASCAD as claims, giving us the first blocks of the safety case. Based on the concept and PHA we state that:

- The system should provide measures to avoid errors when registering orders.

This requirement gives us the first claim in our Claims-Arguments-Evidence network. The claim is linked to the requirements specification and the PHA to provide information about why this claim was included. In figure A.1 we show how the ASCE-tool looks after we have added the first claim in the model along with a description of the claim and links to the relevant documents.

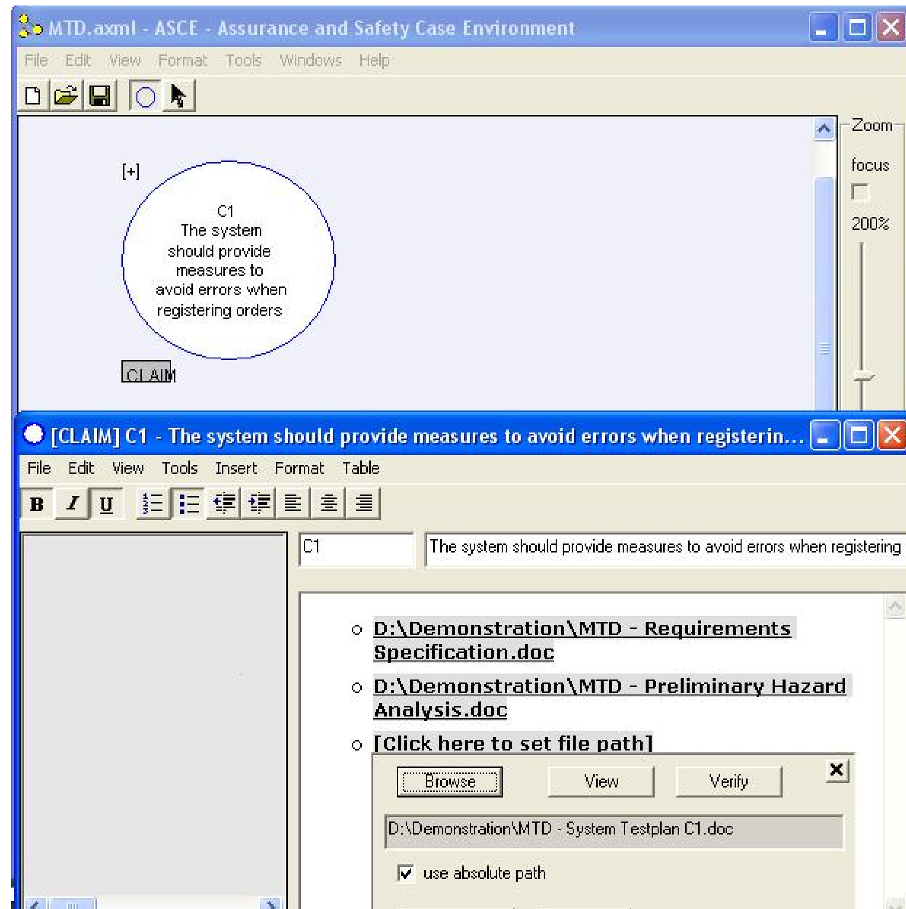

Figure A.1: Adding the first claim to the ASCE network

According to chapter 8.2, we should start to develop the system test plan based on our requirements. Our requirement is not yet detailed enough to make us able to write a complete test, but we can provide some headlines to be filled in as we are getting more details about the hazard. We have proposed to split the Preliminary Safety Case phase between the Inception and Elaboration phase of RUP, and are not detailing the test plan yet as we do not have a design to base it upon. Table A.2 is a snapshot of how the system-test for our first claim may look like. Note that we are already assigning the tests to people, thereby making it easier to follow up the testing and results.

We do not dwell too much on the RUP part of this phase. The tasks found in the Inception phase of RUP along with the goals of the milestone are mostly concerned with the budget aspects of the project. In this example we assume that the stakeholders are satisfied by the project plans.

A.3 Architectural phase

It is now time to start investigating how the claims from the Preliminary phase will be met. We also begin to develop a system design, which in turn will enable us to develop more claims. From the information collected during the design iterations we can also find some of the arguments needed to support the claims.

The first task that should be undertaken in the Architectural phase is to import the claims from the preliminary element. Using ASCEd the first step is to continue working on the Claims-Arguments-Evidence network. We have the claim from the Preliminary phase, but this is a vague claim. One of the tasks is therefore to develop this claim into one or more detailed claims.

