Towards a Unified Model-Based Formalism for Supporting Safety Assessment activities

by

Fredrik Forssén

LIU-IDA/LITH-EX-A--09/051--SE

2010-02-17
Towards a Unified Model-Based Formalism for Supporting Safety Assessment activities

by

Fredrik Forssén

LIU-IDA/LITH-EX-A--09/051--SE

2010-02-17
Abstract
Safety assessment is a rational and systematic process for assessing the risk associated with the usage of a product. While the safety assessment process is important even when making a simple product, the true importance of this process comes into light when designing for example an aircraft, where a failure could possibly lead to the loss of human lives. However, even though this process is vital for certain industries, it is plagued by a lack of tools. The existing tools are focused on specific parts of the process and do not make use of work done in earlier steps of the process which often means that the safety engineer needs to manually do work that could have been calculated automatically from information that is already present from an earlier step in the process.

This thesis shows that by creating a model of the product that can be present and augmented throughout every step in the process, many calculations that are currently done by hand can be automated or semi-automated by examining this shared model. The thesis proposes a specification for a modeling formalism that is simple enough to be used as early as the requirements phase of a project, but powerful enough to provide important information all the way throughout the safety assessment process.

The thesis also specifically shows how this model can be used to help in the creation and updating process of Failure Mode and Effects Analysis (FMEA) documents as a proof-of-concept implementation based on Sörman Information AB’s product “Uptime BPC Standard”. Algorithms for synchronizing between the model and the FMEA representation, as well as algorithms for automatically calculating the next level effect and global level effect of failure modes based on the hierarchy and connections made in the model are also presented.

The prototype implementation shows that even though the entire safety assessment process cannot be automated it is possible to extract information from the model by analyzing its hierarchy and connections. While more work still needs to be done before the entire safety assessment process can be encompassed, the initial results shows that the proposed modeling formalism allows us to create models from which relevant information that can be used to support the safety assessment process can be calculated.
Acknowledgement

I acknowledge nothing.

Fredrik Forssén
Linköping, February 2010
Table of contents

1 INTRODUCTION ..................................................................................................................... 1
  1.1 BACKGROUND .................................................................................................................. 1
  1.2 PURPOSE .......................................................................................................................... 3
  1.3 OBJECTIVE ..................................................................................................................... 3
  1.4 LIMITATIONS .................................................................................................................. 3
  1.5 THESIS OUTLINE ........................................................................................................... 3

2 FAILURE MODES AND EFFECTS ANALYSIS (FMEA) .......................................................... 3
  2.1 MISSION PROFILES ......................................................................................................... 6
  2.2 FMEA EXAMPLE ............................................................................................................ 6

3 UPTIME PLATFORM ............................................................................................................... 10
  3.1 BPC .................................................................................................................................. 10
  3.2 UPTIME LIB ................................................................................................................... 11
  3.2.1 Data relations ............................................................................................................ 12
  3.3 OTHER COMPONENTS/LIBRARIES USED ..................................................................... 14

4 A BRIEF BACKGROUND TO MODELING .............................................................................. 14

5 THE PROPOSED MODELING FORMALISM ............................................................................ 15
  5.1 CREATING A HIERARCHY ............................................................................................... 16
    5.1.1 The first steps towards a working hierarchy ......................................................... 16
    5.1.2 Supporting black-box design of systems ............................................................. 18
    5.1.3 Putting it all together ............................................................................................ 20
  5.2 CONNECTION SEMANTICS ............................................................................................... 23
    5.2.1 Internal connections of a black box function ....................................................... 23
    5.2.2 Redundancy ........................................................................................................... 23
    5.2.3 Connecting failure modes .................................................................................... 25
  5.3 EXTRACTING USEFUL INFORMATION FROM THE MODEL ............................................ 29
    5.3.1 Calculating the global effect ................................................................................ 29
    5.3.2 Calculating next level effect ................................................................................ 31
  5.4 SUMMARY ....................................................................................................................... 45

6 DOMAIN MODEL .................................................................................................................... 45
  6.1 TRANSFORMING THE MODEL HIERARCHY TO FIT THE FMEA DOMAIN ................. 45
    6.1.1 Keeping reuse in mind ............................................................................................ 48
  6.2 ADDING MORE RELATIONS TO THE GRAPH ................................................................ 51
    6.2.1 Affected functions ................................................................................................. 51
    6.2.2 Next level effect ...................................................................................................... 52
    6.2.3 Manufacturers ........................................................................................................ 52
    6.2.4 Phrases .................................................................................................................... 52
    6.2.5 Ports and connections ........................................................................................... 52
    6.2.6 Mission profile/phase objects ............................................................................... 52
    6.2.7 FMEA and Model objects ..................................................................................... 53
  6.3 THE FINAL DATA MODEL ............................................................................................... 53

7 IMPLEMENTATION .................................................................................................................. 54
  7.1 MODEL CLASS HIERARCHY ............................................................................................. 54
    7.1.1 Connections ............................................................................................................ 54
  7.2 FMEA MODULE ............................................................................................................. 55
    7.2.1 Grid configurations ............................................................................................... 56
    7.2.2 Importing Excel FMEA’s ..................................................................................... 57
    7.2.3 Standard phrases ................................................................................................. 61
    7.2.4 PDF Generation .................................................................................................... 61
  7.3 CONNECTING THE MODEL AND THE FMEA .................................................................. 61
    7.3.1 Relating the FMEA and the Model system trees ................................................... 61
    7.3.2 Tree synchronization ............................................................................................. 62
Table of figures

**FIGURE 1:** The ARP4761 Safety Assessment Diagram (picture taken from [1]) .................................................. 2
**FIGURE 2:** FMEA Process Flow (picture taken from [3]) .................................................................................. 5
**FIGURE 3:** Uptime Studio, an environment for creating and editing after-sales documentation ........ 10
**FIGURE 4:** Uptime Web Viewer, a web interface working with the same data that was entered in

  Figure 3 .................................................................................................................................................. 11
**FIGURE 5:** Parent-child relation ................................................................................................................. 12
**FIGURE 6:** Super-subobject relation .......................................................................................................... 12
**FIGURE 7:** Simple reuse example .............................................................................................................. 13
**FIGURE 8:** Link relations .......................................................................................................................... 13
**FIGURE 9:** Object field relation ................................................................................................................ 14
**FIGURE 10:** A small part of a NASA satellite modeled in the tool RODON .............................................. 15
**FIGURE 11:** Component hierarchy example ............................................................................................... 16
**FIGURE 12:** Parent-child hierarchy ........................................................................................................... 17
**FIGURE 13:** Function hierarchy example .................................................................................................. 19
**FIGURE 14:** Revised parent-child hierarchy .............................................................................................. 20
**FIGURE 15:** Model specification in MetaGME ........................................................................................... 22
**FIGURE 16:** Black-box functions are considered to be maximally connected inside ................................ 23
**FIGURE 17:** Modeling redundancy with 3 functions ................................................................................... 24
**FIGURE 18:** An easy pitfall, three components connected this way does not provide redundancy ....... 24
**FIGURE 19:** Modeling redundancy between components ........................................................................ 25
**FIGURE 20:** Failure mode model example ................................................................................................ 26
**FIGURE 21:** Failure modes on a serial connection component example ................................................... 26
**FIGURE 22:** Failure modes can be connected to one or more ports of a function ................................... 27
**FIGURE 23:** Making an OR-relation to a failure mode ................................................................................. 28
**FIGURE 24:** If want to make failure modes that are detailed to certain component ports, we can

  connect the failure mode to these ports by use of observation ports ....................................................... 29
**FIGURE 25:** Global effect example .............................................................................................................. 30
**FIGURE 26:** First two steps in setting the global effect on the tree given in Figure 25 .............................. 31
**FIGURE 27:** An example model for calculating next level effects on several components in a series

  ............................................................................................................................................................... 32
**FIGURE 28:** The paths taken of the next level effect calculation algorithm in the model seen in

  Figure 27 ........................................................................................................................................... 33
**FIGURE 29:** An example model for calculating the next level effect of several components in parallel

  ............................................................................................................................................................... 35
**FIGURE 30:** The paths taken of the next level effect calculation algorithm in the actuator model

  in Figure 29 ........................................................................................................................................... 36
**FIGURE 31:** A slightly more complicated model to calculate the next level effect of ............................. 37
**FIGURE 32:** Visualization of how the algorithm calculates the next level effect on the model in

  Figure 31 ........................................................................................................................................... 38
**FIGURE 33:** A more complicated model that features loops, for a clearer view of the actual loops,

  see Figure 34 ........................................................................................................................................ 42
**FIGURE 34:** The connections in the model shown in Figure 33 drawn as a directed cyclic graph, green

  nodes are function ports, yellow nodes are component ports and gray nodes are the ports of the top system ..... 43
**FIGURE 35:** Figure 34 with the paths that the algorithm takes drawn in it ................................................ 44
**FIGURE 36:** SAE ARP4791 [1] Functional FMEA Worksheet Example .................................................... 46
**FIGURE 37:** Flattening the function hierarchy is simply a matter of moving each function upwards

  to its nearest subsystem parent ................................................................................................................. 47
**FIGURE 38:** Moving components/subsystems that are children of functions to their nearest

  subsystem parent is a similarly simple transformation ............................................................................ 47
1 Introduction

1.1 Background

Engineering systems grow more complicated every year. Moore’s law states that the number of transistors that can be placed on an integrated circuit would double approximately every two years. As engineering systems grow more complicated it gets harder to predict what happens when one or more of its components fail. Almost every system under deployment is exposed to component failures or is under the risk of suffering a major breakdown under its lifetime. This means that today it is even more important to be able to assess that the systems that are produced can fail safely, especially when human lives may depend on them.

Safety assessment is a rational and systematic process for assessing the risk associated with the usage of a product. A safety assessment process helps safety engineers to answer the following questions:

- What might go wrong? This question is related to the identification of hazards (list of all potential accident scenarios and the potential outcomes and effects).
- How likely is it that a particular accident happens and how severe will the effects be if it happens? The risk factors need to be evaluated in terms of effects and likelihood.
- Can the situation can be eliminated or improved? Once the risks have been identified and their effect and likelihood quantified what are the regulatory measures that can be taken to control and reduce the identified risk?
- What would be the cost for the regulatory action and how it would improve the situation? Cost–effectiveness is an important part of the safety assessment process as for any product development process. A risk control option should be economically feasible.
- What are the actions that need to be taken? Recommendations for decision-making need to be elaborated based on the previously answered questions.

Various safety standards has been put into use to address this problem, some of the most widely used are named in the list below:

- **RTCA/DO-178** - "Software Considerations in Airborne Systems and Equipment Certification" [13]
- **Def Stan 00-56** - Safety Management Requirements for Defense Systems
- **+SAFE** - A Safety Extension to CMMI [14]
- **SAE ARP 4754** - "Certification Considerations for Highly-Integrated or Complex Aircraft Systems" [10]
- **SAE ARP 4761** - "Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment" [1]
- **Def(Aust) 5679** - "The Procurement Of Computer-based Safety Critical Systems" [16]

The standards detail how to integrate the safety life cycle into the product life cycle. However all of them will require some customization in order to make it fit with the current workflow of the organization, with the provision that the prescribed safety requirements are still met. Thus even if two organizations use the same standard, they will - in general - not use it in the exact same way.
The SAE ARP4761 safety assessment diagram is shown in Figure 1 below. This diagram details the safety assessment life cycle of a product according to this standard. As can be seen in this figure, the safety assessment will generally go through several main stages (Functional Hazard Analysis (FHA), preliminary Fault Tree Analysis (Prelim FTA), Common Cause Analysis (CCA) and so on) to derive the safety requirements before the engineering process takes over to implement the system. During the initial phases a limited amount of functional information about the system would be required, while later during the implementation and verification phases more detailed information would be available (and required).

### Figure 1: The ARP4761 Safety Assessment Diagram (picture taken from [1])

While there already is software on the market for supporting every aspect of the safety assessment process, these tools lack one certain critical aspect. Information is not reused properly. There is a lack of a clearly defined interface between the tools used in the different aspects of the process. Furthermore, much of the information entered could be calculated automatically or semi-automatically from a model, eliminating potential human error. However, generation of Failure Mode and Effect Analysis (FMEA) and FTA from a model using modeling tools that are currently on the market requires that the safety assessment work is performed by a bottom-up approach instead of the top-down approach that for example SAE ARP 4761 suggests.

