NTNU – Trondheim
Norwegian University of
Science and Technology

CHASSIS Tool

A model-driven approach

Juni Angelfoss

Master of Science in Informatics
Submission date: May 2014
Supervisor: Guttorm Sindre, IDI
Co-supervisor: Christian Raspotnig, IFE

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Safety and security, and unifying these two aspects are important when eliciting requirements to form both new and existing computer systems. As a means to support this, the CHASSIS method was developed. With this a new issue emerged: a need for a computer tool supporting CHASSIS during requirement elicitation. This report presents an initial approach to a tool for CHASSIS, focusing on generation of the Failure Sequence Diagram (FSD). Included in the report are CHASSIS' background, related safety and security techniques and their limitations compared to CHASSIS. Further on, different approximations to a CHASSIS tool are presented resulting from industry input. The background, related techniques and input together form a proposed solution. The report continues presenting requirements of the tool as well as how these where implemented programmatically. An investigation of the tool and its abilities to fulfil the stated functional and non-functional requirements was carried out through a student experiment. The results of the experiments and other techniques utilised during the project is then evaluated and validated. A short summary and notes on further work concludes the report.

Preface

Denne masteravhandlingen er den avsluttende delen av et to-årig masterstudie innenfor Informatikk med spesialisering innefor retningen informasjonsforvaltning ved Norges teknisk-naturvitenskapelige universitet (NTNU). Arbeidet med oppgaven begynte i august 2013 og ble ferdigstilt i slutten av mai 2014.

Jeg vil her benytte anledingen til å takke min hovedveilder Guttorm Sindre for avgjørende veiledning jeg mottok under arbeidet med oppgaven. Jeg ønsker også å takke min biveileder, Christian Raspotnig, for innsikten han gav meg til CHASSIS som metode, men også for gode tips til utformingen av et verktøy for CHASSIS. Denne oppgaven hadde ikke vært til hadde det ikke vært for ham. En takk sendes også til Avinor som tok meg godt imot og bidro med verdifull informasjon om tiltenkt bruk av et CHASSIS verktøy.

Min familie, og da speiselt mine kjære foreldre, fortjener også en stor takk for støtte gjennom hele prosessen.

Contents

List of Acronyms

BDMP - Boolean-logic Driven Markov Process

CHASSIS - Combined Harm Assessment for Safety and Security of Information Systems
CTA - Concurrent Thinking Aloud
CORAS - no abbreviation

D-MUC - Diagrammatical Misuse Case
DSL - Domain Specific Language
DSR Design Science Research

EMF - Eclipse Modeling Framework
EMP - Eclipse Modeling Project
ETA - Event Tree Analysis

FHA - Failure Hazard Assessment
FMEA - Failure Mode and Effect Analysis
FR - Functional Requirements
FSD - Failure Sequence Diagram
FTA - Fault Tree Analysis

GEF - Graphical Editing Framework
GMF - Graphical Modeling Framework
GTBS - Goal- or Task-Based Scenarios

IDE - Integrated Development Environment
IDI - Institutt for datateknikk og informasjonsvitenskap
IFE - Institutt for Energineknikk

IS - Information System

KAOS SE - Keep All Objectives Satisfied/Knowledge Acquisition in Automated Specification Security Extension

HAZOP - Hazard and Operability Study

MBSA - Model Based Safety Analysis
MDSD Model Driven Software Development
MDT - Model Development Tools
MOF - Meta-object family
MUC - Misuse case
MUSD - Misuse Sequence Diagram

NTNU - Norges Teknisk-naturvitenskapelige universitet

OMG - Object Management Group

RP - Retrospective Probing

SD - Sequence Diagram
SdEdit - Quick Sequence Diagram Editor
SDL - Security Development Lifecycle
ST - Secure Tropos

T-MUC - Textual Misuse Case
TEA - Threat Effects Analysis

UC - Use Case
UCD - Use Case Diagram
UML - Unified Modeling Language

VSU - Visual Studio Ultimate

List of Figures

List of Tables

Chapter 1

Introduction

This chapter serves as a presentation of the work conducted and documented in this report. It begins with a background of the current situation, providing the context of study in this report.

1.1 Background

Safety, defined by the Oxford Dictionary as *condition of being protected from or unlikely to cause danger, risk, or injury*[11] and security, *the state of being free from danger or threat*[12] are two central terms in functional requirement elicitation area. Techniques in the safety field, including the Hazard and Operability Study (HAZOP), Failure Mode and Effect Analysis (FMEA), Functional Hazard Assessment (FHA) and Fault and Event Tree Analysis (FTA/ETA), have remained more or less the same the last couple of decades.[3] In the security field however, techniques such as Secure Tropos (ST), KAOS Security Extension (KAOS SE) and CORAS tend to be based on more modern modeling languages.

Property	CHASSIS	HAZOP	FMEA	FTA/ ETA	FHA
Can cover safety aspects	yes	yes	yes	yes	yes
Can cover security aspects	yes	yes	yes	yes[1]	-
Unifying safety and security aspects	yes	-	-	-	-
Graphic analysis visualisation	yes	-	-	yes	-
Address interaction between components	yes	-	-	-	-
Modeling multiple failures	yes	-	-	yes[2]	-

Table 1.2: Overview of CHASSIS compared to popular safety techniques

Despite including a wide range of options, very few techniques in the safety and security field provide the means to or aim at unifying safety and security aspects when performing system modeling activities.[4] A method aiming at providing just such an approximation, and thereby give a more nuanced view of the system under analysis, is the Combined Harm Assessment for Safety and Security in Information Systems (CHASSIS) method. Table 1.2 and 1.4 presents an overview comparing CHASSIS to other popular techniques from the safety and security field, respectively. These techniques are further discussed in section 4.1-4.3.

One advantage with CHASSIS is the fact that the method utilizes the Unified Modeling Language (UML). This provides means to visualize how failures in a system can propagate (safety) and how an attacker can exploit the vulnerabilities in a system (security). In CHASSIS, these two aspects are presented by a Failure Sequence Diagram (FSD) and a Misuse Sequence Diagram (MUSD), respectively, both based on the UML Sequence Diagram (SD). FSD and MUSD have both been applied in experiments, presented in [2, 4, 5, 6]. The same papers presents the problem addressed in this report: approximating a tool for CHASSIS with the focus on the Failure Sequence Diagram (FSD)

[1] Only FTA[3]
[2] Only FT (see 4.1)

Property	CHASSIS	ST	KAOS SE	CORAS
Can cover safety aspects	yes	-	yes	yes
Can cover security aspects	yes	yes	yes	yes
Unifying safety and security aspects	yes	-	-	-
Graphic analysis visualisation	yes	yes	yes	yes

Table 1.4: Overview of CHASSIS compared to popular security techniques

1.2 Tool need

The need for a CHASSIS tool facilitating the generation of FSDs emerges in several papers describing conducted CHASSIS experiments. Here, it is suggested that a tool for the FSD would give the diagram more structure and that this structure would facilitate the collection of all relevant information directly in the FSD. Yet another issue during the experiments was related to drawings becoming to complex and taking up too much space on the whiteboard. Creating a computer tool can provide solutions to support this and other issues related to whiteboard-drawings; storing and sharing files, simplifying diagram editing, structure in large and complex diagrams, facilitate common understanding of the system through a common notation set and collaboration possibilities.

An investigation of existing modeling tools (see section 4.4) reveals that none of these encompasses the functionality need for a tool approximation to CHASSIS, further justifying the tool development effort presented in this report.

1.3 Research questions

In this report, the research questions revolve around approximating a tool for CHASSIS, approximating being the stressed word. The work conducted and described in this report has been performed by one person during the course of a singe academic school year. For these reasons, this report only means to present a first approach to a CHASSIS tool and not a completed product.

The fact that the CHASSIS method has not been included in any previous attempts at tool creation, leads to two interesting research questions:

1. *What approximations are there to a CHASSIS tool?*
2. *Does such an approximation work in a realistic setting?*

Methods and activities supporting the realisation of the CHASSIS approach, and means to answer these two research questions are presented in section 1.4.

1.4 Research and development method

This report utilizes several methods and activities to support the CHASSIS tool approximation. The main research method, Design Science Research (DSR), provides structure guiding the developer through the the process resulting in a novel artefact: the CHASSIS tool. In addition, the iterative development process method supports the main research method by allowing further decomposition of the phases in DSR. The Model Driven Software Development (MDSD) is yet another supporting modeling method applied in this report. MDSD offers metamodeling, a technique used to create the domain specific language (DSL) for the first attempt at a CHASSIS tool approach.

Deciding on a development platform is based on a CHASSIS background study and a set of high-level requirements provided by a CHASSIS domain expert, Christian Raspotnig. The functional requirements of the tool to be developed on the chosen platform are results of the CHASSIS background, related papers and input from the aviation industry and Raspotnig. In addition, the OMG (Object Management Group) UML superstructure supported the election of metaclasses to be extended in the FSD DSL. The tool analysis was performed by conducting student experiments, by the means of a usability testing technique.

1.5 Structure of the report

This section aims at giving an overview of the contents of the chapters throughout this report.

Chapter 1 presents the background, identifies the tool need and research questions this report addresses.
Chapter 2 describes the main research method, as well as supporting methods and activities applied during the development process.
Chapter 3 gives a detailed background on the CHASSIS method.
Chapter 4 compares the CHASSIS method to other popular safety and security techniques and takes a look at existing modeling tools.
Chapter 5 presents the problem and proposal and the intended solution.
Chapter 6 discusses the platform selection.
Chapter 7 presents the implementation of the CHASSIS tool approximation.
Chapter 8 documents the conducted student experiment with the CHASSIS tool and its results.
Chapter 9 evaluates the methods and techniques used during the development period and answers the research questions.
Chapter 10 draws a conclusion on the work presented in the report and suggests further work.