In the transition from the Preliminary to the Architectural phase we have developed the first top-level system designs. These are merely drafts that are used to get information for the hazard analyzes. Each design idea requires its own HAZOP and FMECA. Because of the workload this imposes, it is unfeasible to explore every possible design. To be able to cover as many alternatives as possible we will run several iterations of the hazard analyzes, discharging troublesome designs as we go.

For our demonstration, we are not developing the designs, as this will require much more information about the system than what is needed to demonstrate our concepts. It is also difficult to provide an example to show what HAZOP should discard a design, as this is dependent upon project

System test for	C1			
Tester	'Person A'		Results manager	'Test leader'
Date planned	ddmm	Date conducted	ddmm	
Resources needed	??		Time needed	??
Steps	Done	Expected result	Achieved result	Comment
1. Add correct order		Correct order registered		
2. Add wrong quantity measurement		Error message, order line is not registered		
3. Exceed maximum items allowed		Error message, order line is not registered		
4. Too few items in order		Error message, order is not registered		
5. Add erroneous discount		Error message, discount is not registered		
Approved	Yes/No	By		
Date	ddmm			

Table A.2: The system test plan

specific factors like the designers knowledge, programming environments and stakeholder's decisions of how the hazard is classified. Whether the designers are able to figure out a way to overcome the hazard or not is also an important factor.

Not all hazards are dependent on the design - some are present due to the system. To avoid having to investigate these hazards again and again, a document containing common hazards can be developed and a link to this document will be added to all designs. This can be hazards connected to the functionality of the system, hazards stemming from input outside the developers control, or customers demands.

For our system, a design decision is made that the operator is able to give discount to a customer based on the size of the order and the customer's history. The store has a profile where the rules for awarding the discount is somewhat vague, leaving much of the decision to the operator. For the designers to receive the information in a structured way that will help them decide on the most feasible design for the system, we are developing a use-case that describes the functionality. The use-case is shown in table A.3. Numbering the use-case makes it easy to connect it to the relevant claims and requirements. In addition we are including links to the use-case document wherever necessary in the ASCE network [1].

The discount functionality is to be implemented no matter what design is chosen for the system, and it may therefore be analyzed for hazards in a design-independent analysis. Before the analysis is conducted, we are including this new information in our Claims-Arguments-Evidence network. The new claim that is to be included in the information used for the hazard analyzes is formulated:

- An operator can give a discount on an order based on the store rules.

Our ASCE network now has two claims (figure A.2, both of which are to be used in the hazard analysis. It is important that no information is held back from the analyzers, even if we think that it is not providing anything useful, or that it has been analyzed before. Many hazards are the result of interrelations of information and components, and even analysis techniques that do not take this interrelations into account may discover more hazards when the analyzers are able to see the system as a whole.

In our case, the combination of our two claims is clearly pointing one hazard. When the operator is deciding upon the discount as he is entering

[1]For more on use-cases and their uses, we suggest readers to consult [8].

Use-case name	C2 Operator discount
Iteration	Filled
Summary	An operator wants to give a discount to an order
Main path	1. The operator registers the order 2. The operator enters the discount 3. The operator approves the order 4. The system accepts the order
Alternate path	1. The operator registers the order 2. The operator enters the discount 3. The operator approves the order 4. The system does not accept the order
Intent	The operator should be able to give a discount on orders