This thesis proposes an environment which supports both the top-down and the bottom-up approach. In the top-down approach safety assessment aspects like FMEA and FTA are performed manually and a model is semi-automatically generated from the information entered. In the bottom-up approach we could, from the model, automatically derive aspects of the safety assessment process. By keeping the model updated throughout every step of the safety assessment process this thesis proposes that we gain the following important advantages:

- The model becomes a clearly defined interface to exchange information between the different aspects of the safety assessment process, making it significantly easier to utilize earlier calculated information to the fullest extent thus saving time.
• Design errors have a greater chance to be identified and corrected in an early phase, thus potentially saving great amounts of time and money.
• By providing a clearly defined interface between the different aspects we can in an easier way see what happens to the entire system when changes to a single subsystem are introduced, thus encouraging testing out alternate designs by making it easier to see the results of such a change.

1.2 Purpose
The purpose of this thesis is to provide the specification of a modeling formalism that can be used to generate models that can help supporting the entire safety assessment process.

1.3 Objective
The goal of this thesis is to produce a modeling formalism for the safety assessment process and develop a prototype application for working with this formalism based on an existing system for after-sales information processing, called UpTime. For an overview of the UpTime system, see Chapter 3.

1.4 Limitations
As can be seen in Figure 1, the safety assessment process contains many interconnected steps. This thesis addresses only one of the aspects of the process, namely the creation of an FMEA from a functional or component structural model.

1.5 Thesis outline
The rest of the thesis is organized as follows:
Chapter 2 - Failure Modes and Effects Analysis (FMEA) (page 3), contains a brief overview of the Failure Modes and Effects Analysis (FMEA) process.
Chapter 3 - UpTime platform (page 10), contains an overview of the UpTime platform that was used as a base to develop the application that this thesis describes.
Chapter 4 - A Brief Background to Modeling (page 14), gives a brief overview of modeling.
Chapter 5 - The Proposed Modeling Formalism (page 15), presents a specification for the modeling formalism presented in this thesis together with some examples of how to use it and also shows how to calculate some interesting data from models created according to this formalism.
Chapter 6 - Domain model (page 45), presents the translation of the modeling specification into a domain data model for the UpTime platform and discusses design decisions and previous iterations of the domain model.
Chapter 7 - Implementation (page 54), presents part of the software architecture of the implementation of the project.
Chapter 8 - Conclusion (page 65), summarizes the conclusions of the thesis and presents the problems that are left unanswered and the future work.

2 Failure Modes and Effects Analysis (FMEA)
Failure Modes and Effects Analysis (henceforth referred to as FMEA) is a systematic method of predicting and analyzing possible faults within a system. It is widely used in the
manufacturing industries in various phases of the product life cycle. Section 2.2, FMEA Example, gives a very simple example of how to construct an FMEA by examining a regular bicycle. For an exhaustive overview of the entire process the reader is encouraged to also read The Basics of FMEA [9] or SAE ARP4761, Aerospace Recommended Practice [1].

Failure Mode and Effects Analysis is, as the name states, about analyzing the effects that certain “failure modes” has on a system. A failure mode can be said to be the manner in which a fault occurs [2]. A fault, in turn, is an inability for an object to work in the desired manner. For example, for a normal switch a potential failure mode could be “switch partially open”. Another example of a failure mode could be “Bulb broken” for a light bulb.

Typically, when working with an FMEA, the safety engineer starts with a hierarchical view of the system and then considers the failure modes of each component and subsystem. He records what happens on each level if the failure mode is active (the local effect), what happens one hierarchy level up when that failure mode becomes active (next level effect), and what happens on the system-wide level (the global effect). After this the safety engineer then works with a spreadsheet, which has a set of columns that are specified by the specific FMEA standard, and fills in the spreadsheet with the correct data. Normal columns include, apart from the earlier mentioned local/next level/global effect, integer values from 1-10 for rating the failure modes severity, occurrence and detection value, and a textual description of the cause of the failure mode and several other columns depending on the used FMEA standard. Please note however that even these columns can be different and depends on the used FMEA standard.

An FMEA can be performed at any level of the system at any time during the development. The FMEA may be either quantitative or qualitative (if a quantitative FMEA is being performed, a failure rate is determined for each failure mode) and it may be performed on all types of systems. The two basic types of FMEA’s that this thesis will study are the “Functional FMEA” and the “Piece-part FMEA”. These are usually performed for different purposes; the functional kind is usually used for top-down analysis while the piece-part kind is often used for bottom-up analysis [4].

The functional FMEA concerns itself with breaking down the system into functional blocks and identifying the failure modes for each of these blocks. For example, a power supply circuitry could be called a functional block, and an example of a functional failure mode could be “Short to ground”.

A piece-part FMEA, on the other hand, is similar to a functional FMEA apart from the fact that instead of analyzing at the functional block level, the failure modes of each component performing these functions are analyzed instead.

One potential drawback with an FMEA is that only single faults are examined. For example, an FMEA takes into account what happens if the failure mode “Tire blown” is active or if the failure mode “Brake system defective” is active, but it does not take into account what happens if both of these failure modes are active at the same time. In order to be able to examine this another method will have to be used, for example Fault Tree Analysis (FTA).

No matter if we are making a functional or a piece-part FMEA; we usually follow a work process flow that is very similar. This workflow is summarized in Figure 2 below.
Figure 2: FMEA Process flow (picture taken from [3])

The process flow seen in Figure 2 above will be detailed in section 2.2 by a simple example of how to perform an FMEA.

Apart from the functional and piece-part FMEA’s an FMECA (Failure Mode and Effects Criticality Analysis) can also be created. This is performed in order to evaluate reliability and safety by identifying critical failure modes on and their effects on the system. It can also be performed on parts that are especially critical to the systems function (and wellbeing of the people using the system). An FMECA is, basically, an FMEA with an added criticality analysis. An additional section is added to the FMEA table that is filled in with the
information specific to the criticality analysis. To calculate the criticality data it is necessary to have failure data and knowledge of the complete system.

2.1 Mission profiles

An FMEA often contains a reference to a mission profile. A mission profile is a collection of mission phases that captures the typical usage of the product. This could be an example of a mission profile for a passenger aircraft that will mostly reside in Reykjavik, Iceland.

<table>
<thead>
<tr>
<th>Mission Profile: Passenger Aircraft in Reykjavik</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mission Phase</strong></td>
</tr>
<tr>
<td>Takeoff</td>
</tr>
<tr>
<td>Cruising</td>
</tr>
<tr>
<td>Landing</td>
</tr>
</tbody>
</table>

While the mission profile might look similar for a passenger aircraft that will mostly reside in Alexandria, Egypt, the mission profile still tells us vital information. In Egypt there is plenty of sand and other residue in the air and the average temperature is very high, something that we will need to consider when doing our FMEA, while in Reykjavik the temperature is much lower and there is much less sand and similar residue in the air. Thus when we create an FMEA we must always start with considering the mission profile.

The mission phases that the mission profile consists of are also important to consider. For example, consider the “Passenger Aircraft in Reykjavik” profile given earlier. If the aircraft loses its radar during takeoff or landing, this is not as severe as if it does so while cruising. Thus the severity of the “radar not working”-failure mode is lower during the landing and takeoff phases than during the cruising phase.

The information from the mission profile can also be used during other parts of the safety assessment process, such as for example reliability prediction.

2.2 FMEA Example

This section shows how to construct an FMEA for a bicycle. The bicycle considered in our example handbrakes and gears. In order to keep the example simple, this example will only concentrate on the braking system of the bicycle. An exhaustive FMEA of every single conceivable failure mode will not be constructed in this example; it will instead focus on one simple failure mode “Loss of braking system”.

According to the process specified in Figure 2 safety engineer in charge of the FMEA process should start by identifying the targets to be protected, in order to keep this simple the example will only concern a single target, the user of the bicycle. In the next step the safety engineer defines that this example will only concern the mission phase “Travelling in heavy traffic”. The mission phase will affect some of the ratings during the FMEA process (for example, it is more severe if the braking system stops working in heavy traffic than if it stops working when the bicycle is parked in the garage). The mission profile will then consist only of this single mission phase. When the mission profile is defined the next step is attempting to identify the ways that the bicycle can fail (the bicycles failure modes).

At the top system level the failure mode “Loss of braking system” can be identified. The local effect for this will be “User cannot brake”, since the top system does not have a next level the next level effect is left blank, and the global effect becomes “User cannot brake”. Since the
defined mission phase is “Travelling in heavy traffic” it’s easy to see that the loss of the braking system would be catastrophic, and thus the severity rating of this failure mode should be 10. Detecting that this has happened is, however, quite easy and the detection rating should be 1. At this point the safety engineer cannot determine what the occurrence value for this failure mode should be, and thus this is left blank until more data has been examined.

The bicycle has a braking system, which consists of a wheel brake. These components are noted in the FMEA table as well. The wheel brake slows down the bike if the user pedals backwards. The wheel brake has a single failure mode “Skipped chain prevents user from applying brake”. This might happen if the chain skips and gets caught in a way that prevents the user to pedal backwards and thus apply the wheel brake. If this happens the local effect will be “Cannot apply wheel brake”, the next level effect (on the entire braking system) will be “Loss of brakes” and the global effect will become “User cannot brake”. At this level it is pretty easy to see that the loss of wheel brakes in heavy traffic will be very severe, but easy to detect. The severity rating becomes 10 and the detection rating becomes 1 by the same reasoning as earlier. Based on his personal experience, the safety engineer can approximate that during a bicycles lifetime this happens at least a couple of times, this leads to the occurrence rating becoming 10 as well since it’s almost a certainty that this will happen at least once with each bicycle. Since this failure mode now has a severity, detection and occurrence rating the Risk Priority Number (RPN) can be calculated. This is calculated by multiplying the severity, detection and occurrence values. Performing this calculation gives this failure mode an RPN of 100.

Looking at the braking system, it’s easy to see that since the bicycle only has the wheel brake the braking system actually is the wheel brake. The braking system thus gets copies of the values from the wheel brake. At this point it is also possible to fill in the occurrence values on the “Loss of braking system” failure mode on the bicycle system. Some simple reasoning will suffice to notice that the occurrence rating should be a 10. The final FMEA is shown in Table 1 below.

At this point, the FMEA process requires the safety engineer to look over the table and make sure that all values are within acceptable limits. There are not any preset limits, and thus it is up to the safety engineer to make a judgment call on each failure mode based on previous experience. In the case of the bicycle a safety engineer might, for instance, judge that the risk of losing brakes in high traffic is not acceptable. In this case, some countermeasures needs to be developed. After thinking about it the safety engineer might decide that a redundant brake system that will not get affected if the wheel brake fails is a good solution and after looking at the technologies available the engineer might choose to add a handbrake to the bicycle. Of course, since the system has now changed, the risks need to be re-evaluated and thus the FMEA process makes its first iteration.

First the handbrake is added to the braking system and the failure modes of the wheel brake are identified. The handbrake has a single failure mode “Worn down brake pads”, which would cause the handbrake to become unresponsive. This would be very severe, an 8 would represent that the primary function of the handbrake is lost or seriously degraded but that it will most likely not result in a safety issue, since if this is the case it is assumed that the wheel brakes still work and thus the user of the bicycle does not lose all of his/her braking capabilities. Again, based on previous experience, the safety engineer can be almost sure that this will happen a couple of times during the bicycles lifetime which leads to an occurrence rating of 10, but it will be very easy to detect which leads to a detection rating of 1. This leads
to an RPN of 80. At this point the safety engineer can easily see that the countermeasure “Advice user to change brake pads once per year” is both easy and inexpensive to implement, and thus it should be noted immediately in the countermeasures column of the FMEA table.

Since the bicycle now has a backup brake, the 10 in the severity column for the wheel brake can be downgraded to an 8 (with the same argument as for the handbrake). And the occurrence value of the “Loss of braking system” failure mode of the braking system can be lowered from a 10 to a 4 (with the motivation that it is much less likely that both the wheel brake and the handbrake would fail at the same time). The same argument can be used to downgrade the 10 in the occurrence column of the failure mode “Loss of braking system” for the bicycle to a 4. This in turn changes both of these failure modes RPN from 100 to 40, which might be a value that the safety engineer performing the FMEA considered more acceptable.