Chapter 2

Research method

This chapter provides an insight to the main research method used in this project, as well as supporting methods and activities.

2.1 Main research method - overview

The design science research (DSR) involves creating new knowledge. The creation is realised through *design of novel or innovative artifacts (things or processes that have or can have material existence) and analysis of the use and/or performance of such artifacts along with reflection and abstraction.*[37] The purpose of the research method is to both improve and understand behavioural aspects of Information Systems (IS).

In this particular project, the artefact is a new tool: The CHASSIS tool. As a method, DSR is conveniently organised into four phases;

1. Artifact design
2. Construction
3. Analysis
4. Evaluation

These are presented in further detail, along with appropriate activities supporting the development of the CHASSIS tool, in sections 2.2-2.5.

2.2 Artifact design

The initiator to the DSR is a problem. The problem may originate from several sources, here from reference disciplines. [6, 4, 2, 5] and the supporting doctorate dissertation, [3], all enlighten the matter: the need for an artefact covering the CHASSIS method that aid experts in requirement elicitation activities.

The result of the problem awareness step is a formal or informal proposal, containing a new research effort. Here, the proposal suggest creating a tool for CHASSIS as a mean to support the method. Figure 2.1 presents the suggestion step, also known as the creative step, as the recipient of the proposal. The proposal is then investigated in further detail and the a design sketch for the artefact, the solution to the problem, is created.[38] Now having covered the problem, proposal and a solution, the CHASSIS tool design and how to implement it needs to be described in further detail. Here, three design activities, choosing a development approach, gathering tool requirements and choosing software development method support this. These activities are covered in subsections 2.2.1-2.2.3.

2.2.1 Design activity - choosing a development approach

In software engineering, a development method is a framework used to plan, structure and control the information system process.[45] As the initial plan of a project, especially with respect to the requirements, often is flawed and therefore in need of editing an iterative development approach is preferred.[32] Figure 3.1 gives a graphical presentation of a single iteration in a iterative development cycle: planning (see Chapter 5), requirements, analysis and design and implementation (see Chapter 7 and 6), test (see Chapter 8) and finally evaluation (see Chapter 9). Comparing the iterative development methodology in Figure 3.1 to the stages of DSR, it is evident that they have a great deal of overlap. Using DSR as a main method and an iterative support method gives the artefact developer opportunity to return and to previous steps in the cycle several times during development should factors defining the tool need editing, as well as aid in the decomposition the activities in each step of DSR into manageable sized tasks.

2.2.2 Design activity - gathering tool requirements

In the world of software development, there are fields of opportunities related to the gathering of system requirements, [39, 40, 41] and [42] to mention a few. Most of these, however, focus solely on user interaction during the elicitation process. In this project, the elicitation used has a more theoretical centered approach, based on investigating the CHASSIS method (see Chapter 3), related modeling tools (see section 4.4), input from domain experts (Raspotnig), the formal OMG UML structure[30] and [6, 4, 2, 5, 43, 44], representing articles where the CHASSIS method has been evaluated. These sources together gives a holistic and nuanced view of CHASSIS and the requirements related to approaching a CHASSIS tool, and thus provides sufficient information for the extraction of functional as well as nonfunctional requirements. Here, the functional requirements covers how the artefact should behave, specifying what is needed for the development. The non-functional, or user-, requirements describes user expectations and enlightens how the user will interact with the artefact.[64]

2.2.3 Design activity - choosing software development method

Central in the selection a sub-method to aid in the software development is the list of functional and nonfunctional requirements (see Chapter 5). As it suggests, the CHASSIS tool can be realised through metamodeling, one of the most important aspects of the Model-Driven Software Development method (MDSD).[7] As the name suggests, a metamodel is a model that can make statements about modeling.[7] In this project, a metamodel of FSD, detailed in a Domain Specific Language (DSL), is created through the extension of the unified modeling language (UML) model (see Chapter 7).

2.3 Artifact construction

As the title suggests, the second phase of DSR involves implementing the design developed in the previous phase. Here, the techniques for implementation will vary depending on what type of artefact is to be created.[38] For instance, if the artefact is a system, software development is a natural construction activity and if the artefact is an algorithm, the construction activity would involve the development of a formal proof. In this project the artefact to be created is, as previously mentioned, a tool, and the construction phase would imply software development. The subsection describes a final activity conducted before the actual implementation of the CHASSIS tool can take place: choosing a development platform.

2.3.1 Construction activity - choosing a platform

Choosing a development platform is a crucial decision in developing the CHASSIS tool. Here, demands the platform needs to address and meet are not only a set of high level requirements, but functional and non-functional requirements as well. Failing to meet these could result in the project failing. Additional important factors that should be evaluated when a development platform is chosen are the developers familiarity with the platform, but also supporting information related to the platform including tutorials and other publications. One platform addressing these factors are the Eclipse IDE (integrated development environment) and one of its available plugin tools: Papyrus. Both of these are further described and discussed in Chapter 6.

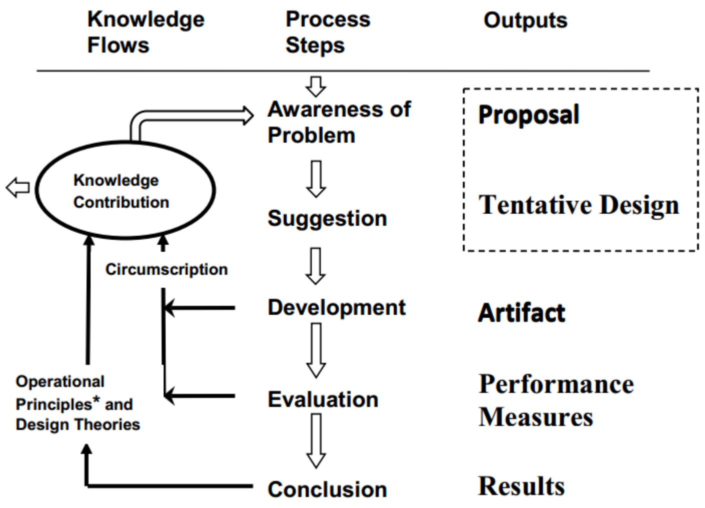


Figure 2.1: Graphical representation of the design science process model[37]

2.4 Artifact analysis

When the software development has taken place and an artefact has been developed, the third phase, the artefact analysis, takes place. Here, the artefact is evaluated according to the elicited requirements (see Chapter 7). Any deviations from expectations discovered here are noted and must be tentatively explained.[38] In other words, this is where the hypothesis, which in this project is represented by the second research question (see section 1.3), are either confirmed or contradicted. The evaluation of the CHASSIS tool approximation is, in this project, evaluated through the means of an activity called usability testing.

2.4.1 Analysis activity - usability testing

According to [51], usability testing involves the evaluation of a product or a service with representative users. As a general, the purpose of this activity is identifying any useability problems, collecting data and determine the participant's level of satisfaction with the artefact. A successful test-execution requires solid preparatory work: developing a well thought out test plan and acquiring participants who satisfy a set of properties. Here, the test plan documents what parts of CHASSIS tool is being tested, how the test is going to be conducted and details regarding what data is to be collected. Additionally, the plan describes scenarios the participants are to complete via the artefact, including requirements supporting these scenarios. As the testing is completed, the data collected needs to be analyzed and findings reported..

2.5 Evaluation

The evaluation phase of DSR, in this project, represents the finale of a research effort.[38] The results of the study performed in the artifact analysis phase are discussed and evaluated. In this project, the evaluation additionally covers all methods and supporting activities performed during the project. The evaluation is further detailed in Chapter 9.

Chapter 3

Background (CHASSIS)

This chapter presents the Combined Harm Assessment for Safety and Security in Information Systems (CHASSIS) method in detail, focusing on how CHASSIS is applied and two of the methods visualization utilities: failure sequence diagram (FSD) and misuse sequence diagram (MUSD).

3.1 What is CHASSIS?

The CHASSIS method is a unifying process, joining both assessment of the safety and security fields. This results in a method covering both harm and attack identification and analysis.[4] In an iterative development project, a project consisting of several consecutive system development life cycles[32], CHASSIS belongs mainly to the software development stage covering requirements activities (the blue arrow in Figure 3.1).[33, 5] At this stage, the functional and physical needs related to how a product must be able to perform are stated.[34] Here, the CHASSIS method, through team activities, enables visual modeling and a structured harm assessment to support the elicitation of safety and security requirements based on the functional needs.[4, 5] More active usage of models during the assessments of safety and security can give several benefits. One of them is improved understanding and better discussions of the system that is under assessment.[3]

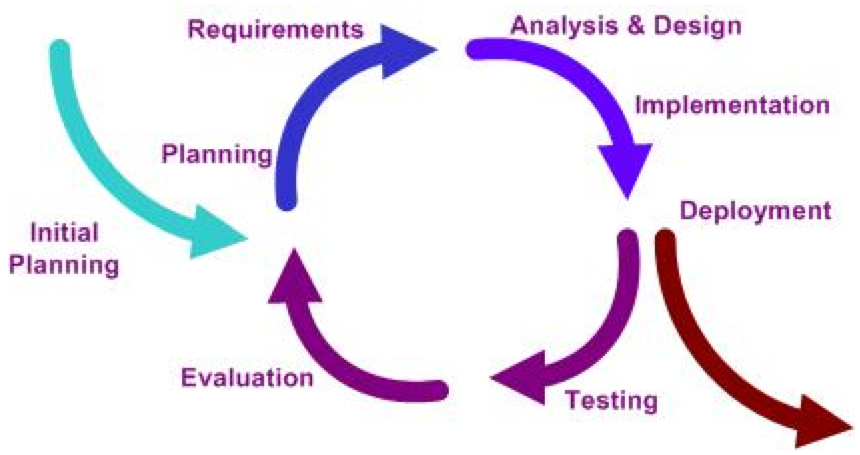


Figure 3.1: A single iteration in an iterative development life cycle[76]

The visual modeling part of CHASSIS covers both security and safety aspects of the requirement elicitation development stage. The security features includes Textual and Diagrammatical Misuse Case (T- and D-MUC) and the MUSD, whereas T-MUC and D-MUC is used alongside the FSD in the safety field.[4] (see sections 3.2-3.4) The harm assessment functionality is realised by utilizing HAZOP.(see section 4.1.1) In the CHASSIS setting, HAZOP takes the role as the creativity enhancer, allowing a structured elicitation process and specifying the requirements based on the safety and security output from CHASSIS itself.[4] Together, MUC, FSD, MUSD and HAZOP is combined into a process; the CHASSIS process which is covered in the next section.