Table A.3: The discount use-case

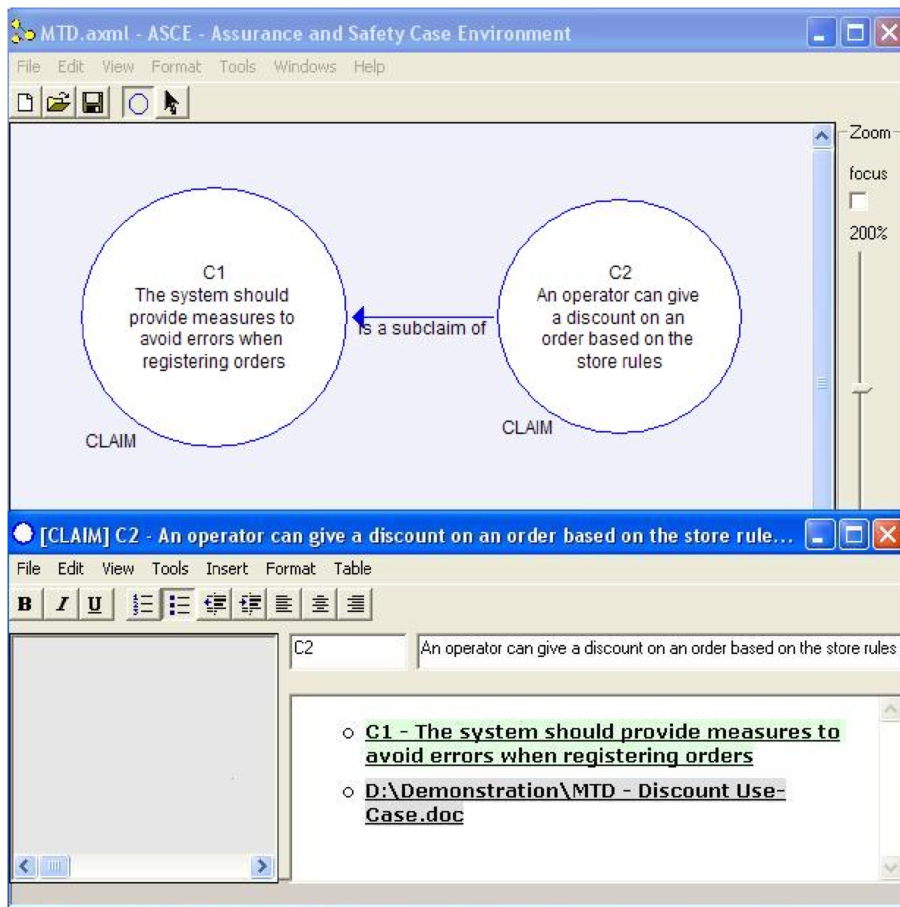

Figure A.2: Our second claim, linked to external documents and the first claim

Guide-word	Hazard	Consequences	Reason	Suggested solution
No	The operator inserts no discount	...	...	...
Too much	The operator inserts too high a discount	Loss	Misreading/ -punching, misbehavior	Warnings/ alarms
Too little	The operator inserts too low a discount	The customer receives a too high invoice	Misreading/ - punching, misbehavior	Negligible
...	...	...	...	...

Table A.4: The HAZOP table

the order, it is a possibility that she is giving the wrong discount. Too low a discount is not so much a problem, as the store policy is not to promise any amount of the discount on beforehand. Too high a discount is more troublesome, as this may lead to the store loosing money. The hazard is discovered during the HAZOP analysis, and is documented in table A.4

This is not to be interpreted as a complete HAZOP table. Each guide-word may contain several hazards, or no hazard at all. As much information as possible regarding the hazards, their consequences and causes should be recorded. We are only including what we find necessary to show our proposed methods, and not what is needed for a complete manual to system safety engineering.

The new hazard calls for a new addition to our ASCE network. To show that the hazard of a customer receiving too high a discount is related to both the claim regarding erroneous entering of data and that the operator is able to give discounts, it is natural to have a link from our new node to both the previous claims. To achieve this, we add a sub-claim, that is connected to the two other claims. We phrases the claim as:

- There should be mechanisms in the system to avoid too high discounts being granted to a customer.

As we move along, it is important to remember that we also have to update our testplan as we discover new claims and hazards. There is no specific type of testplan or testing strategies connected to our proposal, so