But, what happens if only one of the brakes fail? This should be examined as well to make sure that everything is alright. The failure mode “Braking system degraded” is added to the braking system, and its local effect becomes “User can only brake with either the handbrake or the wheel brake, not both”. This gets the severity value 3, which not so severe (but this is still a partial loss of function). The occurrence value becomes a 10 (since it occurs if either the wheel brake or the hand brake fails, which in turn has an occurrence value of 10 from before), and the detection value becomes 1, since it is very easy to detect. The RPN for “Braking system degraded” becomes 30, which the safety engineer should most likely consider as acceptable. At this point the next level effects for the wheel brake and the handbrakes single failure modes should be changed to this new failure mode instead.

At the top level (Bicycle) a failure mode called “Braking system degraded” should also be added (since this affects the entire bicycle), after some careful examination it can be seen that this failure mode becomes an almost identical copy (sans the local and next level effects) of the failure mode of the braking system with the same name. At this point the global effect columns should be updated so that the correct global effect is shown on all failure modes (by following the next level effects to the top) and then the safety engineer should again carefully examine the new FMEA (which can be seen in Table 2 below). After reviewing the table the safety engineer might as this point judge that the risks are acceptable, and thus this FMEA is finished.
<table>
<thead>
<tr>
<th>Component</th>
<th>Function</th>
<th>Failure Mode</th>
<th>Local Effect</th>
<th>Next level effect</th>
<th>Global effect</th>
<th>Severity</th>
<th>Occurrence</th>
<th>Detection</th>
<th>RPN</th>
<th>Countermeasure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicycle</td>
<td>Braking</td>
<td>Loss of braking system</td>
<td>User cannot brake</td>
<td>N/A</td>
<td>User cannot brake</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Braking System</td>
<td>Braking</td>
<td>Loss of braking system</td>
<td>Braking system lost</td>
<td>Loss of braking system</td>
<td>User cannot brake</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td>Introduce redundant brake system</td>
</tr>
<tr>
<td>Wheel brake</td>
<td>Braking</td>
<td>Skipped chain preventing user from applying wheel break.</td>
<td>Cannot apply wheel brake</td>
<td>Loss of braking system</td>
<td>User cannot brake</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: First FMEA example table

<table>
<thead>
<tr>
<th>Component</th>
<th>Function</th>
<th>Failure Mode</th>
<th>Local Effect</th>
<th>Next level effect</th>
<th>Global effect</th>
<th>Severity</th>
<th>Occurrence</th>
<th>Detection</th>
<th>RPN</th>
<th>Countermeasure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicycle</td>
<td>Braking</td>
<td>Loss of braking system</td>
<td>User can only brake with either the handbrake or the wheel brake, not both.</td>
<td>N/A</td>
<td>User can only brake with either the handbrake or the wheel brake, not both.</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Braking System</td>
<td>Braking</td>
<td>Loss of braking system</td>
<td>Braking system lost</td>
<td>Loss of braking system</td>
<td>User cannot brake</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Braking System</td>
<td>Braking</td>
<td>Loss of braking system</td>
<td>Braking system lost</td>
<td>Loss of braking system</td>
<td>User cannot brake</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Wheel brake</td>
<td>Braking</td>
<td>Skipped chain preventing user from applying wheel brake.</td>
<td>Cannot apply wheel brake</td>
<td>Braking system degraded</td>
<td>User can only brake with either the handbrake or the wheel brake, not both.</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>80</td>
<td>Advice user to change brake pads once per year</td>
</tr>
<tr>
<td>Handbrake</td>
<td>Braking</td>
<td>Worn down brake pads</td>
<td>Applying handbrake has no effect</td>
<td>Braking system degraded</td>
<td>User can only brake with either the handbrake or the wheel brake, not both.</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: FMEA Example after introducing the countermeasure
3 UpTime platform

UpTime is a platform for creating windows applications for after-sales information processing. This platform is configured (tailor-made) for the customer’s needs using a combination of C# programming and XML configuration files. The UpTime platform consists of the following components:

1. A database
2. A database access library (UptimeLib)
3. A UI framework
4. A set of reusable UI components
5. A report generation language (UXL)

For the application developed in the context of this thesis the report generation language (UXL) has not been used.

3.1 BPC

Uptime BPC (Best Practice Configuration) is originally a program made for handling and creating technical documentation. It is a commercial application developed in-house by Uptime Solutions AB, and consists of the actual application and an oracle database. Uptime BPC is a configurable platform. Adding new features to it is done by specifying “pads” in XML that then uses reflection to create instances of C# classes and showing them in the BPC.
program. Uptime BPC also contains tools for user permission handling, language handling, translation, versioning and several other useful features for helping with coordinating a work process containing several people.

Uptime BPC can generate the technical documentation in several formats. The workflow usually consists of entering the information via the Uptime Studio application (as can be seen in Figure 3 above) and then generating publications that is published in one of several ways (for example, exported to PDF or exported to web where it can be viewed via the Uptime web viewer that can be seen in Figure 4).

Uptime BPC puts much focus on reuse of information and has good support for reusing objects (such as, for example, images, phrases and entire modules containing a mix of text, images and other objects). It also contains a system for tagging these data objects with a profile and then provides functionality for filtering what is shown based on these profiles. Combining all of these features produces a very powerful platform where complicated documents can be created and maintained with a relative ease of use.

![Figure 4: Uptime Web Viewer, a web interface working with the same data that was entered in Figure 3](image)

### 3.2 UptimeLib

All Uptime installations use a database with a given schema which can represent the customers’ data as objects. Retrieving and storing data in the database is done through the UptimeLib. UptimeLib works with filters that can be defined in C# code or in XML. Filters can be combined to make more complicated filters. It abstracts away all database code and introduces a way to work with objects in the database by using filters and operations.

Using UptimeLib means that you need to conform to using the predefined relations that are in UptimeLib. While UptimeLib is an object-database, it does not support inheritance between database types and thus cannot be called an object-oriented-database. Uptime is often used to store information in tree-like structures where the structure is separated from the information (the structure is saved in what is called a “superobject” while the information, or metadata, is
stored in what is called a “subobject”). UptimeLib contains plenty of operations for working with trees structured like this in an easy, high-level way.

### 3.2.1 Data relations

UptimeLib defines four main ways of relating data with each other.

#### 3.2.1.1 Parent-child

![Parent-child relation](image)

Figure 5: Parent-child relation

The parent-child relation is often the easiest relation to work with in Uptime. Each object must have one, and only one, parent and operations exists for finding all children of a certain object, the parent of a certain object, an entire tree of these parent child relations and so on.

#### 3.2.1.2 Super/subobject

![Super-subobject relation](image)

Figure 6: Super-subobject relation

Super and subobject does not, as the name might imply, have anything to do with inheritance. Instead, this relation allows us to create proxy-objects (the superobjects) of an object (the subobject). Each subobject can have one or more superobjects. This provides a way of implementing reuse of data by having several superobjects that shares the same subobject that each have a different parent. So in Figure 6 the three superobjects depicted could all have different parents, which would provide at least three different paths that could be used for finding the subobject. An example of a normal implementation of reuse of data can be seen in Figure 7 below.
In Figure 7 the subobject is reused by creating two different superobjects of it and then making each of these superobjects into children of each of the parents. By constructing a structure that looks like this both parent 1 and parent 2 can reach the same subobject by going through their child superobject. By constructing the hierarchy this way we have separated structure from content and can change each independently of the other (the superobjects represents the structure, and the subobject represents the content).

3.2.1.3 LinkSource/Target

Apart from the more structural parent-child and super-subobject relations there are also the possibility of linking between objects. An object can contain one or more link sources, which then has one or more link targets. This means that one link source can link to several places at the same time.
3.2.1.4 Object fields

Figure 9: Object field relation

Another way of specifying non-structural relations between objects is by using object fields. This is the weakest of the relations, since the object that is pointed out by the object field has no knowledge that another object is referring to it and when performing deep copy operations, the objects referred to via object fields are not copied. It is possible to refer to more than one object with an object field.

3.3 Other components/libraries used

- Antennahouse XSL Formatter [6]
  This component is used in Uptime in order to generate PDF documents from an XML file combined with an XSLT file which adds XSL-FO [7] tags to the data in the XML file.

4 A Brief Background to Modeling

Modeling is an important and cost-effective technique that together with simulation can be used for assuring that a system works as intended, and for diagnosing faults in the system. Modeling is often done via a modeling language. The modeling language can be graphical (and thus use diagrams with named symbols and connections to build the model) or textual (and thus use standardized keywords accompanied with parameters to express the model), many modeling languages also fall somewhere in between graphical and textual. For example Modelica [17] is a declarative modeling language which is textual with a syntax that resembles languages like Java; however there are graphical front-ends that let the user create and connect components in a graphical way. UML [18], Unified Modeling Language, is also an example of a popular and well known modeling language that has a very widespread support, especially for modeling of software.

No matter how the modeler goes about modeling, whether by using graphical methods such as diagrams or by writing code, the goal is in most cases to create a model of a system (real or imaginary) that is as close as possible to the original. If the modeling language used is one that is executable the modeler can use the model to run simulations in a systematic and cheap way. For example, in Figure 10 below is a model of part of a NASA satellite in a modeling tool called RODON [19]. This model can be used for diagnosing faults in the satellite when the satellite itself is orbiting the earth and tests cannot be done on it in an easy way.
Of course, creating an accurate model of a system is not an easy job and requires much domain knowledge. Thus, the modeler should often be a domain expert. In many cases, the modeler has no programming experience. Many modeling languages solve this dilemma by being declarative. The users do not have to worry about specifying the logical flow and can concentrate on specifying *what* the model should accomplish instead of *how*. The modeler can then specify how the system works via equations, logical expressions and constraints. When the model is finished it can be simulated.

Models are often hierarchical in order to alleviate the workload for the modeler (and in many cases, to better simulate the systems modeled). Some modeling languages have a “component” concept where the modeler is encouraged to create a set of reusable components and then reuse and connect these to create larger and larger models. These components are often parameterized in order to allow them to be configured for more generic usage across many models. Some examples of common components include resistors, batteries, wires and other basic physical components used for building complicated systems. Having a rich component library can extensively cut down the modeling time.

5 The Proposed Modeling Formalism

Modeling a system should be done in a way that is easy, yet powerful. The intended user is a safety engineer and the modeling formalism should reflect this and provide a formalism that is
less general and more tailor-made for this special purpose. Because safety engineers are generally not programmers, the modeling formalism should not require the user to write code or perform similar programming tasks in order to use this formalism to the fullest.

5.1 Creating a hierarchy

The systems we model are inherently hierarchical; a component consists of other subcomponents which in turn are consists of other subcomponents. Thus we need to be able to present the user with a component hierarchy. Since this modeling formalism is supposed to be used in top-down development, we should also support having “black-boxes” in our hierarchies and be able to work with these as well. Having a hierarchy also provides us with important information that we can use to filter the data that we present to the user, which is a good option when the data set grows large.

5.1.1 The first steps towards a working hierarchy

So, how can a system model be constructed? We always start out with a system. Even if we are constructing a purely functional model of a system, we are still modeling a \textit{system}, even though the system might only contain functions (and in most cases we are actually modeling a physical system that consists of several levels of subsystems and components).

A simple model of a system hierarchy is depicted in Figure 11 above. The “Top” system is made up of two subsystems, “S1” and “S2”, and “S1” is in turn made up of one subcomponent “C1”. This provides information about which ports are valid to directly
connect to for each component. For example, the component “C1” can only connect to the “Top” system through the subsystem “S1”. When we have this hierarchy we can help the user to make valid choices by filtering out all options that are not valid (for example, when connecting a port of C1 to another port, we can show the user that it is not allowed to connect a port of C1 to a port of S2), reducing the risk for mistakes and motivating the users to introduce some structure to their data as early as possible.

It’s easy to see that as long as we are only interested in the structure of the component hierarchy it will become a normal tree structure. However, when we add connections between component and function ports and other kinds of links between the nodes (for example, failure modes can affect functions) this quickly creates a more complicated graph structure. In this section, however, we are only interested in the hierarchy. More details about the other relations that turn the tree into a graph are given in section 5.2 on page 23.

Safety engineering is failure mode-centric, that is, we are in general only interested in what happens when something fails. A failure mode belongs to a subsystem or a component and affects one or more functions that this subsystem or component contains. This in turn leads to a hierarchy that looks something like Figure 12: Parent-child hierarchy.