3.2 The CHASSIS process

As Figure 3.2 shows, the CHASSIS process is separated into three stages. First stage, the elicitation of functional requirements, covers the definition system functions and services. The system definition can be performed in three different ways. The first one, using operational and environmental descriptions of the system. The second one is realised through discussion with stakeholders, and the third one using both former approaches in combination.[4] To further aid the elicitation, UML diagrams are included at this stage. One of them is a Usecase Diagram (UC). Usually detonated as a behaviour diagram, a UC describes a set of actions a system can perform in collaboration with one or more external users.[35] Additional description of the UC's is covered in the Textual Usecases (T-UC). The other UML diagram included in the first third of the CHASSIS process is Sequence Diagram (SD).[33] As a common interaction diagram, SD is focused on message flow between lifelines.[36] Here, a lifeline represents a single component of a system.

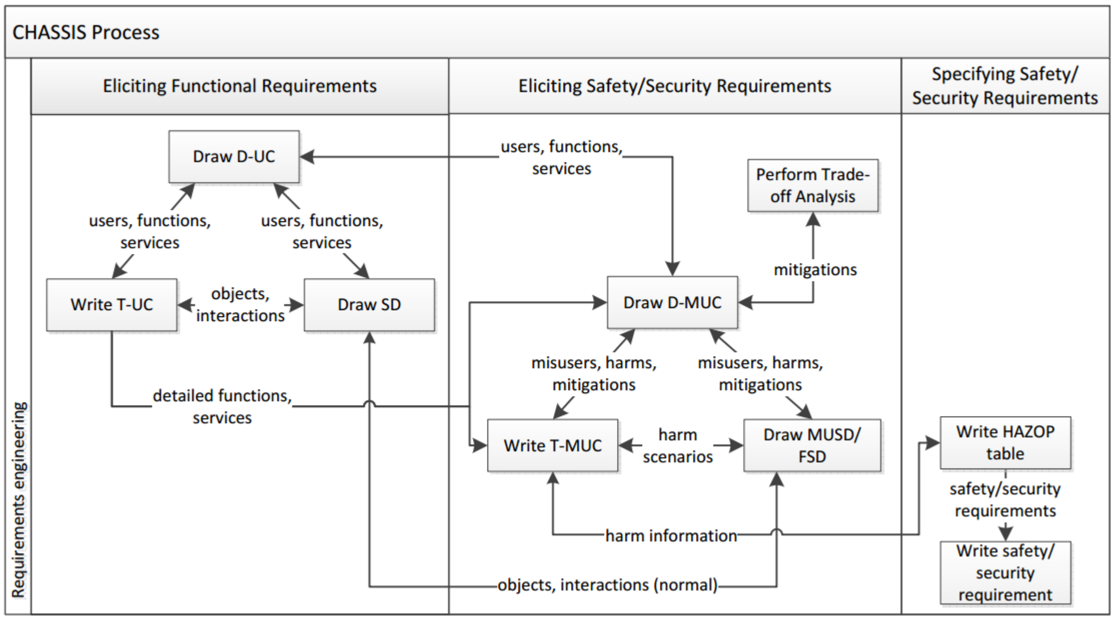

CHASSIS Process
Requirements engineering
Eliciting Functional Requirements
Eliciting Safety/Security Requirements
Specifying Safety/ Security Requirements
Draw D-UC
users, functions, services
users, functions, services
users, functions, services
Write T-UC
objects, interactions
Draw SD
detailed functions, services
Perform Trade-off Analysis
mitigations
Draw D-MUC
misusers, harms, mitigations
misusers, harms, mitigations
Write T-MUC
harm scenarios
Draw MUSD/ FSD
harm information
objects, interactions (normal)
Write HAZOP table
safety/security requirements
Write safety/ security requirement

Figure 3.2: The CHASSIS process[33]

The elicitation of safety and security requirements are the central activities in the second CHASSIS process step, performed by creating MUC, T-MUC, FSD and MUSD. MUC are created based on the UC from step 1, by extending it and adding notation to the UC indicating possible misusers (mischievous external users) and harm, and the MUC if further detailed through the creation of T-MUC. The T-MUC then provides a list of harm scenarios which are refined in the FSD and MUSD. If any new relevant information is discovered when refining the sequence diagrams, this is fed back to the T-MUC, and new mitigations are defined as new usecases. The mitigations will then initiate a new iteration starting at the first step in the CHASSIS process. When the T-MUC is completed, HAZOP tables are prepared, the corresponding security and safety requirements are defined, thus completing the third and final step in the CHASSIS process.

3.3 Failure Sequence Diagram

At the same time as the first approach to a CHASSIS tool was developed, the long term scope was kept in mind. The natural next step after the completion of the functionality for a FSD editor, would be adding new functionality to the tool: the means to create MUSDs. Because of this, and because the plan for a short time was to create a tool for both methods simultaneously, MUSD is covered in detail equally to FSD in the next section. But first of is the Failure Sequence Diagram.

As mentioned in section 3.2, the FSD is a graphical diagram utilized in the elicitation of safety and security requirements step of the CHASSIS process. The diagram is the result of the adaption of the security field of MUSD into the safety engineering field.[2] Here, FSD poses as a modeling technique for detailed safety assessment, thus facilitating failure analysis and the modeling of unintentional system failures.[2, 3, 4] The failure analysis feature is realised by developing the FSD with the Sequence Diagram (SD) of UML as a base and adding supporting functionality.[3]

In practise, FSD offers an overview of the system under evaluation. The overview covers components of the system, details related to how these components interact (both of these SD functionality) and relevant failure effects (additional FSD specific functionality).[6] In short, FSD focuses on the identification of how components of a system can fail and how the failures propagates through said system.[2, 6] During an analysis, FSD is created in three steps:[3]

1. Drawing normals interactions in a SD
2. Brainstorm for failures and include these visually in the diagram drawn in 1.
3. Brainstorm for mitigations, relate these to the failures identified in 2. and include them visually in the diagram drawn in 1.

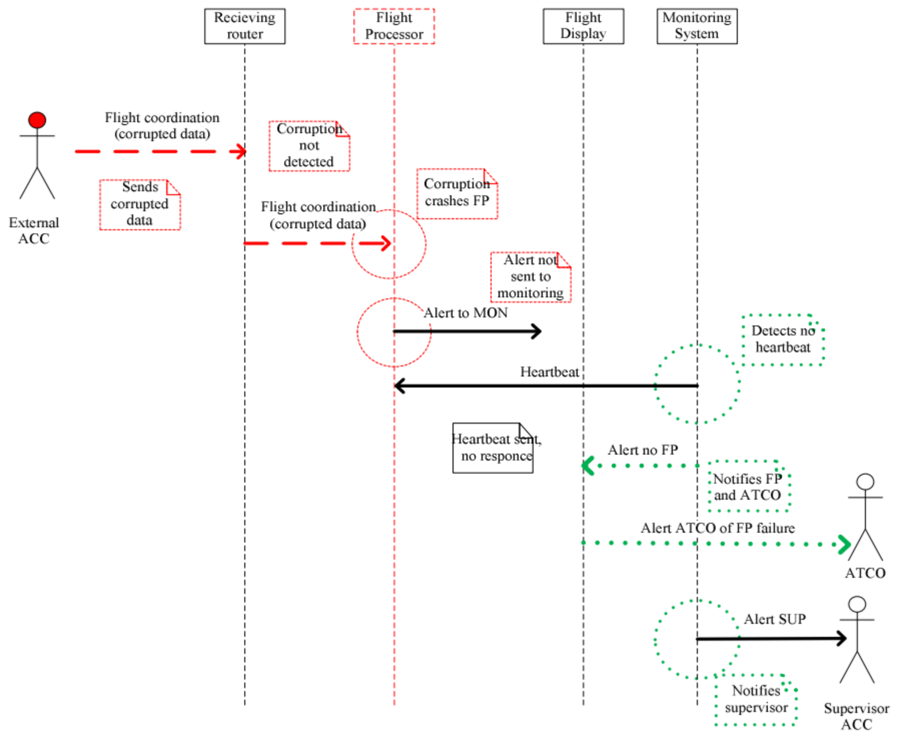


Figure 3.3: Example use of the FSD[2]

The overview of the system, its components and their interactions provided by FSD can help increase an analysts understanding of the system to be analysed.[6] In addition, the FSD notation enhances creative thinking during a system analysis.[2]

3.4 Misuse Sequence Diagram

As with FSD, the Misuse Sequence Diagram (MUSD) is a modeling tool utilized in the second phase of the CHASSIS process. MUSD is a modelling technique at system level that can be employed in assessment of security, as opposed to FSD that covers safety aspects.[2] The technique is inspired by Misuse Cases (MUC)[3], redefining its notation and combining this with the notation of the UML SD, in addition to MUSD specific notation. Here, the additional MUSD notation represents visualization of the steps attackers take against system components by exploiting vulnerabilities.[5] When modeling with MUSD, the diagram relates components in a system and displays interactions between an attacker and and these components, resulting in an overview of attacker-sequences. The MUSD focus is therefore on vulnerabilities and exploit events, as apposed to FSD that aims at modeling unintentional system failures.[2]

Chapter 4

Related techniques and modeling tools

This chapter presents techniques and modeling tools related to the CHASSIS method. It focuses on representing the techniques and tools, and comparing them to CHASSIS. The fist and second section covers safety and security techniques, respectively. The third section presents techniques that are subject of cross-fertilization and techniques used in both the safety and security field. The final section covers example modeling tools.