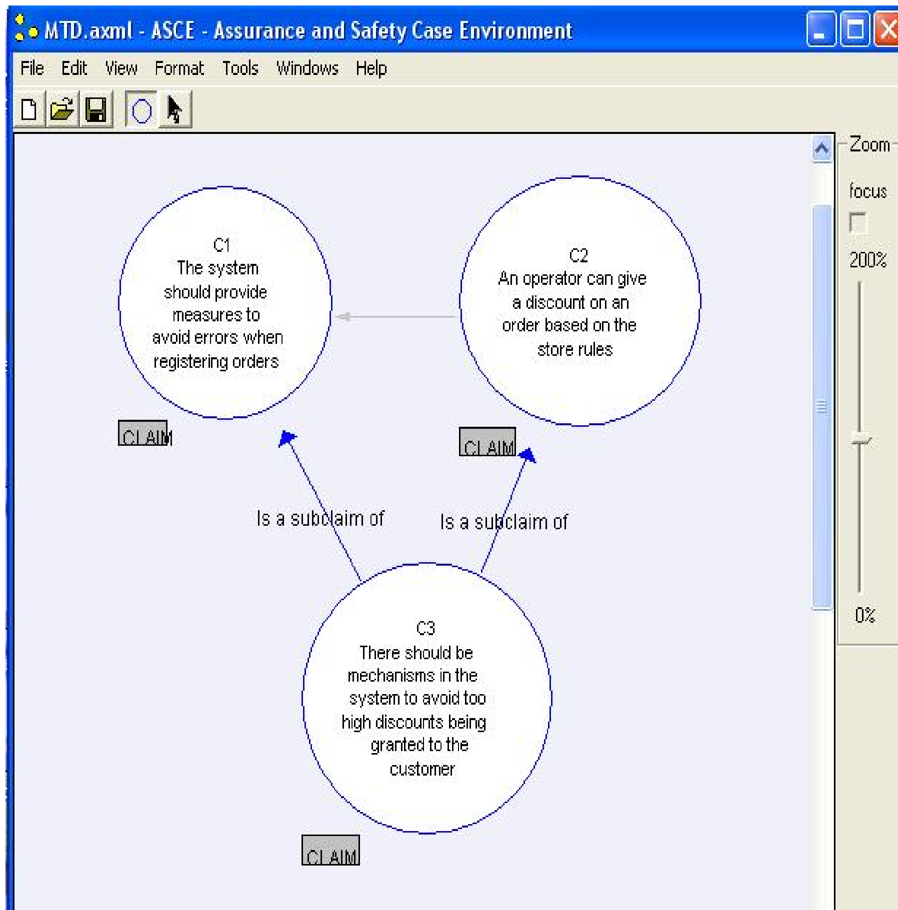

Figure A.3: The subclaim connected to the previous claims

we are presenting the testplan so far through our demonstration in a generic table, containing what we find to be the information that should be included in a testplan. Depending on testing strategies, project and product demands, and customer demands, more information may be added, or the existing information may be organized in a different way. In figure A.3 we have included the third claim as a subclaim of the two previous claims.

A testplan should include the person responsible for carrying out the test along with the person in charge of collecting the results. This helps us to make sure that all tests are carried out, and that the results are recorded in an appropriate manner. The testplan from the Inception phase will often be connected to several testplans from the Architectural phase as we increase the level of detail in the system design. In our case, the Inception phase contains only one system level test, that the system should help the operator to register

Component test for	C3			
Tester	'Person B'		Results manager	'Test leader'
Date planned	ddmm	Date conducted	ddmm	
Resources needed	??		Time needed	??
Steps	Done	Expected result	Actual result	Comment
1. Add correct order		Correct order is registered		
2. Add too high discount		Warning message, discount is not registered		
Approved	Yes/No	By		
Date	ddmm			

Table A.5: Testplan for Claim 3

orders correctly. There are many ways this could be tested and implemented, and so this test is difficult to carry out alone. In the Architectural phase, we have two more tests that are influencing the system level test. Table A.5 shows how we have been able to provide more details to the test plan.

With the information gained from the HAZOP and the FMECA (which we are not showing in this iteration), we decide to make some changes to the design of our system so that we may handle the discovered hazard. The store wants to keep their discount policy, but we add some restrictions to the system. The first restriction is that if an operator inserts a discount that exceeds 15%, a warning message pops up. The operator then has to confirm that she wants to give the entered discount. We expect this warning to reduce the possibility of entering a wrong discount. Erroneous discounts below 15% is considered to be part of the stores service, and the consequences not dire enough to add any more control functionality. The second addition to our design is that an operator are not able to give discounts exceeding 25%, but has to pass the order to a supervisor for approval. A warning message is not considered to be enough in this case, as we do not want the operator to be able to approve this discount at all. It is therefore decided that we build a barrier into the system, blocking all orders that are including a discount of