![Diagram of Parent-child hierarchy]

**Figure 12: Parent-child hierarchy**

In Figure 12 the following relations are illustrated.
- A subsystem can contain one or more other subsystems, if a subsystem contains no other subsystems it is instead considered to be a component (a component is thus a subsystem that only contains functions and failure modes). To exemplify this, a component could be a resistor which we can consider to only perform a single function and which could be the smallest replaceable part of for example a toaster. A subsystem, on the other hand, could...
be the gearbox of a car, which performs a function by combining several other components

- A function, on the other hand, can be seen as a black box that performs a single function and never fails (and because of this, a function can not contain any failure modes). A function can never fail because it is not an actual physical object. A function is thought of as an abstract black-box that always succeeds in performing its function. If one wishes to model a failing black box, it should be done by creating a component containing the function and then let the component contain the necessary failure modes. A function thus represents a function that is performed by a component or a subsystem (such as “shift gear” for the gearbox, or “resistance” for the resistor).

- A subsystem or component can contain one or more functions.
- A subsystem or component can contain zero or more failure modes.
- Each failure mode contains one or more mission phase specific data containers that contain data that is specific to a certain mission phase. Each failure mode contains one of these objects per mission phases in the current mission profile.

5.1.2 Supporting black-box design of systems

It is unreasonable to expect that an engineer should be able to design a whole new system in detail from scratch in a single iteration. The design of a new system is more often done in an iterative way where the engineer in some places puts “black boxes” instead of a subsystem/component and then comes back later on and adds more detail to these black boxes. Such a black box represents a component that we know can take a certain input and transforms it into a certain output. However we cannot have a look inside it and see how it works. Working with these black boxes is a much needed feature in order to support the entire safety assessment process.

In our approach function objects are used as these black boxes. A function can either contain other functions or subsystems, or it can be empty. An empty function is treated like a black box, while a function that contains other functions and/or subsystems is treated like a function that is implemented by the objects that it contains. This also provides important information that can be used for, for example, next level effect prediction (for more about this see chapter 7.3, Connecting the model and the FMEA).
Figure 13: Function hierarchy example

As an example, in Figure 13 we see that we have a system “Top” which contains 5 functions (“Function 1” through “Function 5”) and one component (named “Component”) which in turn contains a single function “Component Function”. The black box functions in Figure 13 are Function 3, 4, 5 and the component function. Function 2 is implemented by the function 3 and 4, which even though they are black boxes provide useful structural information (maybe function 2 contains of two steps that needs to be done in separate subsystems, or function 2 is a very important function and thus are going to be implemented by two functions working in parallel. However we cannot know which until connections are added to the hierarchy.) Function 1 is implemented by the component that resides inside of it, this component in turn contains a single black box function which might be detailed later on (though it does not have to be, that depends solely on what level of detail the safety engineer deems necessary).

This feature does however require a revision of the parent-child hierarchy presented in Figure 12 on page 17. After the revision, the parent child hierarchy looks like Figure 14 below.
It is important to note here that the subsystem object type cannot be merged with the function object type even though they exhibit certain similarities. A subsystem represents an actual physical component that potentially contains other physical components while functions represent a collection of components that while working together fulfills an abstract function such as, for example, “Extend landing gear”. Also, the semantics of port connections for a function port and a subsystem port differs greatly (see chapter 7.3, Connecting the model and the FMEA on page 61).

### 5.1.3 Putting it all together

These observations create a basic tree structure that provides a logical place for each piece of information that is normally used when working in the safety engineering field as well as providing a solid base for augmenting the tree structure with more information.

Mission profiles are not something that we concern ourselves with when working in the model view; however it is very important when we actually use the model to generate FMEA’s later. Thus we do not include mission phase specific data in our modeling formalism, but rather add it when we find that it is needed. For more discussion about mission phase specific data, see chapter 6, Domain model.

The modeling formalism that has been developed contains the following basic building blocks

- **Components/Subsystems.**
  A component is a subsystem that only contains one or more functions and represents one physical component (for example a resistor). A subsystem contains one or more other components/subsystems, and also one or more functions, it represents one
physical subsystem (for example a power supply). Both components and subsystems can contain zero or more failure modes.

- **Function.**
  A function can both be a way of specifying a black-box function (in case of a function that contains no children), or a way of grouping which components/subsystems that realizes a certain abstract function (for example, “provide power”). Functions that contain no children are thought of as black-boxes that provide a certain function and cannot fail. By adding detail to a black-box (adding subsystems, components, sub functions etc. to it) the function becomes an implemented function.

- **Failure modes.**
  A failure mode is a way for a component/subsystem to fail, making it not provide one or more of its functions. A failure mode is specific for a certain component/subsystem, but it affects one or more function ports and thus also one or more functions. A function cannot contain failure modes since functions represent an abstract grouping of components/subsystems, however a failure mode can target ports inside the function by connecting to them via an observation port.

- **Ports**
  Apart from the three above identified types we also need something to connect these to each other. The above three types can contain ports, which in turn can be connected to other ports. The connection semantics can be found in chapter 5.2 on page 23.

The model specification in the modeling language MetaGME [8] is provided below in Figure 15.
Figure 15: Model specification in MetaGME

The connections and ports specified in Figure 15 have the following meaning:

- **ConnectionPort**
  Provides a super type for function and component ports, allowing us to treat them equally. Function ports and component ports have differing semantics, and thus function and component ports are separated into two separate types.

- **FunctionPort**
  Represents an in or out port of a function.

- **ComponentPort**
  Represents an in or out port of a component.

- **ObservationPort**
  An observation port allows us to observe one function or component port inside a function from a failure mode that belongs to the function's owner.

- **FailureModePort**
  Represents a failure mode port, failure modes share similar semantics to function ports, however they may only connect to function ports or observation ports, never directly to a component port.
- **Connection**
  Represents a connection between two ConnectionPorts.

- **Observed**
  This connection represents a connection from an observation port to the port that it actually observes.

- **ObservationPortChain**
  This connection represents that several ObservationPorts may connect to each other and create a chain of observation ports.

- **Affected**
  This connection represents the connection between a FailureModePort and an observation port or a function port.

### 5.2 Connection semantics

#### 5.2.1 Internal connections of a black box function

A black box function is a function that does not contain any children. Such a function is thought of to be maximally connected. That is to say that each of the functions input ports is connected to all of the functions output ports. An example of how this would look, should the connections be drawn in the model, can be seen below in Figure 16. Of course, drawing all of these connections would be both annoying and error-prone, and thus this is not done (apart from in the example figure below).

![Figure 16: Black-box functions are considered to be maximally connected inside](image)

#### 5.2.2 Redundancy

Redundancy is the duplication of critical components of a system in order to make the system more tolerable to failures. For example, large cargo trucks can lose a tire without any major consequences. They have so many tires that a losing a single tire is not critical (with the exception of the front tires, which are used to steer). Redundancy is an important engineering technique that is used in a plethora of systems. The more safety critical the system is, the more one needs to consider adding redundancy. Of course, our model needs to be able to model redundancy in an accurate way.

The simplest case of redundancy is a model consisting of three black-box functions A1, A2 and B where B only requires one of either A1 or A2 to work. (A1 and A2 could be two ways of supplying power, and B could be a function that requires power to work.) This case is modeled as in Figure 17 below.
In Figure 17 above, the failure of a single function A1 or A2 will not cause the failure of function B, however if both function A1 and A2 fails at the same time, function B will not get any input and will also fail. Uptime Engineering supports the above definition of redundancy. However, now it is easy to think that if you connect three components in the same way (as seen below in Figure 18) you would also have modeled redundancy and this is not true.

In Figure 18 above the port on component B is not considered to be a normal port, it is considered to be a bus, taking two separate inputs (one input from component A1, and one from A2) and thus the failure of either A1 or A2 will cause B to not receive all of its specified inputs. So, how do we do if we wish to model redundancy between components?
Figure 19: Modeling redundancy between components

The solution is simple and can be seen above in Figure 19. By combining the models in Figure 17 and Figure 18 we arrive at a model where the functions from Figure 17 are implemented by the components in Figure 18 and then gain redundancy from the connections between the functions.

5.2.3 Connecting failure modes

Of course, the interesting thing about modeling for the safety assessment process is not just modeling the system; it’s modeling what happens when the system fails. And by adding failure modes into the mix and connecting them to ports, we can do just that.
In Figure 20 above is a very simple model with added failure modes. A failure mode can be read as an “explanation” to how a function failed. Either function 2 performs its function, or failure mode 2 is active. A failure mode is always associated with one or more function ports, and it can be associated both with in- and out-ports (with differing semantics). These different semantics are demonstrated in Figure 21 below.

Figure 21: Failure modes on a serial connection component example

As can be seen in Figure 21, there are two top level failure modes, “FailureMode 1 or 2” and “FailureMode 3”. “FailureMode 1 or 2” is connected to the in-port of function 3, and
“FailureMode 3” is connected to the functions out-port. However even though they are connected to the same function, the semantics for them are wildly different.

When “FailureMode 3” is active, we know that the function that failed is function 3. It cannot be any other function that triggered this failure mode. We can say that “FailureMode 3” is closely connected to and affecting function 3. If “FailureMode 1 or 2” is active, we know that some function that provides input to the port that this failure mode is connected to has failed. This means that either function 1 or function 2 (or both) has failed and is now providing faulty output that is fed to the input of function 3, causing this failure. “FailureMode 1 or 2” is distantly connected to function 1 and 2, and is affecting function 3.

Note that in Figure 21 these failure modes can be likened to observation points. At the points we have specified failure modes we can imagine that we test the current output value. At the point of “FailureMode 1 or 2” we have a test that tells us if an error has occurred somewhere on the path before this test. At the point of “FailureMode 3” we have a more specific test that lets us know that some error has occurred that is caused specifically by function 3 failing.

Figure 22: Failure modes can be connected to one or more ports of a function

This means that the semantics is quite powerful since we can model complicated dependencies amongst components without having complete information about them. Figure 22 above is a model of a simplified actuator that shows off some more advanced capabilities failure mode connections. There are two failure modes connected to a single port of the inner function (Provide Power), these failure modes represents tests that test a certain output of the
function (in this case, if one of the internal generators has failed) and reports a fault if this specific output is faulty (if one of the internal generators has failed). The failure mode “Actuator does not receive power” however is connected to the output of the outer function and reports a fault if both of these outputs are faulty. This is because the failure mode has two ports, each connected to a single out-port of the function (marked red in the figure). So if either one of the internal generators fail the corresponding failure mode (internal generator 1 or 2 failed) is active. But at the top level “Actuator does not receive power” is not active since only one of the internal generators has failed (and the other one still provides enough power to power the translational motion), however if both of the internal generators fails, there will not be enough power to power the actuator and the failure mode “Actuator does not receive power” will become active.

But what if we wanted a failure mode on the top level that represented “either of the internal generators failed”? (Maybe the actuator requires a lot of power?) In this case we can make use of the redundancy notation specified in chapter 5.2.2 on page 23 as shown in Figure 23 below.

![Figure 23: Making an or-relation to a failure mode](image)

In the above figure, “Actuator does not receive power” will trigger if either of the inner ports fails to pass the test (in this example, if either of the power generators fails). This is since the failure mode has a single port that is connected to both of the functions out-ports (marked with red in the figure), thus forming an OR-relation between these ports (if either of the ports reports a failure, this failure mode should become active). While this might not be a situation that happens very often (and should preferably be handled with two different failure modes on the top level, just as on the inner level), this modeling formalism can still represent this case if the modeler wishes to use it.
Sometimes, connecting failure modes to function ports alone might not be detailed enough. If we want to model that a failure mode can be connected to a component port we can connect to it via an observation port as seen below in Figure 24.

![Figure 24: If want to make failure modes that are detailed to certain component ports, we can connect the failure mode to these ports by use of observation ports.](image)

In Figure 24 we have implemented function 1 with two components, and since we want as detailed failure information as possible we want to have one failure mode for the failure of each component. To connect the failure modes to the component we use observation ports to connect through. These observation ports can in turn connect to other observation ports, however only functions may have observation ports. Observation ports cannot be used as input or output ports, a different port should be created for that purpose (this is shown in Figure 24).

**5.3 Extracting useful information from the model**

When we have a valid model of a system we can use it to extract useful information from it. This information *could* be extracted from the model manually by a user, but to do so would require a lot of work as the model grew larger and the process would still potentially be very error-prone.