Common for both the safety and security field is the importance of communicating associated aspects amongst stakeholders during system development. Failing to do so could result in serious errors and useless systems. Another commonality of safety and security aspects are that they both are concerned with enlightening how a system can fail to perform.[2]

4.1 Safety

Techniques for the identification and analysis of safety aspects has remained the same the last couple of decades, the field continuing relying on established and traditional techniques. Examples of such techniques are Hazard and Operability study (HAZOP) and the Failure Mode and Effect Analysis (FMEA).[2] These two, Fault and Event Tree Analysis (FTA and ETA) and the Functional Hazard Assessment (FHA) are the topics for the next subsections.

4.1.1 Hazard and Operability Study

HAZOP started out as a safety technique for the chemical industry, but its use has since been generalised. Today the technique is included in many industries who are addressing safety as a part of their products.[3] Executing HAZOP starts out with a block diagram of the system that is to be analysed.[1] A block diagram serves as an overview of the principal parts or functions of the system, including links between the blocks representing relationships.[46] Selecting a single block in the diagram, HAZOP is applied to that block by combining predefined guide words with a set of parameters as shown in Table 4.1.[3] Based on this activity a set of possible hazards are collected and structured in a HAZOP worksheet. This worksheet serves as both documentation of the hazards and as a guideline for hazard discussions.[6]

Parameter/ Guide word	More	Less	None
Time	to long/to late	to short/ to soon	sequence step skipped
Start-up/shut-down	to fast	to slow	
Level	high level	low level	no level

Table 4.1: Guide words and parameters, and how they are combined in HAZOP

HAZOP is especially appropriate in the analysis of new systems, that is systems still in the planning or design phase.[1] Additionally, the quite systematic process for communicating and collecting information approach that is HAZOP provide good coverage of hazards.[4, 1] As a result, HAZOP table is included in the third step of the CHASSIS process (see section 3.2), to help sum up information about harm.[5] As a stand-alone technique however, HAZOP is quite limited in terms of graphical visualisation. It uses models as input just like the CHASSIS method, but unlike CHASSIS, HAZOP tends to utilize worksheets to document and discuss hazards.[6]

4.1.2 Failure Mode and Effect Analysis

The Failure Mode and Effect Analysis technique (FMEA) started out as as a reliability analysis for the U.S military. FMEA has, since then, evolved and become a widely used safety technique.[3] This technique is, alongside with i.e HAZOP (see section 4.1.1) one of the traditional methods for hazard analysis.[1] The main focus of FMEA is threefold: the identification of failure modes of components in a system, the effects of these failure modes and finding the factors that are causing failures.[6] As a result, FMEA is often complemented by FTA (see section 4.1.3).[2] In executing FMEA, the analyst reviews as many components, assemblies and subsystems as possible.[47] Then for each component the failure modes found during the review and the resulting effect they have on the rest of the system is listed in a specific FMEA worksheet/table.

In [2], it is investigated how FSD can be used to support FMEA. During the case study, the participants agreed that the optimal use of the FMEA worksheet was to structure the safety analysis-process. The participants also preferred using the worksheet when brainstorming failure modes, whereas when using FSD alone, this tended to be neglected. The worksheet was in addition used to structure discussions, and thus to ensure that the local and system effects of a failure represented in the FSD where specified and agreed upon. Here FSD posed as an overview of the system, giving participants the means to physically point at a specific component in the system. The conclusion therefore stated that it would be most beneficial using FSD and FMEA in parallel.

During safety assessment, FMEA offers a systematic process for communication and the collection of information. The technique relates failure modes to both the system component and the complete system. However, FMEA does not address interactions between components[6], a functionality covered in both FSD and MUSD in the CHASSIS method. (see sections 3.3 and 3.4) Another weakness of FMEA is related to failure propagation. The technique has no support for how a failure propagates through the system other than reasoning about the failure mode effect locally or system wide.[3] The propagation between interacting components in a system is covered diagrammatically in CHASSIS, for instance through FSD (see section 3.3).[6] Yet another weakness with FMEA becomes visible when assessing multiple failures. This situation is not supported by FMEA and was not supported graphicly in the first edition of FSD either (see section 7.2), but was included in the second version (see section 7.3).

4.1.3 Fault and Event Tree Analysis

The Fault Tree Analysis (FTA) is a deductive systematic process that can be separated into four steps:[48]

1. Definition of an undesired event, a failure or a hazard
2. Resolving the event downwards into its immediate causes
3. Continuing resolution of events until the base is identified, the consequences of the root event node
4. Construction of a fault tree showing the logical event relationships (see Figure 4.1)

Following these steps, the analyst traverses from an event and through causes related to this event until the base cause (or causes) for this event, the faults of the system, is discovered. Since its maiden voyage in the modeling of Minuteman Missile System in the early 60's, FTA has been used in several industries; ATM, aerospace and nuclear to name a few.[3]

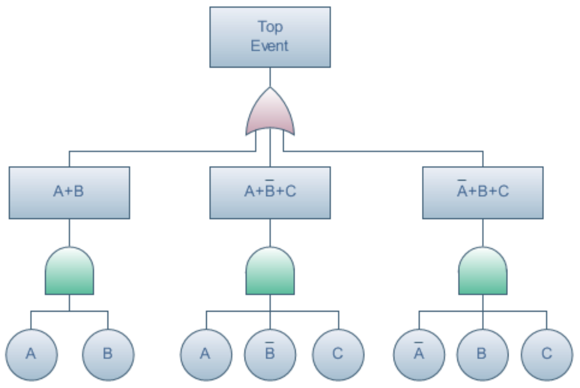


Figure 4.1: A fault tree showing a single top event, three intermediate causes and eight base causes[77]

Unfortunately, the fault tree often became impossible to manage when it was created for complex large scale nuclear power plant systems. The solution, developed by the nuclear industry, was the inductive Event Tree Analysis (ETA).[3] As apposed to the fault tree, an event tree starts with bottom node containing a failure or hazard (the leaf node of the FT), expanding in an upwards manner in the tree, identifying consequences at each level and ending up with a causing event.[2] The industry also combined the FTA and the ETA into a single structure, often referred to as the Bow Tie technique. Here the failure or hazard represents the knot that ties the two trees together into a single cause-consequence diagram.[2]

Unlike the first edition of FSD (see section 7.2), the fault tree facilitates the modeling of multiple failures, representing them as nodes in the tree structure. However, the undesired event that act as the initiator in the construction of the fault tree has to be foreseen and all intermediaries anticipated by the analyst.[49]

Limits with ETA includes the addressing of only one undesired event at a time, whereas diagrams in CHASSIS can present several in a single diagram. Moreover, distinguishing partial failures are not possible in the event tree, but in the second version of the failure sequence diagram this can be visualized by extending the means of a part component failure (see section 7.3).

4.1.4 about functional hazard assessment

The functional hazard assessment (FHA) is a technique primarily operating at system level. Over the past decades, FHA has facilitated system-analysis in the aviation industry.[3, 55] Here, the technique is applied to high-level- and/or sub-functions. Focusing on high-level-functions FHA allows for identification and analysis of hazards in the system, whilst the focus looking at sub-functions centers around identifying and analysing failures.[3] Furthermore, FHA is recognized as the first technique included in the Safety Assessment Methods (SAM). As the first of three steps, FHA is combined with other safety techniques such as HAZOP, FMEA, FTA and ETA (see sections 4.1.1, 4.1.2 and 4.1.3, respectively). Despite being able to identify and analyse failures in a function, and the option to include FHA in SAM, the security aspect included in CHASSIS is not covered in FHA, posing as a significant disadvantage.

4.2 Security

As mentioned in the previous section, most safety techniques currently used today are relying on established and traditional techniques with minor modifications. In the security field however, many of the techniques are based on modern modeling languages.[3] Examples of such modeling techniques are Secure Tropos, the KAOS Security Extension and CORAS, all of which are presented in this section.

4.2.1 Secure Tropos

Secure Tropos (ST), an extension of the Tropos methodology, aims at capturing security concepts such as security and functional requirements from in early parts of a system development process.[57] By the means of four different modeling techniques, relevant requirements are elicited from the system in the early and late requirement as well as architectural and detailed design phases.[56] By extending Tropos, ST includes the means of graphical notation[4], allowing the analysts to visualize security constraints, dependencies and entities.[3] Being able to graphically visualize threats like ST offers could help increase an analysts understanding of the system at hand. Compared to CHASSIS however, ST only considers security and not safety aspects, thus making it deficient.[4]

4.2.2 KAOS Security Extension

As opposed to Secure Tropos, KAOS to a greater extent focuses on requirement engineering, taking a more goal-based approach.[3] Here, the KAOS Security extension (KAOS SE), expands KAOS by including semi-formal graphical security notation. The notion presents, among others, malicious obstacles to security goals and vulnerabilities and countermeasures.[58] Unfortunately KAOS, and KAOS SE by extension, does not reference a unifying method resulting in a less structured requirement-elicitation method then the one CHASSIS represents.[4]

4.2.3 CORAS

According to [59], CORAS consists of three artifacts: a language, a tool and a method. The language of CORAS is a diagrammatic language consisting of simple graphical symbols and relations based on the Unified Modeling Language (UML). A CORAS diagram, such as the one in Figure 4.2 are created using the tool. The method gives a detailed description of how a assert-driven risk analysis can be conducted.[59] In early work with CORAS, UML was combined with HAZOP (see section 4.1.1) and FMEA (see section 4.1.2). Thus, when comparing CORAS to CHASSIS, it is the modeling technique with most similarities. Despite the work on combining UML with HAZOP and FMEA, CORAS remains focused at modeling security and does not aim at combining safety and security aspects like CHASSIS does.