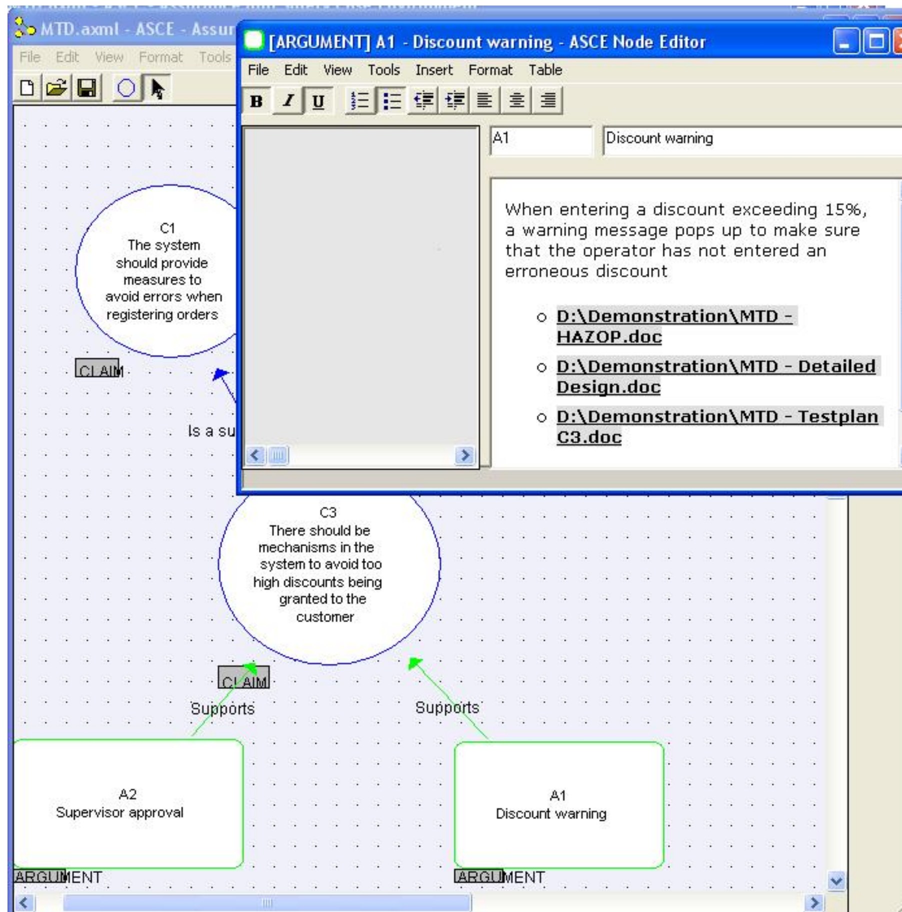

Figure A.4: The arguments connected to the appropriate claim

more than 25% and requesting the operator's supervisor for approval.

This new functionality is providing the first arguments in our example ASCE network. As can be seen in figure A.4, the two arguments are connected to the appropriate claim (actually a sub-claim in our example). We also see the links to the HAZOP and design document containing the reasons for developing the arguments. Finally, we have a link to the testplan that will provide us with the evidence needed for the arguments to satisfy the claim.

After we have changed the design, we do a new iteration of hazard analyzes to check that the changes really did remove the hazard, and that no new hazards have been introduced. It is still possible to enter erroneous orders, and to give wrong discounts, but we have blocked the possibility of giving high discounts by mistake, or without supervisor approval. In this iteration

System: MTD
Ref. drawing:

Conducted by: Ole-Johan
Date: 27.09

Page: 1 of 1

Unit description			**Hazard description**			**Failure effect**	
Ref.nr.	*Function*	*Operational state*	*Failure mode*	*Failure cause*	*Failure detection*	*On other units*	*On main unit*
01	Order registration	Accepting orders	Fail to accept order	Erroneous order		None	None
02	Order registration	Registering orders	Fail to register order	No database connection	Catching exception	None	System halt
03	Discount registration	Registering discount and recalculating sum	Registers wrong discount	Algorithm fail	Operator	None	None

Ref. nr	Failure rate	Failure effect rank	Failure reduction	Comment
01	10%	Negligible	Order registration routines	
02	2%	Critical	Connection control procedures	Can stem from multiple synchronous fails
03	1%	Critical	Testing	Discovery should lead to code review

Figure A.5: The FMECA table

we are showing the results of the FMECA in A.5, which is somewhat different than the HAZOP table[2]. Though we have not removed all hazards from the previous iteration, we have not introduced any new hazards. In many cases it is impossible to remove or even control all hazards, but this does not mean that we have to let the unresolved hazards run free. If we cannot prevent the hazard from taking place, we can prepare for the hazard state and minimize the consequences of the failure.

After the hazards are reduced to an acceptable level, a level that is decided by the stakeholders, we should be down to one or two design options. A choice of design at this point will be focused on the effort and resources needed to implement it in accordance with the quality of the final product.

[2]Because of the size of the table, we have split the table in two parts, repeating the reference numbers to make it easier to read.

In our example, we where able to prevent the hazard by introducing a barrier. As we do not have any other design proposals that could solve our problem in a better way, nor any need of removing the other hazards, we can now move on to the next phase of development.