**5.3.1 Calculating the global effect**

As you might remember from chapter 2, the global effect of a certain failure mode is which of the top systems failure modes becomes active if the failure mode is active. If we have the next level effect on each level of the system tree, we can easily calculate the global effect of each
failure mode – thus further removing some manual work from the process. The global effect is always one of the failure modes on the top level of the system tree, and by moving from the top and downwards in the tree we can easily calculate the global effect on each level.

Figure 25: Global effect example

Demonstrating the concept, if we have a system tree that looks like the tree in Figure 25 above we can (since it is not very large) look at it and follow the next level effect arrows from any of the failure modes until we either find a failure mode on the top level, or we find a stop in the references (like in “FailureMode S12”). If we find one of these stops, then we know that the failure mode has no global effect, and if we found a failure mode at the top level, we know that this failure mode is our global effect. This does not have to be done bottom up (even though it might look like the logical way of doing it when we have the graphical representation in front of us).
The algorithm we use for setting the global effect looks as follows

- Set the top systems global effects to themselves.
- Set each global effect on the next hierarchy level to the global effect of their next level effect.
- Call the algorithm recursively on each of the current systems children.

Figure 26: First two steps in setting the global effect on the tree given in Figure 25

As can be seen in Figure 26 above, in step 1 we set each failure mode on the top level to be their own global effect. Then in step 2, we follow the next level effect on each of the failure modes on the next hierarchy level and set the global effect accordingly. The failure mode S11 has T1 as its next level effect, and thus it will share global effect with this failure mode (which happens to be T1). S12 on the other hand has no next level effect, and thus it will not have a global effect (also note here that in the next step, S22 will also be set to have no global effect since its next level effect does not have a global effect. This is a contained fault; a failure of this kind does not affect the top system in any way.)

**5.3.2 Calculating next level effect**

There are two basic ways to connect components in a model, in series or in parallel. This section describes how the next level effect calculation algorithm works by showing how it works on two simple models, one consisting of several components connected in series, and one consisting of several components connected in parallel. After that we will see how the algorithm works on some more complicated models. The pseudo code for the algorithm presented in this chapter can be seen in Code sample 1 on page 41.
To start off, we can look at what the result should be. According to the model specification presented in chapter 5, the correct next level effects for the model presented in Figure 27 should be the values presented in Table 3 below.

<table>
<thead>
<tr>
<th>Failure mode</th>
<th>Next level effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top::FM1</td>
<td>No next level effect (there is no next level)</td>
</tr>
<tr>
<td>Top::FM2</td>
<td>No next level effect (there is no next level)</td>
</tr>
<tr>
<td>Top::FM3</td>
<td>No next level effect (there is no next level)</td>
</tr>
<tr>
<td>Component 1::FM1</td>
<td>Top::FM1 or Top::FM3</td>
</tr>
<tr>
<td>Component 2::FM1</td>
<td>Top::FM1 or Top::FM3</td>
</tr>
<tr>
<td>Component 2::FM2</td>
<td>Top::FM1 or Top::FM3</td>
</tr>
<tr>
<td>Component 3::FM1</td>
<td>Top::FM3</td>
</tr>
<tr>
<td>Component 4::FM1</td>
<td>Top::FM2 or Top::FM3</td>
</tr>
</tbody>
</table>

Table 3: Correct next level effects for the model in Figure 27

As can be seen in Table 3, we are often unable to separate between the failure mode Top::FM3 and either Top::FM1 or Top::FM2 (except for one case). This is as it should be, because that is how the current model is designed. In this case we’ll just have to settle with filtering the users’ options, telling the user that “we know that it is one of these two, but we’re not sure which one of these”.

In Figure 28 below the paths taken by the algorithm to find the next level effect for Component1::FM1 and Component 3::FM1 are drawn (as well as two special points of interest). Calculating the next level effect for the last three failure modes (calculating the next level effect for the failure modes on the top level is trivial, since they have none) is very similar to calculating the next level effect of Component1::FM1 or Component3::FM1.
The easiest next level effect to calculate is, not so surprisingly, the one of Component3::FM1. The path taken by the algorithm is drawn with a green line in Figure 28. This path is produced by starting at the failure mode, and then going outwards (to the right) until we end up in a port that belongs to the failure modes parents owner. Here the failure mode is Component3::FM1, the failure modes parent is Component 3, and the owner of Component 3 is Top. So, we go to the right until we find a port that belongs to Top. If we enter a junction, we branch into two paths and follow them separately (more about this later). If we cannot find a port that belongs to Top, then we discard the current path (since in that case, there is no output to test for faults on the next level and we have a fault that we cannot actually “see” at the next level). We stop when all paths has either been discarded or found an end port on the next level (in this example, Top).

To return to the current example, the path of ports that we get for Component3::FM1 is marked with green. Now that we have this path, we examine all failure modes on the path. On this path, there are three failure modes (Component4:FM1, Top::FM2 and Top::FM3). We now examine these in order.

- **Component4::FM1**
  Component4::FM1 cannot be the next level effect (since it is not on the next level), but maybe it becomes active if Component3::FM1 becomes active? A quick examination shows that Component4::FM1 is connected to the out port of the Component4::Function, this means that (according to the specification in chapter 5) this failure mode is active if and only if Component4::Function has failed. And the way that this model is done, Component4::Function does not fail when Component3::Function fails. Thus Component4::FM1 is not active.

- **Top::FM2**
  Top::FM2 is bound to the out port of Component4 (marked with a purple circle in Figure 28), just like with Component4::FM1, this means that Top::FM2 is active if and only if Component4 has failed. But this is not the case (Component 3 has failed, not Component 4, and they are siblings), and thus Top::FM2 is not active.

- **Top::FM3**
  Top::FM3 is bound to the out port of Top Function. Top Function happens to be in Component3’s parent hierarchy. This means that the failure of Component 3 affects Top Function (since the path has not been neutralized by means of redundancy before reaching Top Function’s out port) and Top::FM3 is a candidate for the next level effect.
This means that after examining the failure modes on the path in order, we have eliminated all of them except Top::FM3, which we then choose as the next level effect.

So, what happens when doing the same thing with Component1::FM1? The path taken this time is marked with red in Figure 28. This time we find some more failure modes found on the way, the list of failure modes found this time is Component2::FM2, Component2::FM1, Component3::FM1, Component4::FM1, Top::FM1, Top::FM2 and Top::FM3. Just like last time, we examine these in order.

- **Component2::FM2**
  Component2::FM1 is bound to the in port of Component2::Function (marked with a blue circle in Figure 28) and thus, this means that (according to the specification in chapter 5) this failure mode is active if and only if the input at this point is faulty. And in this case the input is faulty. This means that Component2::Function will also fail as a chain reaction of Component1::Function failing. Component2::FM1 can however not be the next level effect since it isn’t on the next level.

- **Component2::FM1 (and also Component3::FM1, Component4::FM1 and Top::FM2).**
  These four failure modes are handled in the same way. All of these failure modes are assigned to the out port of a function of a sibling component (except Top::FM2 which is assigned to the out port of Component 4, but the important word here is not “function” but rather “out port”). This means that they will not become active as a chain reaction of Component1::FM1 failing, and they cannot be the next level effect (since only Top::FM2 is on the correct level for that, and we have already established that Top::FM2 will only become active if Component4 fails).

- **Top::FM1**
  Top::FM1 is bound to the out port of Function 2, which is connected to a port that is affected by Component2::FM2 which, as we earlier established, became active as a chain reaction of Component1::FM1. Since Top::FM2 is on the next level (from Component1’s viewpoint) it becomes a valid candidate for the next level effect. (Note here that if Component2::FM2 would have been bound to the out port of Component2::Function instead, Top::FM2 would not have become a valid candidate.)

- **Top::FM3**
  Top::FM3 is bound to the out port of Top Function. Top Function happens to be in Component1’s parent hierarchy. This means that the failure of Component 1 affects Top Function (since the path has not been neutralized by means of redundancy before reaching Top Function’s out port) and Top::FM3 is a candidate for the next level effect.

When all of the failure modes on the path have been examined we see that we have two options left, Top::FM1 and Top::FM3. Both of these could potentially be the next level effect, and depending on what we are modeling the answer is different. So in this case we let the user choose one of them.
Figure 29: An example model for calculating the next level effect of several components in parallel

So how does the algorithm perform when calculating the next effect of a system that is connected in parallel? Figure 29 shows one example of how such a system might look; the system in this example is a revisit of the actuator from before. Again, applying the model semantics by hand we can see that we end up with the results shown in Table 4.

<table>
<thead>
<tr>
<th>Failure Mode</th>
<th>Next level effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actuator::Wind power component faulty</td>
<td>No next level effect (there is no next level)</td>
</tr>
<tr>
<td>Actuator::One of the power components is faulty and needs repair</td>
<td>No next level effect (there is no next level)</td>
</tr>
<tr>
<td>Actuator::Actuator fault</td>
<td>No next level effect (there is no next level)</td>
</tr>
<tr>
<td>Propeller::Propeller stuck</td>
<td>Actuator::Wind power component faulty or Actuator::One of the power components is faulty and needs repair</td>
</tr>
<tr>
<td>Power transformer::Transformer faulty</td>
<td>Actuator::Wind power component faulty or Actuator::One of the power components is faulty and needs repair</td>
</tr>
<tr>
<td>AC Adapter::AC Adapter faulty</td>
<td>Actuator::One of the power components is faulty and needs repair</td>
</tr>
<tr>
<td>Actuator component::Mechanical device stuck</td>
<td>Actuator::Actuator fault</td>
</tr>
</tbody>
</table>

Table 4: Correct next level effects for the toaster in Figure 29

Applying the algorithm on “Propeller::Propeller stuck”, “AC Adapter::AC Adapter faulty” and “Actuator component::Mechanical device stuck” leads to the following paths being calculated (“Power transformer::Transformer faulty” does not differ much from “Propeller::Propeller stuck”).
Figure 30: The paths taken of the next level effect calculation algorithm in the actuator model in Figure 29

Starting with the easiest one, the blue path on the far right on Figure 30 shows the path taken if “Actuator component::Mechanical device stuck” is found to be active. The only candidate that we find on this path is “Actuator::Actuator fault”, which happens to be the on the correct level and affecting the correct function and it is thus chosen as the next level effect.

The red path shows the path taken if “Propeller::Propeller stuck” is active. This path contains the following failure modes (in order): “Power transformer::Transformer faulty”, “Actuator::Wind power component faulty”, “Actuator::One of the power components is faulty and needs repair”, “Actuator component::Mechanical device stuck” and “Actuator::Actuator fault”. Examining them in order yields the following results:

- **Power transformer::Transformer faulty**
  This failure mode is on the wrong level (since it belongs to a sibling) and is discarded.
- **Actuator::Wind power component faulty**
  This failure mode is on the correct level and assigned to the out port of a function in Propeller’s parent hierarchy. This, as you might remember from the previous example, lets it become a candidate.
- **Actuator::One of the power components is faulty and needs repair**
  This failure mode is on the correct level, and assigned to an in port of a function. This means that it becomes active if the input is faulty, which it is (since “Propeller::Propeller stuck” is connected to it and active). Thus this failure mode also becomes a candidate.
- **First point of redundancy (FPR)**
  Now before we examine the rest of the path we notice that the last examined port has two redundant connections to it. Since we only have one path to follow we cannot pass this point (what has failed is only one of the two redundant inputs, so the other one will still provide the correct input for the “Mechanical Translational motion”- function). This means that a single failure of Propeller will not affect anything beyond this point. With this in mind, we stop examining this path, since anything that we find after this port is invalid and cannot be a next level effect.

The remaining candidates after this are “Actuator::Wind power component faulty” and “Actuator::One of the power components is faulty and needs repair”. Both of these could
potentially be the next level effect, and depending on what we are modeling the answer is different. So in this case we let the user choose one of them. In this example with our actuator we choose the failure mode “Actuator::Wind power component faulty”, but if this had been another system with a similar design we might have been inclined to choose the other one instead.

The green path shows what happens if “AC Adapter::AC Adapter faulty” is active, this is very similar to what happens if “Propeller::Propeller stuck” is active except that “Actuator::Wind power component faulty” is not on the resulting path this time.

Do note here that if “Actuator::Wind power component faulty” and “Actuator::One of the power components is faulty and needs repair” had not existed in the mode, no single failure of a component before the FPR could trigger a failure mode in the Top system.