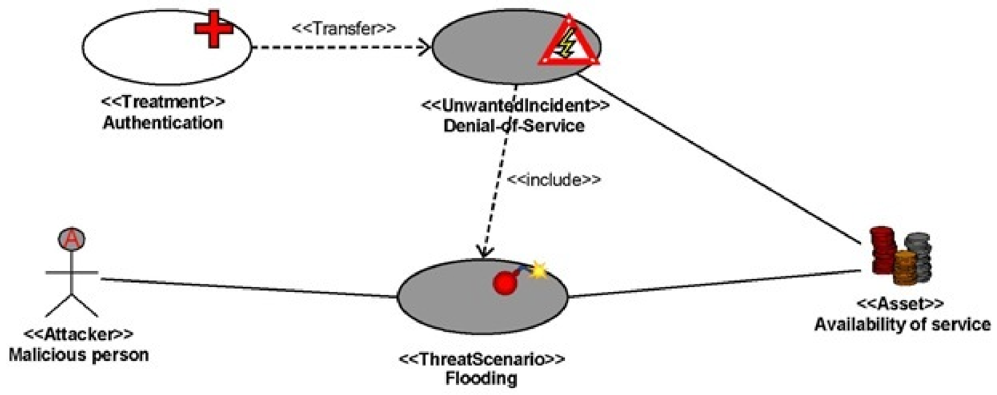


Figure 4.2: Threat diagram using the CORAS UML profile[78]

4.3 Safety and security

Only modeling the part of the system that succeed will never lead to satisfactory solutions. [2] references the importance of modeling failures related to safety and security aspects as it can aid in acquiring more nuanced knowledge, and then using this knowledge to identify solutions to failures within a system. A common weakness for the techniques presented in the two previous sections are the focus on safety aspects and security aspects only, respectively. The remainder of this section sheds light on to two proposals that aim at reducing the limitations of the techniques in sections 4.1 and 4.2: Cross-fertilization and combining safety and security.

4.3.1 Cross-fertilization

In terms of safety and security techniques, cross-fertilization entails adapting a safety technique to the security field and vice versa. The activity aims at promoting a better understanding of a system as it might lead to identification of risks that otherwise would have been overlooked.[60] Two techniques subject to cross-fertilization is HAZOP (see section 4.1.1)and FMEA (see section 4.1.2).

An attempt at adapting HAZOP to the security field involves establishing specialised guidewords and attributes for security. [61] presents another approach to HAZOP cross-fertilization where the original guidewords in HAZOP where combined with elements of the Misuse Case (MUC); guidewords where here systematically applied to flow of events in textual MUCs.[3]

By the means of small modifications, FMEA has been used in the analysis of threats and intrusions.[3] FMEA cooperates with the Threat Effects Analysis (TEA) technique facilitating the identification, classification and analysis of threats and mitigation-suggestions aiming at reducing risk. Another attempt at FMEA cross-fertilization combines the technique with the Intrusion Mode and Effect Analysis (IMEA) in order to perform dependability analysis.[3]

Despite the possibilities presented in the cross-fertilization of HAZOP and FMEA, the other limitations these techniques holds (see sections 4.1.1 and 4.1.2) gives CHASSIS an advantage.

4.3.2 Combining safety and security

As the title suggest, the techniques in this section aims at unifying safety and security aspects. One example of such a technique is previously presented in section 4.2.2; KAOS. Even though it does not present an explicit safety-security-combination and does not aim at combining these, a union may be achieved by the means of KAOS goals.[4] One of these goals, namely the obstacle feature, provides the means to represent different goals; for instance safety and security goals. Hazard and threats pose as two goal-obstacles and can be visualized in a KAOS obstacle model.[62]

Boolean-logic Driven Markov Process (BDMP) has been adapted from the safety to the security field[63], but now it poses as a technique capable of combining both aspects.[3] The technique, resembling the graphical notation of FTA (see section 4.1.3) has extended features allowing more extensive modeling.[5] In [43] however, CHASSIS proved as a more suited model for the visualization capabilities caused by its use of UML.

4.4 Modeling tools

This section covers a selection of modeling tools. In addition, these are evaluated against CHASSIS and FSD.

4.4.1 SDL Threat Modeling Tool

This Microsoft tool is meant to be a part of the design phase of the Security Development Lifecycle (SDL). This way, the tool facilitates so software architects can identify and mitigate potential issues related to security in an early stage of system development. The idea is that nipping the issues in the bud will reduce the development-costs, i.e the time and money it takes to resolve it, as well as how difficult it will be to resolve.[14]

As opposed to several other modeling approaches that centers on assets or attackers or a complex blend of the two, SDL is centered on the software. This is one of the key areas that separate SDL from other tools. The other key point is it's analysis focus: The tool is a focused design analysis technique instead of a requirements analysis technique.[14] Since the CHASSIS process aims at unifying both safety and security aspects during requirement elicitation, the SDL modeling tool does not comprise the correct functionality as this tool focuses on design analysis, and does not aim at covering both safety and security aspects.

4.4.2 SdEdit

Quick Sequence Diagram Editor (SdEdit) is a tool that allows the user to create sequence diagrams (SDs) based on a formal metamodel; UML. Following a simple syntax, the diagrams are created by rendering a pure textual description, meaning that there is no need to drag and drop figures onto a canvas.[65] Looking at FSD, the SdEdit tool will cover only the first step of FSD creation (see section 3.3): creating the UML SD, as SdEdit provides no means to add new figures to the diagram generator. The tool, in other words, have no way of expressing either safety nor security aspects. It can also be discussed if a pure textual editor would aid or hinder a team brainstorming activity that the CHASSIS method encourages. Yet another model based tool, the Model Based Safety Analysis (MBSA)[66] is, unlike SdEdit, a graphical tool editor. MBSA unfortunately suffers the same extensibility limitations as SdEdit, making it an ill fit for generating Failure Sequence Diagrams.

4.4.3 SeaMonster

SeaMonster is a graphical security modeling tool.[8] The project was initiated in 2007 by SINTEF, the largest independent research organisation in Scandinavia.[18, 19] From 2007-2008 the tool was developed together with students at NTNU. In 2008, SeaMonster was included in the SHIELDES project where it was further developed for two additional years.[18]

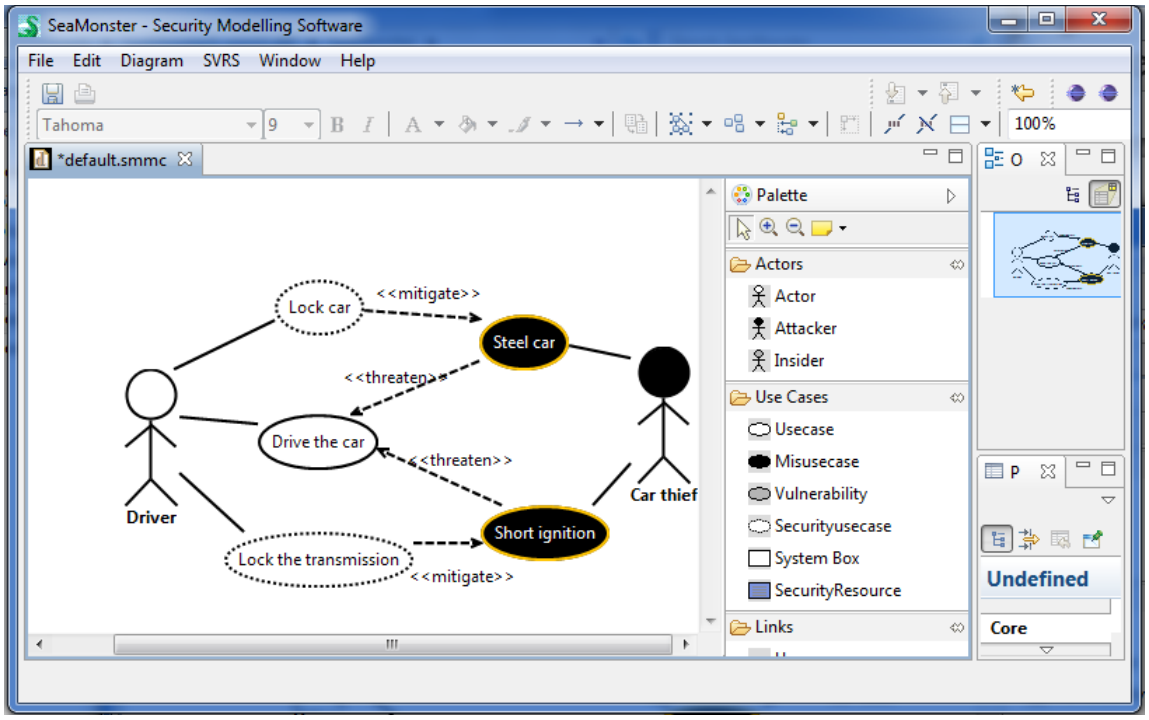


Figure 4.3: The SeaMonster tool with an example MUC drawn

The purpose of SeaMonster was to create a common platform for security modeling. The tool was meant to facilitate the reuse of models, and thus reducing the time it takes developers and security experts to model security.[18] An advantage with this modeling tool is the notation and modeling techniques it supports, namely the Attack Tree and misuse case diagram (MUC). These are familiar models for both security experts and analyzers and strengthens this tool in regards of learning curve and usability.[18] On the other hand, SeaMonster does not have any support for the generation of UML SD, and like SdEdit (see section 4.4.2) no options for adding new diagram types or figures to an existing diagram type to meet the additional notation requirement of FSD.