A.4 Construction phase

As we have mentioned before, neither RUP nor ASCAD is much concerned about how the system is implemented. In our chapter on ASCAD we presented the implementation strategies that are advocated, but it is still the developers choice how to use them. What should be the developers biggest concern is that all requirements behind the design are implemented.

In our setup, the construction phase also includes the testing of the system. Again, we are not advocating any specific strategies, but to make the most out of the information in the safety cases, we recommend that both implementation and testing is conducted in a bottom-up manner. In addition to making it easier to track the work that has been done, this strategy will lessen the workload imposed on the developer team if any irregularities are discovered, especially during the implementation.

There are several strategies for implementing systems, and there are also different strategies for testing the system. The testplans can be the same, much of the difference being at what times they are run. RUP is advocating the strategy of developing software in a series of small iterations. This means that implementation will consist of coding a small part or a module, testing and verifying the code, and then correct the module until satisfying the tests.

As mentioned before, we do not have a design of our system and therefore nothing to implement. Assuming that we were able to implement our barrier correctly, the test is showing that our claim is satisfied. We add the evidence to the Claims-Arguments-Evidence network, linking to the final test record as shown in figure A.6. It may be tempting to just add a bubble stating that the test did verify the claim, but we should not give in to this temptation. After the system has been delivered to the customer, it is possible that an operator performs an operation in a manner that was not predicted. Upon receiving feedback that the system did not function as required, having the links to the test record will help us track down the error. If we discover a way of performing the operation that was not covered by the original test we update the test and run it anew. This run may uncover a flair in the design that can call for a change.

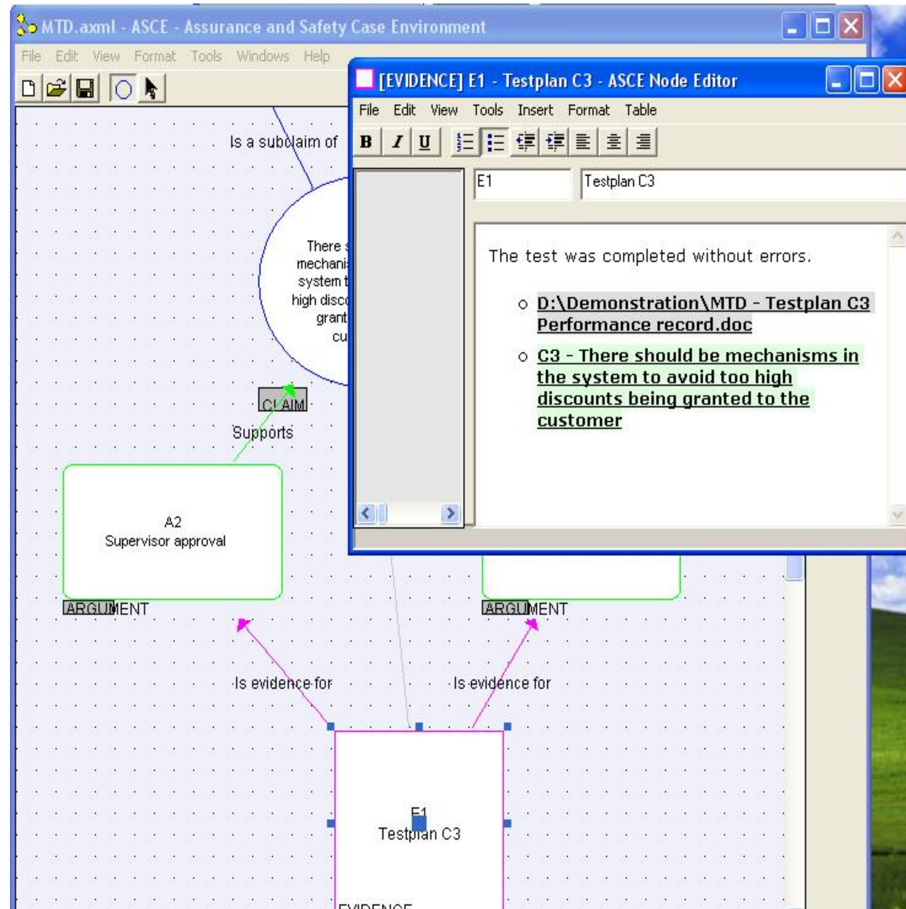

Figure A.6: We have completed one fork of the ASCE network

Completion of the implementation means that the system is ready for transfer to the customer, but it does not mean that the work on the system is finished. We still have one more phase to do. Before moving to the Transition phase we are including the ASCE network as it is after we have moved through all of the previous phases. Figure A.7 gives a simple and understandable overview of how we have satisfied the system requirement regarding discounts.