Of course, calculating simple examples of components in series or parallel is not so impressive, but what if we combine them and make something slightly more complicated?

Figure 31: A slightly more complicated model to calculate the next level effect of

While more complicated than the earlier models, it is still possible to calculate the next level effects of the failure modes of the bike-powered water boiler presented in Figure 31 by hand. Doing so produces the following table:
### Table 5: Correct next level effects for Figure 31

<table>
<thead>
<tr>
<th>Failure mode</th>
<th>Next level effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Water boiler::Water is not boiled</td>
<td>No next level effect (there is no next level)</td>
</tr>
<tr>
<td>Training cycle::Cycle broken</td>
<td>Water boiler::Water is not boiled</td>
</tr>
<tr>
<td>Current transformer 1::Broken</td>
<td>No next level effect (redundancy)</td>
</tr>
<tr>
<td>Voltage transformer 1::Broken</td>
<td>No next level effect (redundancy)</td>
</tr>
<tr>
<td>Current transformer 2::Broken</td>
<td>No next level effect (redundancy)</td>
</tr>
<tr>
<td>Voltage transformer 2::Broken</td>
<td>No next level effect (redundancy)</td>
</tr>
<tr>
<td>Boiler::No/Not enough power</td>
<td>Water boiler::Water is not boiled</td>
</tr>
</tbody>
</table>

So, applying the algorithm on these failure modes will produce the following calculations (shown in Figure 32 below).

Starting out with the easy one, the green line that goes from “Current transformer 2::Broken” will not trigger “Boiler::No/Not enough power” since it passes the junction just before the “Boiler”, which means that redundancy takes care of this path.

The interesting thing about this model is what happens if the “Training cycle” fails. This is illustrated with red lines in the figure. Since “Training cycle” provides input for two other components, we examine both of these paths and notice that the redundancy in the junction in front of the “Boiler” does not save us since both of the paths will have failed. Thus a failure of “Training cycle” will trigger “Water boiler::Water is not boiled” even though we have redundancy on all of the paths from “Training cycle” to the top system. If we want this model to be safe from a single failure of “Training cycle”, we have to add more redundancy that is not dependant on “Training cycle”.

The algorithm used in the above examples is presented below in Code sample 1. This pseudocode example makes use of some basic functions that needs to be present in order for the algorithm to work; these are detailed in the following list:
- **Make Stack, Make List, Make Hashtable, Make Path**
  These functions create and return a new object of the correct type. These objects are assumed to have the basic functionality you could expect from such a construct (Stacks have methods for Pushing and Popping). Paths are essentially lists of Ports with some added functionality for searching and iterating through the path.

- **Path.FindFirstPointOfRedundancyFrom(Port)**
  Returns the first function in-port on the path (starting the search from the port given as argument) that has more than one connection into it.

- **Path.ContainsRedundancyFrom(Port)**
  Returns true if there is a first point of redundancy on the path (starting the search from the port given as argument).

- **Port.HasLinkedFailureModes, Port.GetLinkedFailureModes(), Port.GetLinkedFailureModePorts()**
  Provides convenience methods for getting failure modes linked to a certain port.

- **Port.LinkedOutPorts()**
  Convenience method for getting all out-ports that a certain failure mode links to.

- **Intersection(Collection, Collection)**
  Returns a collection containing all items that exists in both collections.

- **SubSystem.GetAllSubSystems(), Function.GetAllSubSystems()**
  Returns all components/subsystems contained by the container that this method is called on. For the purposes of using this function, a subsystem and function is considered to contain all of the subsystems that its child functions (and their child functions, recursively) contains as well.

Using the above helper functions, the algorithm can be written as seen below.

```cpp
// The failure mode that we wish to find the next level effects of is
// represented by the variable "This".

// Prepare a stack for searching for paths
S <= Make_Stack;
Paths <= Make_List;
ForEach(Port p in This.Ports) {  
    S.Push(Make_Path(p));
}

// Construct the paths
While(S is not empty) {  
    CurrentPath <= S.Pop();
    LastNode <= CurrentPath.Last();
    If(LastNode.Parent is This.Parent.Owner OR
       LastNode.NextConnectedPorts() = 0) {  
        Paths.Add(CurrentPath); // Path is finished
    }
    ForEach(Port next in
       LastNode.NextConnectedPorts()) {  
        If(next not in CurrentPath) {  
            // Only follow each loop once.
            S.Push(CurrentPath + next);
        } else {  
            Paths.Add(CurrentPath + next); // Found a loop, path is finished
        }
    }
}

// Find all failure modes on all paths, and all ports that those failuremodes
// are connected to.
FailureModePortsOnPaths <= Make_List;
FailureModesOnPaths <= Make_List;
RedundancyMarker <= Make_HasTable(Key => Port, Value => List<Path>);

ForEach(Path current in Paths) {
```
if(current.ContainsRedundancyFrom(current.Begin())) {
    // Calculate where to stop searching the path if it is redundant
    End <= current.Begin();
    Repeat |
        FoundEnd <= False;
        P <= current.FindFirstPointOfRedundancyFrom(End);
        if(RedundancyMarker.ContainsKey(P)) {
            // Found this redundancy point earlier
            OldPaths <= RedundancyMarker[P];
            SameSourceAsEarlier <= False;
            ForEach(Path OldPath in OldPaths) {
                // Check if we are on the same path as one that we’ve
                // found earlier:
                if(OldPath.Find(P).Previous() = current.Find(P).Previous()){
                    SameSourceAsEarlier <= True;
                }
            }
            if(SameSourceAsEarlier) {
                FoundEnd <= True;
                End <= P;
            } else {
                // We might be able to break through this redundancy.
                // Check if there’s enough paths here to break through
                RedundancyMarker[P].Add(current);
                if(RedundancyMarker[P].Count = P.NumberOfRedundantConnections) {
                    // Broken through!
                    if(not current.ContainsRedundancyFrom(End) |
                        FoundEnd <= True; |
                        End <= current.End(); |
                    } else {
                        End <= P;
                    }
                }
            } else {
                // Found a new redundancy point
                RedundancyMarker[P] <= make_list(Current);
                End <= P;
                FoundEnd <= True;
            }
        }
    Until(FoundEnd); |
} else {
    // No redundancy on the path, search the entire path
    End <= current.End();
}

// Search the path for failure modes and the ports that
// they are connected to.
Iter <= current.Iterator();
While(Iter.Current /= End) {
    Iter.MoveNext(); // Skips first port, makes sure that we include last.
    If(Iter.Current.HasLinkedFailureModes) {
        // Grab any linked ports with failure modes, and failure modes
        if(Iter.Current.GetLinkedFailureModePorts() not |
            in FailureModePortsOnPaths) {
            FailModePortsOnPaths.Add(
                Iter.Current.GetLinkedFailureModePorts());
        }
        if(Iter.Current.GetLinkedFailureMode() not in FailureModesOnPaths) {
        }
    }
}

// Removes all failure modes that has an AND condition on a port that
// is never visited by our paths.
ForEach(FailureMode fm in FailureModesOnPath) {
    if(fm.Ports.Exists(p => p not in FailureModePortsOnPath)) {
        FailureModesOnPath.Remove(fm);
    }
}
// Initializes a list of failuremodes that gets triggered as a chain effect of the failuremode “This” becoming active
Chained <= Make_List;
Chained.Add(This);
// Adds all failuremodes on any path that triggers on faulty input
ForEach(FailureMode fm in FailureModesOnPath) {
    if(not fm.Ports.Exists(p => p.LinkedPort.Direction /= In) {
        Chained.Add(fm);
    }
}
// Tries to add new chained failuremodes
Repeat {
    NumChained <= Chained.Count;
    ForEach(FailureMode fm in FailureModesOnPaths) {
        if(fm in Chained) {
            continue;
        }
        NoFoundTrigger <= True;
        // Only need to check outports, inports are automatically failed if on the path
        ForEach(Port outport in fm.LinkedOutPorts()) {
            ForEach(SubSystem sys in outport.parent.GetAllSubSystems()) {
                if(Intersection(sys.FailureModes, Chained).Count /= 0) {
                    NoFoundTrigger <= False;
                }
            }
        }
        if(not NoFoundTrigger) {
            Chained.Add(fm);
        }
    }
} Until(Chained.Count = NumChained);
// The candidates for the next level effect is all chained failuremodes that are on the correct hierarchy level.
Candidates <= Make_List;
Candidates.AddAll(Chained.GetAll(fm => fm.parent = This.parent.Owner));
// At this point, the result is stored in the list “Candidates”

Code sample 1: Pseudocode of the algorithm used to calculate the next level effect

An important question is how we handle loops in our model. A loop exists when we can find a path from a certain node back to the same node. Also complicating matters is the fact that a loop can contain subloops, which in turn can contain subsubloops (ad infinitum). An example of a model containing loops and subloops can be found below in Figure 33.
Figure 33: A more complicated model that features loops, for a clearer view of the actual loops, see Figure 34

We cannot simply ignore loops, as the model in Figure 33 shows. All functions wrapping the components C1-C4 has been omitted from the model in order to make it easier to follow the connections in the picture. The loops are also presented in a clearer view in Figure 34 below. Ignoring loops in this model would cause us to erroneously calculate that FM1 would have no next level effect (and as can be seen, in reality all three of the top failure modes get triggered by FM1).
When working with more complicated models it can sometimes become hard to follow the flow when viewing the actual model representation. It can help to instead draw the connections between the ports as a directed cyclic graph instead (as seen in Figure 34).

Figure 34: The connections in the model shown in Figure 33 drawn as a directed cyclic graph, green nodes are function ports, yellow nodes are component ports and gray nodes are the ports of the top system.
After drawing the paths that the algorithm takes into Figure 34 we get a slightly less readable graph that can be seen below in Figure 35.

Figure 35: Figure 34 with the paths that the algorithm takes drawn in it.

While walking through the entire algorithm for this example we can easily see that if we follow the paths taken we will catch all of the failure modes in the loops, and thus the algorithm will return all of the three Top failure modes.
Something that is also visible at this point is that it, for the purposes of this calculation, is totally unnecessary to go through a loop more than once.

5.4 Summary
In this chapter the proposed modeling formalism has been presented. In chapter 5.1 the basic building blocks was presented. These are:
- Subsystems/Components
- FailureModes
- Functions
- Ports and connections
Figure 15 on page 22 showed the metamodel of the building blocks and their relations in MetaGME [8].

In chapter 5.2 semantics was introduced for connecting the building blocks with each other in order to be able to model important concepts such as redundancy. This chapter also illustrated the semantics for connecting failure modes to ports.

In chapter 5.3 two algorithms was presented, one for calculating the global effect of a failure mode with the precondition that the next level effect is known, and the other one for calculating the next level effect of a failure mode. The chapter also showed some examples where the algorithms were applied on some very basic models.

6 Domain model
This chapter presents the data model that was created simultaneously with the modeling formalism presented earlier in chapter 5 for the prototyping purposes. The chapter also discusses some design decisions and previous iterations of the domain model. In order to understand this chapter fully the reader should have read chapter 2 and 3 before reading this chapter.

The domain model for the safety assessment process contains a lot of data, even when limited to the FMEA process. Much of the data is related to the values of other data, and it all ties together in a big web of relations. For example, when doing an FMEA you need to consider which mission phase (for more about mission profile and mission phases, see section 2.1) that is currently active. The failure of the landing gear of an airplane during the planes flight is not necessarily as dangerous as the failure of the landing gear during the planes takeoff, or during its landing. Thus the severity of the failure mode “Blown tire” during the mission phase “In flight” could be very low, while it might be very high during the mission phase “Landing”.

The data model used in this project can be seen in Figure 41 on page 53. This chapter will discuss earlier variations of the data model and motivate how and why it looks the way it does.

6.1 Transforming the model hierarchy to fit the FMEA domain
The hierarchy presented in section 5.1, Creating a hierarchy, on page 16 can be augmented and transformed to fit the FMEA domain. There are a lot of standards and variations on how to write an FMEA and what information it should contain. For example, look below at Figure 36: SAE ARP4791 [1] Functional FMEA Worksheet Example.
The above figure is taken directly from SAE ARP4761 [1], Aerospace Recommended Practice and while it seems straightforward and simple, there are plenty of other variations on how the worksheet should look. Some even have columns for specifying a hierarchy directly on the worksheet. And while the above worksheet has a column for specifying mission phases (called “Flight Phase”), this is not something that is featured in every worksheet.