Chapter 5

Artefact design - CHASSIS

This chapter serves as the last information source, with Chapters 3 and **??**, together defining the constructs of the CHASSIS tool. The following section present input from safety domain experts, a presentation of the problem, proposal and suggested solution, details regarding collecting domain data and techniques aiming at Domain Specific Language (DSL) creation.

5.1 Input from industry

Meeting the aviation industry, here represented through safety experts at Avinor, aimed at shedding light on their process when performing hazard analysis as the means to extract system requirements. Moreover, the meeting focused on how the safety experts envisioned a computerized tool for hazard analysis, a tool for CHASSIS, and how such a tool could fit into the analysis task.

When performing a hazard analysis, Avinor tend to utilize brainstorming. Here, a facilitator, a secretary and several domain experts arrange a meeting where the system at hand is discussed from a safety point of view. The facilitators' responsibility here includes ensuring and maintaining structure during the brainstorming in order to try and cover all relevant input from the domain experts. It is also possible for the facilitator to join in the safety experts discussions during the meeting, whereas the secretary mainly makes notes of the discussion. The notes can be pure text, but is often accompanied with low-level sketches of the system under analysis. After the meeting ends, the safety experts collects notes from the secretary and begins the work with tidying them and structure the conducted analysis and resulting requirements.

Presenting the FSD of CHASSIS, as described in section 3.3, a discussion regarding the extensive functionality provided by the FSD creation process quickly emerged. It was suggested that the constrictions provided by the UML SD followed by additional constrictions when adding failures and mitigating factors would be to heavy. Worst case, this strictness actually would prevent the loose flow of suggestions and ideas during the brainstorming as the sketch-possibilities of the tool would force the direction of the brainstorming. Here, it was stated that the more limited the notation, the better. One participant in the meeting with Avinor even suggested that a circle, a rectangle and a line would be sufficient as notation go. These two statements lead to a proposal of a twofold tool. The first part is utilized in the brainstorming process. This tool has limited or no restrictions connected to the notation, giving the user or users free reigns to sketch as they please. Then afterwards the second tool can be used by the person cleaning up and structuring the brainstorming data. In this version, the tool would be more syntactically strict, thus including UML SD and FSD functionality constricting the notation.

5.2 Problem description

Chapter 1 briefly presented the problem covered in this thesis: a CHASSIS tool for creating FSDs aiding in the requirements elicitation process. For one, the case study looking into how FSD could support FMEA in *"modelling failures and their effects through interactions between system components"* documented in [6] suggests that a tool would give the FSD more structure. Furthermore, this structure would facilitate *"collecting all the relevant information directly in the FSD."*[6] In another paper adapting MUSDs to support failure analysis, an adaption resulting in the FSD technique, a need for a tool is also mentioned. The main problem while conducting the experiment was challenges related to complex drawings taking up too much space on the whiteboard, one of many issues that may be resolved with a computer tool.

As CHASSIS aims at aiding analysis and requirement elicitation in the aviation industry[3], which often are faced with both large and complex systems, the advantages of a tool will increase proportionally with the size of the system: diagrams may be stored, divided into several sub-diagrams (see section 7.3) and distributed between stakeholders easily. Another advantage of using computer software versus an analog approach are programs' ability to make tasks easier.[10] For instance, graphical software allows the user to easily edit and remove notation in the figure without having to redo the entire diagram as you might have to if the figure where drawn on a whiteboard or a piece of paper. The software will also encourage users to apply a common set of notation, thus creating a shared platform for understanding the system and its content. In addition, the software will ensure that notation outside the scope of the diagram in question is unavailable for the user. During a brainstorming, the tool will therefore serve as a base language when discussing the system, a language that may also be used when the notes from the brainstorming is to be tidied up.

5.3 Proposal and suggested solution

The problem, input from Avinor, a thorough investigation of CHASSIS (see Chapter 3),looking at related techniques and tools and comparing them to the CHASSIS method (see Chapter 4) all point at the same conclusion: there is a need for a CHASSIS tool focusing on FSD generation. The input from Avinor (see section 5.1) presented an interesting issue regarding the tool and what was to be covered in this first CHASSIS tool creation effort: creating a twofold tool or focusing on one half only? Deciding on the second option was a result of a set of high level requirements constructed by CHASSIS' domain expert, Raspotnig (see Table 6.1). This version of the CHASSIS tool will, based on these requirements, cover the modeling of FSDs, including all accompanying functionality and restrictions.

Failure Sequence Diagram, as presented in section 3.3, reviles that the diagram to a great extent utilizes a SD. This positions the the tool creation against the metamodel of the SD; Unified Modeling Language (UML). In addition to SD notation, the tool needs the functionality to draw extra FSD specific notation. (see Table 6.1) These two points together suggests that Model-Driven Software Development (MDSD); a method allowing for the creation of a custom metamodel, a FSD model with UML as a base language, would be a perfect fit for the realization of the CHASSIS tool approximation. In MDSD, the activity that is metamodeling covers several relevant challenges:

1. Constructing a Domain Specific Language (DSL) by describing the abstract syntax,
2. Validating the model against constraints defined in the metamodel,
3. Tool generation and

The remainder of this chapter and the two next chapters presents the DSL for FSD, how it is developed and how this is used in the creation of the CHASSIS approximation.

5.4 Collecting domain data

In order to create a DSL that fulfils it's purpose, which in this situation is defining the constructs of FSD, understanding the domain is crucial. If the DSL is incorrect or incomplete, the implementation of that DSL might be erroneous to the degree that it is unable to fulfill its purpose. To ensure that the FSD domain is covered properly, the data collected in order to create a DSL for FSD originates from several sources: the background study of the CHASSIS method (see Chapter 3), domain experts on CHASSIS and the OMG UML superstructure.[30] The CHASSIS background study provided an overview of the CHASSIS method as a whole. Furthermore, the papers covering the experiments conducted with FSD[6, 4, 2, 5] shed light over existing notation as well as possible improvements.

The second information source on the road to a DSL was a CHASSIS domain experts. Here, Raspotnig provided valuable information regarding the syntax and the intended use of the notation in FSD. He gave insight to, and clarification of, each of the FSD notation components by sharing information retaining details about the graphical appearance and the functionality of the notation. By clarifying the intended use for each element in the notation list, Raspotnig also provided the information needed in deciding which UML metaclasses corresponded to the FSD notation. The third source, the OMG UML superstructure, was vital in understanding how notation of the FSD elements would turn out, giving a detailed list of all constraints and associations connected to each metaclass of UML (see section **??**). This is quite useful if an extension of a metaclass includes new constraints, as the listings in the structure can aid the developer so that he or she avoids adding constraints that will be in conflict with existing ones. As the structure of the UML SD, including constraints and associations, is already incorporated in the Papyrus tool (see section 6.3) and is quite complex, I see no need for the complete listing of the constraints and associations of each of the metaclasses that are to be extended. For those interested, the complete superstructure is located in [30].

5.5 Creating a Domain Specific Language

A DSL is a language designed to be useful for a specific set of tasks.[29] In this project, the DSL is contains the information that constitutes the FSD (see sections 3.3, 7.2 and 7.3). DSLs are created to solve specific problems in a particular domain, and only cover this domain.[29] The first decision made when creating a DSL whether or not to build the it on top of existing UML concepts, and thereby limit the DSL to extending or restricting UML meta-types and concepts, or not.

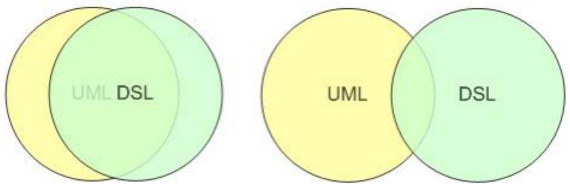


Figure 5.1: UML extension vs. MOF[29]

The answer to this decision can be found in the domain space of the DSL you are going to create. Figure 5.1 displays two different DSL domain spaces, the figure on the right representing Meta-object family (MOF) approach. Extensions using MOF is realised by using the language found in M3 (see Figure 5.2). The result is a meta-model (top level M2 of Figure 5.2) which is applied to a meta-domain model which again is applied to an application model (the model actually visual to the end-user). An advantage with extension using MOF is the modification possibilities. MOF allows both the modification of an existing metamodel as well as the creation of a new metamodel, meaning MOF is not limited to extending UML, but also other modeling languages.