A.5 Transition phase

The transition phase is the last phase of our methodology, and may also be characterized as the longest. During the previous phases, some thought should have been given to how the system is to be delivered to the customer, what kind of training that is necessary to the users, and how the system is to be supported and maintained. The safety case is intended not to be archived when the system is delivered, but to follow the system throughout its lifetime. The claims in this safety case is mostly made up of maintenance requirements, along with support agreements. Even though the maintenance safety case is somewhat separate from the development safety case that we have just worked our way through, it should contain links to appropriate nodes and documents in the development safety case. We may also want to include links the other way. As explained in chapter 8.5, much of the information used in the Operation and Installation Safety Case phase is found in the other phases, but we need to find and organize the information.

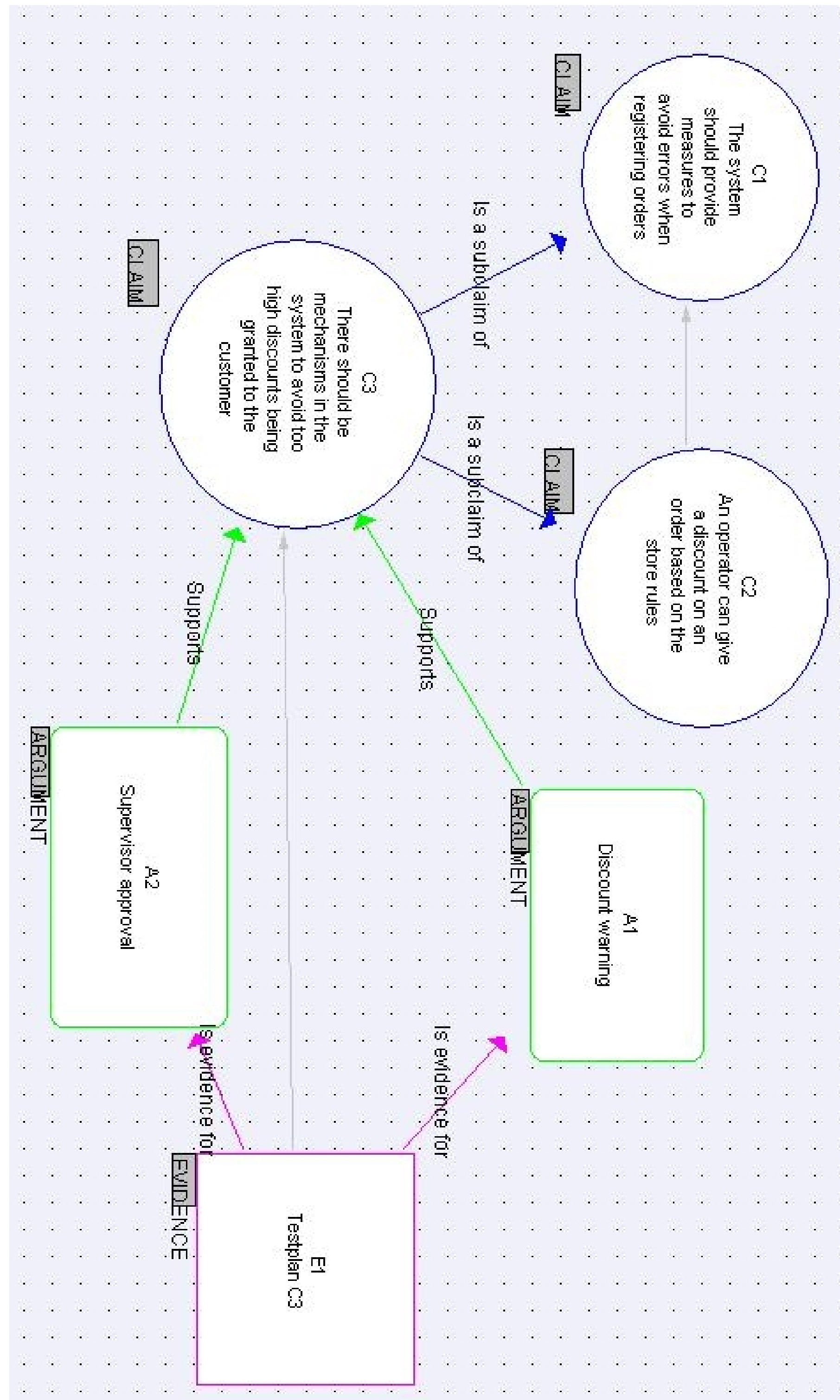

Figure A.7: The complete demonstration network

Appendix B

Definitions

Safety Case A documented body of evidence that provides a demonstrable and valid argument that a system is adequately safe for a given application and environment over its lifetime.