There are some basic differences that need to be taken care of in order to transform the model into a representation useful for FMEA:

- An FMEA does not have hierarchical functions. This means that the function part of the hierarchy needs to be flattened.
- An FMEA does not allow functions to contain subsystems/components. This means that the subsystems/components that are children of functions need to become children of their nearest subsystem parent instead.
- An FMEA makes use of mission phases, which requires the failure modes to have mission phase specific data.
- Ports and connections are not examined in an FMEA. This means that we can simply ignore these parts of the model.

Luckily, all of these are relatively simple transformations. Adding mission phase specific data objects to each failure mode is simply a matter of finding all failure modes in a model and adding mission phase specific data objects to them. Ignoring ports and connections is completely trivial. Since both of these transformations are trivial the will not be detailed further here. While the first two transformations are almost as simple they are most easily shown in image-form, as shown below in the two figures below.
Both of the transformations shown above in Figure 37 and Figure 38 consists of the finding the nearest subsystem parent (hereafter called the *owner*) and moving a node there. A very nice property of both of these transformations is that the order of which we do them in does not matter. Doing them in any order leads to the same result in the end. Also, the result both of these transformations can be calculated effectively. This leads to the conclusion that we do not actually have to *perform* these calculations. We can keep the models system tree structure when working with the FMEA and do these transformations “on the fly” in the application instead. This has one major advantage over actually performing the transformations, and that is that it is much easier to track changes made in either the FMEA tree or the model tree since the last performed synchronization between the model and FMEA trees.
6.1.1 Keeping reuse in mind

Modeling systems is a prime area for reuse of information. The main candidate that came up for reuse was, not surprisingly, the reuse of subsystems. One can easily see that subsystems (or components) are something that could potentially be reused very often when developing a new system. Think about, for example, how many times a normal resistor is used in an electrical system. Maybe there is only one resistor, but there might as well be a hundred or a thousand if the system is large enough. Reusing as much information as possible here felt like a good goal to have in mind. Not only would this save space (even though that is of less concern) but it would also help in other cases. Imagine, for example, a subsystem that is reused in 20 different places in a model. In the beginning of the development of the system this is just a simple black-box subsystem containing just a single function (or maybe several functions). But as the development progresses the system gets detailed further and further. Maybe the black-box function of the subsystem is implemented by two other components connected in parallel? No matter what the implementation might be, if the same component is in 20 places we don’t want to implement it 20 times. We want to implement it once and then have the other 19 synchronize the changes. By making this an easy operation to do, we can encourage the users to be more specific in their safety assessment by taking away the annoying repetition from their work.

By now we can probably all agree that some sort of reuse of information is good and useful, but the interesting question here is not “Should we support reuse of components?” but rather “How should we support reuse of components?”

6.1.1.1 A first attempt at the reuse of components

A component will most likely be used in plenty of models (and probably many times in the same model as well) and it would seem a good idea to only have one actual component object instead of tens or hundreds. This led to an architecture that consisted of three concepts, the structure node, the information node and the instance node.
Figure 39: Example of a shared component (The light bulb)

The component architecture can be seen in Figure 39 above. This contains the three previously mentioned node types, the instance nodes, the structure nodes (super nodes) and the information nodes. The basic idea here is that the structure node is the super type of the information node, and the instance node is the super type of the structure node. This means that in order to reuse a component in another place we simply would create a new instance node and put it as the child of a structure node (in Figure 39, “Flashlight system” and “Headlamp system” are both structure nodes). The instance node would contain any instance-specific information that the component might have (for example, the components name) while the structure node would only contain structural information (such as which subcomponents, functions, failure modes etc. that the component contains). The information node would contain all of the actual information (such as component manufacturer, part number, description and so on).

This would also have the effect that if, for some reason, the information in the information node was changed this change would automatically be visible everywhere in the system that this component is used. For example, if the manufacturer of the light bulb would change. And any addition or change of failure modes or any other structural information would also be
automatically available in any place that the component has been used. For example, if we change the tungsten filament for a carbon filament, this change would also automatically be propagated to all places that the light bulb is used. Also, any new component made by anyone could, potentially, be a “reusable library component” at once without any extra work.

However, it turned out that this level of reuse was not a wanted feature. “Automatic” updates (the kind that “just happens” without user consent) was deemed as a feature that could potentially be dangerous for this domain. The possibility existed that a user could potentially destroy other models by changing shared components to fit his or her system better. The user might take a library component and put it into their model thinking that it is a close enough fit for what the user is looking for. When the user then “adapts” the components by for instance adding a failure mode, that failure mode is added to all other places that the component is reused in. By doing this the user have accidentally changed other models which might have very dangerous side effects.

This data model, even though it might be interesting and useful in some domains, just does not fit the regular use cases of the safety engineers. And while it could potentially be augmented to work (for example, by copying the entire subtree when a user does a change that should be “local only”) it is not worth it. This kind of data model would require us to have superobject-types of other superobject-types (super-super-types), something that the Uptime database has no built in support for (even though it is possible). Implementing support for this would take a lot of time and effort that could be better spent on other more requested features.

6.1.1.2 Going back to the basics

At this point it was obvious that a more low-tech solution for reuse would fit this domain better and be more appreciated by the end-users. Updating all instances of a certain component would still be a useful feature to have in the future, but if it should be done it should be done under controlled forms when the user has specifically chosen to “synchronize” all instances of that component (and seen a preview of the changes that this will cause). It should also be possible to limit the synchronization to just a few components (for example, all instances of this component in a certain model), not every single component instances in all models in the entire system.

Splitting the structure and the information still seemed like a good idea (and was later on shown to make connecting the model with the FMEA much easier, see chapter 7.3, Connecting the model and the FMEA, on page 61), and thus by simply removing the instance nodes we arrive at the new hierarchy model as seen below in Figure 40 below.
This however presented us with a new conundrum. Now there was no actual data sharing going on in the data model, how can we do the synchronization step in the future? The solution here lies in the component information. Each component should have a single manufacturer (which is omitted in Figure 40) and a single part number. This combination is supposed to be unique. So if the “Tungsten filament” is made by “Filaments inc.” and has the part number A42, then if there is another component from “Filaments inc.” with the part number A42 they are equal. This means that if we wish to perform a synchronization step, we simply search all components in the database for components with the same manufacturer and part number combination and then synchronize all found components.

This is, of course, not as effective as sharing the information (it requires both more time and space) but doing such synchronization is hardly an everyday operation and thus that tradeoff can be seen as acceptable. Also, as an added bonus, by laying out the tree structure in this way we can make most of all of the Uptime databases existing features without the need of implementing new custom ones.

6.2 Adding more relations to the graph

Of course, a system is made up of much more than a simple hierarchy. Part of what makes it interesting is how the components and functions that make up the hierarchy are interconnected with each other. As stated in chapter 3, UpTime platform, there are four main ways of relating data to other data. Apart from the super and parent-relations that was used for creating the hierarchy in section 6.1 there are also links and object fields that provides weaker relations that are useful for creating a graph out of the tree.

6.2.1 Affected functions

Each failure mode can affect one or more functions (if it did not affect any function, it wouldn’t be a failure). This relation was realized as a multiple object field in the start of development. But when it became possible to affect more than one function per failure mode this proved to be more complicated than it should. Multiple object fields are not the most pleasant data type to work with in Uptime, and it became easier to use the parent-child relation for the affected functions instead.
This relation is implemented as a new child superobject on the failure mode for each of the functions that it affects. These child superobjects should only refer to functions on the same subsystem as the failure mode, however the Uptime database has no internal checks for this so that constraint cannot be implemented at that level. This changes the parent-child-hierarchy as well.

6.2.2 Next level effect
Each failure mode has a next level effect. However the next level effect is mission phase specific. The next level effect is an FMEA specific field that points out which higher level failure mode becomes active if the current failure mode becomes active. This relation is a 1-n relation (each failure mode can only have one next level effect per mission phase) and is realized as a simple object field referring to the superobject of the next level effect.

6.2.3 Manufacturers
A component has a manufacturer, even if it is made in-house. Registering which manufacturer makes the component is important, but doing it with a plain text field would be unnecessary since most manufacturers are used more than once. We add manufacturer objects to the data model and relate to them from subsystems by the use of a simple object field (which will create a 1-n relation to an already existing manufacturer object). This also provides an easy way for us to search for all components made by a certain manufacturer.

6.2.4 Phrases
FMEA tables often contain a lot of textual data, but a lot of it is nothing more than standard phrases that are repeated in several places. The uptime BPC platform has out of the box support for “standard phrases” which are a single textual string that is linked to from several places. Using these in the FMEA relieves the user from typing the same phrase over and over again (and provides a single point where all of the phrases used can be changed at once). However, this requires that the object that uses the phrase links to it. Thus the objects in the hierarchy which contains data that could contain phrases (subsystems, failure modes and mission phase specific data objects) can all contain links to phrases if they are using them in one of their fields.

6.2.5 Ports and connections
The subsystems and functions can each contain a number of ports, which in turn can be connected to other ports. These ports are represented by their own separate ConnectionPort object type, which can link to other ConnectionPorts by the normal uptime links. Links was chosen here instead of any of the other types since they are easier to work with than object fields, and they support multiple targets (a single link source can have multiple targets). Thus connections were implemented as links to other ports. Each port is a link source, and this link source can have several link targets (connections).

6.2.6 Mission profile/phase objects
Mission profiles is something that is central in creating an FMEA and it is represented as an object that contains references to one or more mission phases (along with extra information about how much time is spent in the mission phase). Each mission profile has one or more MissionPhaseInfo objects as children which in turn refer to a mission phase object that contains information about the actual mission phase. This referral is done via a single object field, since it is a simple 1-n relation.
6.2.7 FMEA and Model objects

Each FMEA is represented by a FMEA object that contains a reference to the top subsystem and the mission profile that contains the information that this FMEA contains and each model is represented by a model object that contains a reference to the top subsystem that the model contains. The top subsystem is, however, in both these cases not the actual root node for the system tree. As legacy from the time when subsystem instance nodes was in the system (see section 6.1.1.1, A first attempt at the reuse of components, on page 48) the real root node of the system tree is actually the top subsystems first child. This does not really affect anything in practice, although some implementation depends on the tree looking like this. In the future, this should be refactored out. For information about how the FMEA object relates to the Model objects, see chapter 7.3, Connecting the model and the FMEA on page 61.

6.3 The final data model

![Diagram showing the final domain model](image)

Figure 41: The final domain model

The final domain model, with all the links and relations drawn into it, is shown above in Figure 41.
7 Implementation

7.1 Model class hierarchy

From the beginning it was obvious that working directly against the database was a bad idea. First of all, it would have been very inefficient and would have led to the user having a non-responsive user interface, but it would also have been much harder to implement features such as “undo”. So, a cache class hierarchy was needed for the database objects. This cache class hierarchy could then be used to wrap most of the database calls, which in turn leads to that much of the source code could be kept clean of database calls. The class hierarchy can be viewed below in Figure 42.

As can be seen, the class hierarchy is designed to follow a slightly modified composite pattern, the subsystem and function classes both implement the ICacheItemContainer interface allowing them to contain other instances of SubSystems and Functions. The SubSystem class is the only one allowed to contain FailureModes, and FailureModes in turn contain one MissionPhaseData object per MissionPhase in the currently active MissionProfile. A ConnectionPort can be put on any implementer of the IItemWithPorts or ICacheItemContainer item and then refers to the ports that connect to it.

Since C# is used as the implementation language, implementing the observer pattern is very easy. All properties of the classes have support for being listened to.

7.1.1 Connections

Most of the interesting information that can be calculated from the model comes from analyzing how everything is connected.
A component is said to be “on top” of another component if the component on top is parent to the component “beneath” it. In Figure 43 the component named “Top” is on top of all other components in the figure, the component named “Component A” is on top of “Component A1”, and “Component A1” is not on top of anything. “Siblings” are defined as two objects sharing the same parent. “Component A1” and “Component B1” are siblings in Figure 43.

A port can be connected to any sibling port of the opposite direction. This is called a “same level connection”. It can also be connected “up” to a port in the same direction of the parent, or it can be connected downwards to a port of the same direction of any of the components children. In Figure 43 there are examples of all of these kinds of connections from Component A’s viewpoint. This, of course, means that a down-connection for one component is an up-connection for another. The connection between Top and Component A is an up-connection when viewed from the viewpoint of Component A, but a down-connection when viewed from the viewpoint of Top. The connections are then stored in the port objects in the database.