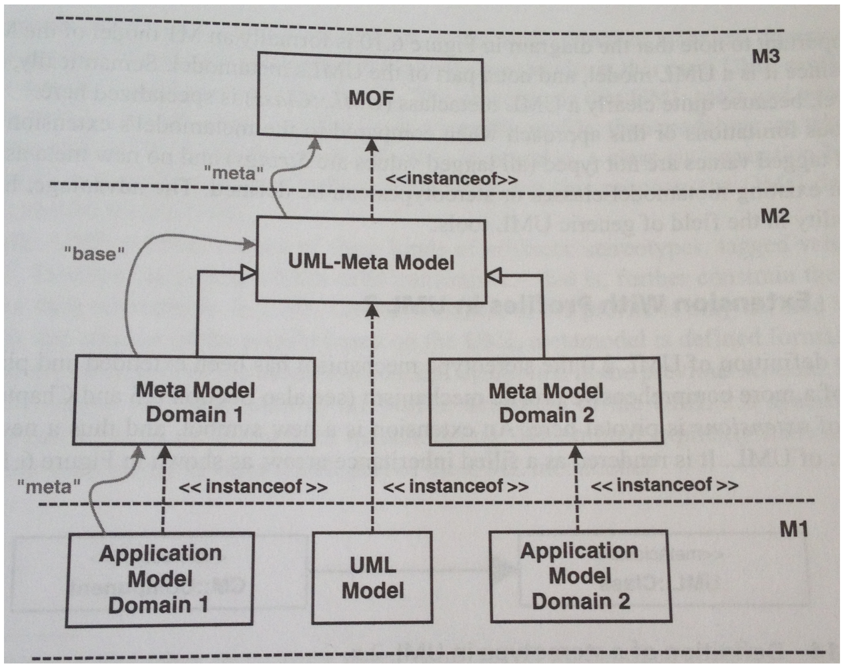


Figure 5.2: The inheritance levels when creating a DSL via MOF[7]

In this setting however, with FSD as the basis for the DSL, and thus the resulting DSL in large part overlapping with UML as previously mentioned (see section 3.3) and like the left image of Figure 5.1 shows, the UML extension option is chosen. The next step in the process will be selecting an appropriate technique for extending UML.

5.6 UML extension techniques

In short, there are four different technique categories for UML extension; featherweight, lightweight, middleweight and heavyweight.[29] Which one you choose is based on your DSL and the amount of extension that will be required in order to adapt UML into what the requirements of the tool, the CHASSIS tool, states.

5.6.1 Featherweight extension

The featherweight extension entails the adding of keywords. A keyword is a reserved term, normally appearing as a text annotation attached to a UML element.[29] The purpose of the keyword is adding the functionality of distinguishing between different elements in a diagram. An example of keyword use is seen in Figure 5.3. Here, the keyword <<interface>> is added to the Interface meta-type (the left figure) in order to distinguish between the UML::Class classifier from the UML::Interface.

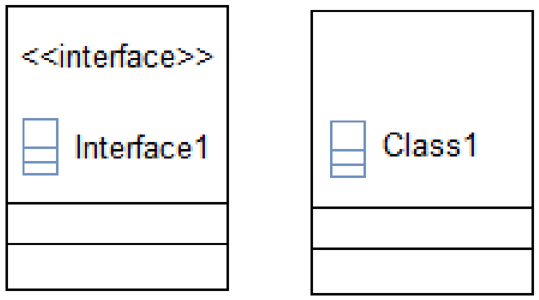


Figure 5.3: Using a keyword to distinguish between a class and an interface

Another use of the keyword functionality is separating different types of relationships.[29] Figure 5.4 shows how adding the keyword <<extend>> helps separating a dependency- from an extension-relationship. A third use is specification of a meta-attribute value (a value attached to a UML concept). An example would be adding the keyword <<singleExecution>> to an Activity, indicating that the isSingleExecution() attribute of Activity is true. A last example of use is stereotype-indication. Adding a keyword, for example <<modelLibrary>>, to a package would indicate that the package contains a set of elements meant to be shared by multiple models.

Advantages of the featherweight extension include the simplicity of adding the keywords and that the functionality of a keyword offers a great way to separate similar looking elements. On the other hand, the extension is limited to keyword-use only and does not provide the means to add new figures to a diagram editor, as FSD will require. The limitation of the featherweight extension leads us to the second alternative; lightweight.

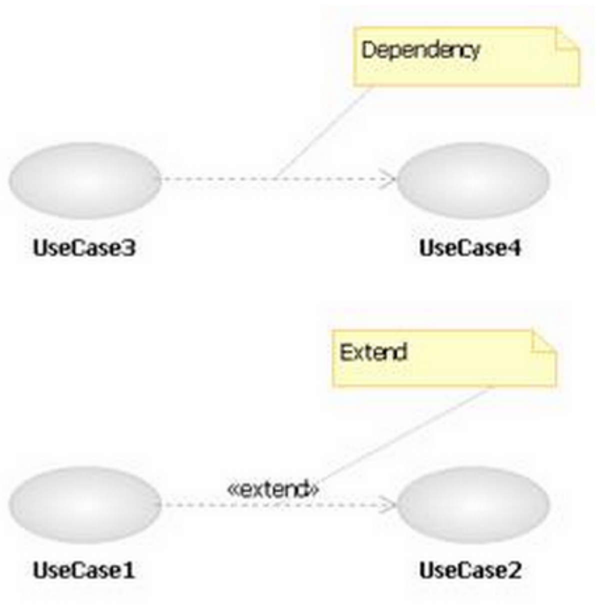


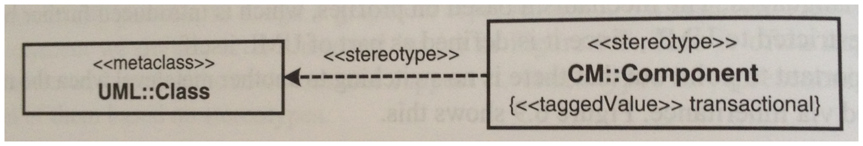
Figure 5.4: Using a keyword to distinguish between different relations in a diagram[29]

5.6.2 Lightweight extension

According to eclipse.org, the developer should, as mush as possible, favor using the lightweight extension. This is conveniently the chosen extension type in this project, and therefore described in greater detail throughout the next three subsections.

Lightweight extension in UML 1.x

This type of lightweight UML extension applies a UML-specific functionality, namely the use of stereotypes. A stereotype is defined as a part of the profile mechanism. Because of this, the UML itself can act as a way of extending the UML metamodel without being required to use the means of a MOF-provided modeling language.[7] The stereotype, in this setting, acts like an instance of a metaclass and defines how that metaclass may be extended.[15] Extending a model using stereotypes is UML specific, meaning that other MOF-based modeling languages have to define extension mechanisms of their own.[7]



[7]

Figure 5.5: Adapting UML by the means of a stereotype

Figure 5.5 is a typical example of UML extension using a stereotype. The UML metaclass is extended by the CM:Component stereotype, who have a tagged value (see section 5.6.2), *transactional.* Formally, this extension is a M1 model of the MOF hierarchy in Figure 5.2 since it isn't a part of the UML's metamodel, but rather a UML model itself. From a semantic point of view however, the extension is on the M2 MOF-level because of the presence of a UML metaclass (the UML::Class).[7] The main advantage of adapting a UML 1.x model using the stereotype-functionality is the usability found in the fields of UML tools.[9, 26, 27, 28] Unfortunately, this solution has serious limitations compared to metamodel extension with MOF (see section 5.5). All tagged values, such as the transactional tag in Figure 5.5 are not typed, meaning that all tags will be detonated as text (often refereed to as Strings in the field of software programming). In addition it is not possible to define any new meta-associations between stereotypes or existing metamodel classes. These issues have fortunately been resolved in UML2 and is presented in the next section.

Lightweight extension in UML 2

In the UML2 definition, the stereotype mechanism has been extended and placed in the context of a more encompassing profile mechanism. Here, extensions are a crucial concept. Looking at Figure 5.6 you will see the extension as a new symbol, detonated by a solid inheritance arrow from the stereotype CM::Component to the UML::Class metaclass. This extension is an entirely new construct of the UML language, formally defined in the UML metamodel, and not a version of existing concepts such as inheritance, association, implementation or stereotypical dependency.[7]

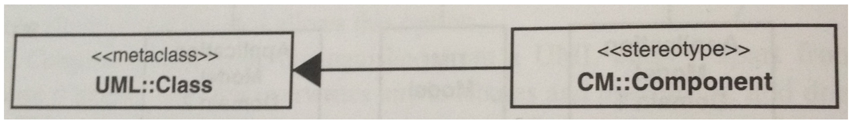


Figure 5.6: Adapting UML by the means of a profile[7]

A stereotype in UML2 can have attributes.[7] As in UML1.x, the stereotypes are rendered as tagged values in the model the stereotype is used. New in UML2 is the assigning of a type to an attribute. The result is the possibility of assigning attributes to a stereotype with type other than String. The extension of the stereotype concept to a new construct and the new functionality of typesetting the attributes is the base for why this is the best solution for the development of a CHASSIS plugin, and therefore also the selected alternative. The next subsection will present the UML profile diagram to further detail the profile construct.

UML Profile diagrams

A profile diagram is a structure diagram [15], showing the static structure of a system and its parts on different abstraction and implementation levels, including how they are related.[16] The diagram describes lightweight extension mechanisms to the UML by allowing the adaption of a metamodel (here: the UML metamodel) with constructs that are specific to a particular domain.[15] The profile is created through the inclusion of metaclasses, definition of custom stereotypes by extensions.

Metaclass - a profile class from an existing metamodel which can be extended through one or more stereotypes.[15]

Stereotype - an instance of a metaclass. This construct defines how a metaclass may be extended as a part of a profile. A Stereotype can not be used by itself, and must be used alongside one of the metaclasses it extends. In addition, it can't be extended by another stereotype.

Extensions - an associative relationship. Its purpose is indicating that a metaclass' properties are extended in a stereotype. Detonated graphically as an arrow (see Figure 5.6), where the tip ties the extension to a metaclass and the end to the stereotype extending it.

Tagged values - properties/attributes of a stereotype (see section 5.6.2).