Failure Nonperformance or inability of system or component to perform its intended function for a specified time under specified environmental conditions.

Reliability The probability that an item will perform its required function in the specified manner over a given time period and under specified or assumed conditions.

Hazard A system state that, together with certain (other) conditions in the environment of the system will lead to an `Accident` (loss event).

Accident An undesired and unplanned (but not necessarily unexpected) event that results in (at least) a specified level of loss.

Hazard level A combination of severity (worst potential damage in case of an `Accident`) and likelihood of the occurrence of the `Hazard`.

Risk The `Hazard Level` combined with the likelihood of the `Hazard` leading to an `Accident` plus exposure (or duration) of the `Hazard`.

Safety Freedom from `Accidents` and losses. Will the system refrain from hurting people or destroying equipment and requirement? In this sense,`Safety` is somewhat like `Security`. It is not practical to achieve a goal of complete and total `Security`, nor is it necessarily possible to achieve complete and total `Safety`. We try to come as close to the ideal as possible. `See also Business-Safe`

Business-Safe A system seldom behaves in such a way that in causes the customer or his users to lose money or important information.

Security Denial of unauthorized access.

Fault An unplanned event that may lead to a `Failure`.

Business-critical Software that is `Business-critical` will probably lead to severe losses that may set a company out of play if a `Failure` occurs.

Intention The idea behind a requirement. The reason for this requirement being included in the requirement specification.

Means-Ends Hierarchies A hieratical view where each level represents a different model of the same system. The information at a level acts as the goals (the ends) for the model at the next, lower level (the means). I.e. the current level specifies *what*, the level below *how*, and the level above *why*. More about Means-Ends Hierarchies can be found in [12].

Bibliography

[1] Adelard. http://www.adelard.co.uk.

[2] Jon Arvid Børretzen, Tor Stålhane, Torgrim Lauritsen, and Per Trygve Myhrer. Safety activities during early software project phases.

[3] BUCS. http://www.idi.ntnu.no/grupper/su/bucs.html.

[4] ASCAD by Adelard. http://www.adelard.co.uk/resources/ascad/index.htm.

[5] Safeware Engineering Corporation. http://safeware-eng.com.

[6] EuroSPI. http://2005.eurospi.net.

[7] Morris Chudleigh Felix Redmill and James Catmur. *System Safety: HAZOP and Software HAZOP*. Wiley, 1999.

[8] Martin Fowler. *UML Distilled Second Edition*. Addison-Wesley, 2000.

[9] Philippe Kruchten. *The Rational Unified Process An Introduction*. Addison-Wesley, second edition, 2000.

[10] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. http://www.safeware-eng.com/publications/IntSpec.pdf.

[11] NEA. http://www.nea.fr/abs/html/nesc0653.html.

[12] Jens Rasmussen. *Information Processing and Human-Machine Interaction: An Approach to Cognitive Engineering*. North Holland, 1986.

[13] Marvin Rausand. *Risikoanalyse*. Tapir Forlag, 1991.

[14] SafeComp. http://www.hrp.no/safecomp2005.

[15] Tor Stålhane. Analyse av systemsikkerhet - Forelesning.

[16] Tor Stålhane and Torgrim Lauritsen. Safety Methods in Software Process Improvement. *EuroSPI 05*, 2005.

[17] Tor Stålhane, Per Trygve Myhrer, Torgrim Lauritsen, and Jon Arvid Børretzen. BUCS rapport - Intervju med utvalgte norske bedrifter omkring utvikling av forretningskritiske systemer. 2003.

[18] Tor Stålhane, Per Trygve Myhrer, Torgrim Lauritsen, and Jon Arvid Børretzen. BUCS report - An Overview over RUP and Some Important Safety Analysis Methods. 2004.

[19] Hans Van Vliet. *Software Engineering - Principles and Practice.* Wiley, second edition, 2002.

[20] Wikipedia. http://en.wikipedia.org/wiki/Rational_Unified_Process.