7.2 FMEA Module

The FMEA module was the first module implemented when work on the thesis begun. A screenshot of the default view of the FMEA module can be seen below in Figure 44. Apart from this view there is also a plethora of dialogs used for input of different kinds of information into the program. Most of the information can be entered via the grid view, but when creating new components, functions or failure modes you have to do it via a dialog.
interface. The entire FMEA module interface makes heavy use of the observer pattern, reacting to events fired by the model class hierarchy (previously discussed in chapter 7.1).

Figure 44: A screenshot of the standard FMEA view in Uptime Engineering

7.2.1 Grid configurations

As said before, no FMEA is completely alike the other one. Because of this, it must be possible to configure the grid view to have the column set the user is used to. This is taken care of by the GridConfiguration class that uses the Memento pattern to store the information needed to order, hide and unhide columns to a certain specification. This class also uses the C# DataContract[5] feature to save these GridConfigurations to disk for later use.
7.2.2 Importing Excel FMEA’s
A system FMEA often contains several sub-FMEA’s. A part manufacturer might provide FMEA’s for its parts, and these should be integrated into the larger system FMEA.

For Uptime Engineering it was decided to start with supporting import from Excel FMEA’s. FMEA’s can be written in many different programs, and Excel is only one amongst many, but one has to start somewhere. The import of FMEA’s is, as with any import, a complicated and delicate procedure. There are no predefined FMEA formats, and not every FMEA is formal enough to refer to the same object with the same name. For instance, a failure mode could have the name “No flow”, while when it is referred to as a next level effect it could be referred to as “No flow of oil in compartment” or “Oil does not flow”. Matching the next level effect text to the failure mode name requires the program to understand the meaning of the name, which is something that the researchers of natural language processing has been dreaming about for years. In the Uptime Engineering import function we recognize that this problem cannot be solved in a good way with the technology of today and instead take a more pragmatic approach to the problem.
Figure 46: Importing can create "false" failure modes which requires merging with "real" failure modes

When importing an FMEA, if we do not find the failure mode that the next level effect refers to we instead create a new failure mode with that name. This means that if the FMEA that is being imported has referred to failure modes by some other means than its name, we will create "fake" failure modes. For example, in Figure 46 the imported FMEA refers to the failure mode "No flow" with the more verbose sentence "Oil does not flow", and thus two failure modes will be created instead of one. The user will then need to merge the failure modes "No flow" and "Oil does not flow" after the import. When the user previews the import he then has the option to merge a pair of failure modes into a single one (see Figure 47 and Figure 48 for examples on how merging failure modes can look). This way the user can merge the "fake" failure modes with the real ones and get the correct import of the FMEA. This, of course, is not automatic. But with today’s state of natural language processing this approach gives the best results (and it is still preferable to doing it totally unaided).
Figure 47: Merging two failure modes can be very simple if they are both "fake" failure modes, or if one of them is "fake" and the other is real.

Figure 48: Merging two "real" failure modes is also possible, although this leads to more work.
As seen above in Figure 48, merging two “real” failure modes is also possible if the user so desires. However this produces far more conflicts that need to be resolved before the operation can proceed.

Of course, the introduction of “fake” failure modes is not the only problem that arises when importing FMEA’s. As said earlier, there is no exhaustive list of “standard columns” for FMEA’s, and even among the columns that are considered to have to be in an FMEA there is no standard order, or standard name to match. Thus we require user input in order to know which column in the FMEA that we are importing to match to which column in our FMEA representation. We do this with the dialog shown in Figure 49 below.

![Figure 49: The dialog for matching columns](image)

Here we let the user drag and drop to match the columns found in the excel file with the Uptime Engineering column names. The user also has the option to save this configuration for later use; in that case it is the matching object is saved as XML using the standard C# DataContract feature [5].
7.2.3 Standard phrases

While an FMEA contains a lot of textual information, much of it is standard phrases that are repeated over and over again in many places. The Uptime platform has support for phrase-handling out of the box, and this has been fitted to work with the FMEA module. Certain FMEA fields have been selected to be able to contain phrases, and the user can, apart from writing normal text in these, drag and drop standard phrases into them. These phrases will then be linked into the text. This also provides help in translation, where a phrase can be translated only once instead of every time it shows up in a text.

7.2.4 PDF Generation

When a user has finished working with an FMEA he might need to email it to someone, print it to paper format, or for some other reason externalize it. Since everything in Uptime is saved in a database, this is in many cases not as simple as simply copying the file containing the FMEA. Instead, functionality was added to generate a PDF version of the FMEA. The PDF is generated with the third party software tool AntennaHouse XSL Formatter[6]. This tool takes as input an XML file and an XSLT stylesheet that transforms the XML file into XSL:FO[7]. In order to transform the FMEA into XML the C# DataContract-feature[5] was used.

![Diagram](image_url)

Figure 50: PDF Generation is done by transforming an XML representation of the FMEA into XSL:FO

7.3 Connecting the model and the FMEA

The main reason for introducing the modeling specification found in chapter 5 was to create relatively simple models for helping out during the safety assessment process. And in order to be of help we need to connect the model to the actual FMEA.

7.3.1 Relating the FMEA and the Model system trees

The simple way to synchronize between the model and FMEA system trees would simply have been to copy one to another. However this is not a good strategy for many reasons, the main one being that we would not have any actual relation between the resulting trees, something that can be useful for many things.

So, in order to preserve as much information as possible the relation between the tree nodes should be kept in some way. The way that was chosen is utilizing the super-subobject relation. As you might remember from earlier, a tree is made up of superobjects that record the structure, and subobjects that contain the information, this is illustrated in Figure 51 below.
Figure 51: A simple system tree.

Copying only the superobjects would create relations to the same subobjects in a completely different tree. This allows us to add, move or restructure the structure of a tree separately from the original tree while still retaining information to match the nodes with each other in an easy way. An example of this can be seen below in Figure 52.

Figure 52: The simple tree from Figure 51 cloned and related to another tree

7.3.2 Tree synchronization

When the FMEA and model trees are related by sharing of subobjects, tree synchronization can be done with an easy two-pass algorithm. Consider the tree in Figure 53 below.
Figure 53 shows an FMEA tree and a model tree related by sharing of subobjects. If we wish to synchronize from the model tree to the FMEA tree we can see that some nodes needs to be added (those marked blue), some needs to be moved (those marked green) and some needs to be deleted (those marked red). In this example it will be shown how to synchronize the model tree to the FMEA tree.

The algorithm recursively does two passes over the model and FMEA trees, first adding new nodes and moving old ones, and then in the second pass deleting the nodes that are no longer there.

During the first pass the following set of rules is applied:

- If an object has a subobject that does not have a superobject in the tree that we are synchronizing to, we need to add this object. (See for example the top subsystem in the model tree)
- If an object has a subobject that has a superobject in the tree that we are synchronizing to, we check if the subobject of the parent nodes is the same in both trees. If they are not, we need to move the node. (See for example the failuremode in the FMEA tree. This has the same parent subobject in both trees and does not need to be moved, however the subsystem that is its parent does not have the same parent subobject in both trees and thus needs to be moved)
- If an object has a subobject that does not have a superobject in the tree that we are synchronizing from (see for example the last subsystem in the FMEA tree) this node needs to be removed. However we should not do this in the first pass of the algorithm since it might have children that need to be moved. Thus we simply ignore this node in the first pass.

Applying the above rules will create the tree shown in Figure 54 below.
When the above rules have been applied to the entire tree, we enter pass two. This is the deletion pass and now we simply apply this rule to the entire tree.

- If an object has a subobject that does not have a superobject in the tree that we are synchronizing from, it is deleted.

Applying the above rule will make the tree look like Figure 55 below.

And as can easily be seen, the trees are now structurally equal, containing the same nodes in the same places.

8 Conclusion and Future Work

8.1 Conclusion
The thesis proposed a new methodology to support a model-based safety assessment process in which safety assessment analysis can be automatically derived or kept synchronized with a model representation of the system. In order to support this methodology a modeling formalism that can capture both the functional and the structural representation of the system was proposed. The models will start out being very limited in scope during the early phases and can then become more and more detailed the further you go in the safety assessment process. This thesis presented the modeling formalism and showed how the use of models generated from this formalism can help to calculate and verify certain parts of the safety assessment process automatically.

In this context we have been able to demonstrate that FMEA’s can be automatically generated from a system model. Also, FMEA’s and their model representation can be kept synchronized and the effect of the model changes can be either automatically propagated to the FMEA or propagated in a batch synchronization operation and provide valuable information to the safety engineers about the effects of such a change to the system. The thesis claims that the principles employed for the FMEA-model synchronization and the underlying modeling formalism employed to represent the system model can be used for other aspects of the safety assessment process as well (for example, fault tree analysis, reliability prediction, architecture optimization, etc.)

Early tests of this type of modeling show promising results. This very general and quite simple way of modeling complex systems has shown itself to be sufficient for performing calculation of important safety assessment data such as the next level and global effects of a failure mode. This thesis shows the specification for this modeling formalism and that it can be used to calculate several useful pieces of information that in the current work process of the safety engineers are calculated and entered manually. Because of its relative simplicity and lack of reliance on actual implementation details it should be possible make use of a model very early in the safety assessment process which could potentially save both time and money in the long run.

The modeling formalism presented is simple to transform into other representations more suitable for the current safety assessment activity (for example the FMEA view, see section 7.3). This means that the information can be presented to the user in a way that he/she is more “used to” which has the effect that even a user that is not proficient in modeling can work with the model.

During the course of this thesis a proof-of-concept implementation has been implemented that has been used to test and determine what changes needed to be done to the modeling formalism in order to make it work in a satisfactory manner.

8.2 Future work
Apart from making sure that the proof-of-concept implementation follows all details of the model specification given in this thesis, there are several things that should be taken care of.

- **Port directions.**
  In the current specification we assume that all ports have a direction. It could be either an input port, or an output port. However, this is not a realistic representation of how
connection ports work in real life. For example, a resistor has no input or output port, each of the connecting pins of the resistor could be the input or the output depending on how it is connected into the system. Ports with the direction “Any/Both” should be implemented and added to the specification and, amongst other things, the algorithm for calculating the next level effect should be updated to work for this kind of ports as well.

- **Improve the modeling environment.**
  The current modeling environment implemented is basically a textual interface with some visual aids. This makes it very hard to enter connections in models that contain more than 10 components. A graphical user interface where the user can easily get an overview of the model would ease the usability significantly (something that has been shown by entering the model semantics in MetaGME and doing tests entering models in GME based on the model semantics) and help create better tests for showing the full potential of the modeling specification given here.

- **Create a standard component library.**
  Currently a new component needs to be created for every single system component. In many systems certain components are reused many times over and over (resistors, for example). Actually having to create the same resistor over and over again is a huge drawback. It should be possible to specify that a user-created component/subsystem is a standard library component (thus making an easily extensible component library). After this is done, a standard component library needs to be created that contains a number of the “standard” components that are often used in modeling (for example, resistors, capacitors, batteries etc.).

- **Specify transforms from the model to FTA, Reliability testing and other important safety assessment techniques.**
  This thesis only concerns itself with the FMEA part of the safety assessment process, but as soon as possible work should begin on the other parts of the process as well.

- **Implement translation, versioning, user permissions and similar work process features**
  Uptime BPC has, as previously stated in chapter 3, good support for many features that is part of a work process involving several persons. During the thesis work, implementation of domain specific rules for some of these features has been disregarded due to time constraints. However, these are important and useful features and in order to support the entire safety assessment process in a satisfactory way these features should be examined and implemented.

### 9 Glossary

GME – General Modeling Environment  
FMEA – Failure Mode and Effects Analysis  
FMECA – Failure Mode and Effects Criticality Analysis  
FPR – First Point of Redundancy  
FTA – Fault Tree Analysis  
RPN – Risk Priority Number  
XML – Extensible Markup Language  
XSL – Extensible Stylesheet Language

### 10 Bibliography

[10] Certification considerations for highly-integrated or complex aircraft systems, Society of Automotive Engineers Inc, 1996
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida [http://www.ep.liu.se/](http://www.ep.liu.se/)

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: [http://www.ep.liu.se/](http://www.ep.liu.se/)

© Fredrik Forssén