Constraints - presents some restriction related to a construct. The construct can be a metaclass, where constraints are build into the class, or a stereotype where the constrains are new additions included with the constraints the stereotype already has inherited from the metaclass it extends.[67]

As apposed to MOF, profiling in this manner allows neither the modification of existing metamodels or the creation of a new metamodel. What is allowed however is adapting or customizing an existing metamodel, the UML metamodel in this particular situation, with constructs specific to a particular domain, platform or domain. Furthermore, it is impossible to remove any constraints applying to a particular metamodel, but it is possible to add new constraints witch will be specific to the profile.[15]

The UML profile diagram provides the exact construct needed to aid in the development of a CHASSIS tool for generating FSD's: a base metamodel, UML, and the means to extend this model by adding new notation to the FSD DSL, metaclass extension by stereotypes. Naturally, this technique is chosen. The middle- and heavyweight extensions are still presented in the two next subsections, further justifying the selection of the UML profile diagram option.

5.6.3 Middleweight extension

Extending UML through middleweight extension entails the specialisation of UML-meta-types. The extension begins with referencing the UML.metamodel.uml a a whole, here containing a merged set of meta-types. The next, and final, step is to add your own specific types to selected meta-type-set. This allows the developer to add and modify the behaviour, structure as well as the constrains of an element. Despite the quantities of development possibilities and the fact that creating a middleweight extension is quite easy[29], this type of UML adaption is often discouraged. The main issue relates to the fact that the extension creates a dependency on a specific version of UML. This could mean that if any extended meta-type form the UML metamodel is edited in any way, the model element representing the actual extension might also need editing. The developer is also forced to extend all of UML, even if he or she only is interested in a subset.[29]The final section briefly present the heavyweight extension.

5.6.4 Heavyweight extension

What separates the middleweight from the heavyweight is what gets extended. When middleweight extension involve reuse by specializing types from the referenced UML.metamodel.uml, the heavyweight extension focuses on copying and merging meta-types.[29] First, the developer selects the units of the language to be extended, and merges them. This facilitates a selective extraction of the UML concepts required for the extension, and only them. The second step is for the developer to add his or hers own types that are to be specific for the domain. However great the abilities to customize and specify behaviour is, this alternative cost to much to develop, both in time and level of difficulty.

Now that the construct that will aid in the extension of UML have been defined, a lightweight extension in UML2 by the means of a profile, the tool approximation implementation can begin.

Chapter 6

Artefact construction - Technology

This chapter introduces a set of high level requirements for the CHASSIS tool and presents a platform containing the functionalities meeting these.

6.1 Technology requirements

The table bellow are high level requirements extracted from a written high level description of the CHASSIS tool, provided by Raspotning. The full version of the description is found in Appendix A.

Id	Requirement
HL1	The tool shall implement FSD of CHASSIS
HL2	The tool should integrate with other development, safety or security analysis tools using XML
HL3	It should be possible to extend the tool to connect to other diagrams, especially MUC diagrams
HL4	The tool should support functionality to save and save as...
HL5	Print jpeg and xml-based reports
HL6	The user should be constrained from specifying elements that are syntactically wrong, e.g, vulnerabilities in FSD
HL7	The system shall be user friendly
HL8	The tool shall implement all the rules and notation from UML sequence diagram (SD)
HL9	The tool shall implement all the notation of FSD
HL10	The tool user interface (UI) should have a palette displaying the notation of 1. SD and 2. FSD

Table 6.1: Initial high level requirements (see Appendix A)

6.2 Platform - Eclipse

In the world of computer programming, Eclipse is an integrated development environment (IDE).[20] An IDE is a software program providing holistic facility to computer programmers for software development.[20] It includes both a compiler that transforms source code to an executable program[21] and an interpreter that directly performs instructions written in a programming language.[20][22]

The Eclipse IDE contains a base workspace where the programmer gathers source code files and other recourses, combining them to a uniform unit.[23] In addition, in order to customize and add specific feature to the unit, Eclipse can access an extensive collection of plugins.[20] An example of a plugin is Papyrus, a plugin for UML model extension, which is covered in section 6.3.

6.3 Eclipse plugin - Papyrus

The Eclipse Modeling project comprises all official projects in Eclipse that focuses on model based development technologies.[24] In the project category Model Development Tools (MDT) we find the Papyrus project. The goal of the project is creating a user-consumable environment that can edit any kind of Eclipse Modeling Framework (EMF) model, focusing on UML and related modeling languages. [17] The result; the Papyrus tool.

Papyrus presents as the glue between UML editors and other model-driven engineering tools.[17] It can be used both as a stand-alone tool or as a plugin to Eclipse.[25] Stand-alone, Papyrus presents a comprehensive tool facilitating the generation of several different formal UML diagrams: class-, sequence-, usecase- and activity diagram to mention a few. Installing Papyrus as a plugin on the Eclipse platform facilitates the diagram editor from the stand-alone-version, as well as an extensive environment for UML Profile diagram generation.[17][13] As described in **??**, an UML Profile is a construct allowing extension of the UML metamodel with stereotypes (HL1, HL9). The result is a DSL customized for a particular domain, constrained by the UML metamodel.(HL6) The editor support in Papyrus is also quite advanced and includes:[13, 9, 25]

- Editor functionality to save diagram-files as .di (digital illusion), .xml (extensible markup language) and image format of your choosing (HL2, HL4, HL5),
- An editor palette displaying the complete notation of UML SD (HL8),
- Functionality to customize the editor, adding new notation to the palette (HL1),
- The creation of several custom editors in the same tool, allowing the creation of different diagrams and facilitating linking between them (HL3)
- Displaying both UML notation as well as new custom notation in the same palette (HL10)

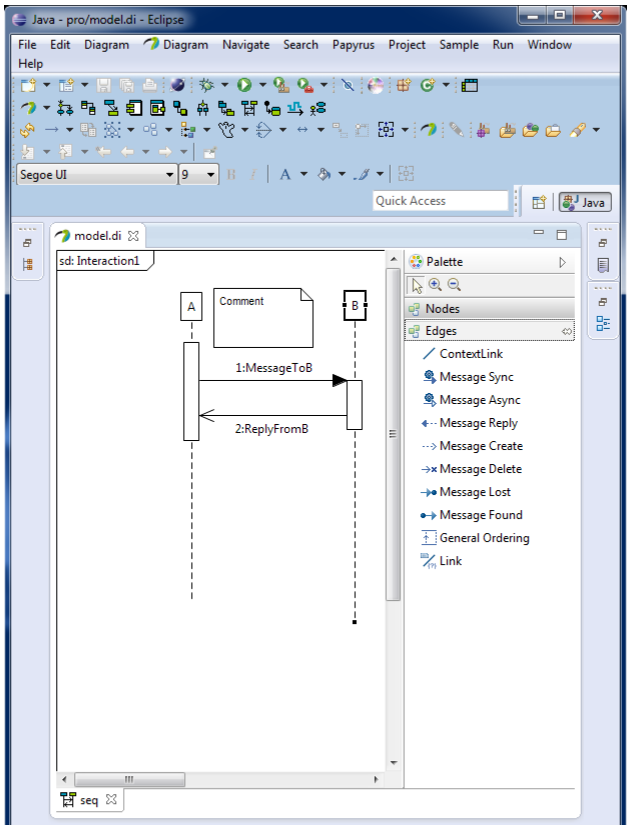


Figure 6.1: A sequence diagram drawn using the Papyrus diagram editor

Even though Papyrus is the chosen development tool for this project, it was not the only tool investigated and evaluated. Table 6.3 therefore compares Papyrus with two other means to tool generation: Microsoft Visual Studio Ultimate (VSU) and the Graphical Modeling Framework (GMF).

Property	VSU	GMF	Papyrus
Development platform	Microsoft VS	Eclipse	Eclipse
Platform licence cost	110,127.56 NOK[68]	Free	Free
Underlying development language	C#	Java	Java
Tool makes the UML metamodel available	Yes	No	Yes
Complete SD notation in the palette (HL8)	No	-	Yes
Adding new notation (HL9)	Easy[3]	Easy	Easy[3]
Presentation of DSL model in the tool	pure xml file	graphical	graphical
Reediting an applied DSL	Easy	Demanding[4]	Easy[5]
Adding shape to new notation	Attaching image-file to the stereotype	Custom made (see 7.6)	Attaching image-file to the stereotype

Table 6.3: Comparing Papyrus, GMF and VSU

Table 6.3 aids in the justification of choosing Papyrus as the preferred development tool for this project. Having made this decision, the next step is constructing the artefact. The next chapter present the functional requirements of the CHASSIS tool approximation for FSD generation, and how Papyrus aids in the construction.

[3]With both Papyrus and VSU adding new notation that is not a part of the UML diagram is not possible.[31] This demands that the DSL of the new model i quite similar to the original diagram model

[4]Editing the tool in any way after the diagram code has been generated is a gruesome process. Editing parts of the domain model will require the regeneration of all the other connected models, resetting up all relations between them (see section 7.6)

[5]Editing the profile diagram after a profile has been applied is a simple process. The developer simply updates the stereotype connections to the specific notation and applies the edited/added stereotypes to the appropriate notation in the diagram, leaving the unchanged extensions untouched

Chapter 7

Artifact construction - CHASSIS tool

This chapter gives a detailed overview of the functional requirements of the CHASSIS tool approximation, how these are fulfilled by extending metaclasses. Further on, the chapter presents how the profile is applied in Papyrus. Finally, the approach selected when realising the degree incompleteness in the Papyrus tool is presented along with it's set of functional requirements and implementation means.

7.1 Functional requirements - first version

Tables 7.1-7.5 display the functional requirements related to the notation of FSD. Because of HL8 in table 6.1 and thus to avoid redundant notation in the tool palette (see section 7.4), the following notation presented in Figure 7.1 are not created as stereotypes:

1. Event message which is an instance of the Message Async of UML SD
2. Note which is an instance of a Comment of UML SD
3. Component which is an instance of a Lifeline of UML SD

Each of the stereotypes extending UML metaclasses in this project will include attributes for a shape and an icon. The shape represent the figure used in the actual diagram and the icon is used as a visual aid in the diagram palette.

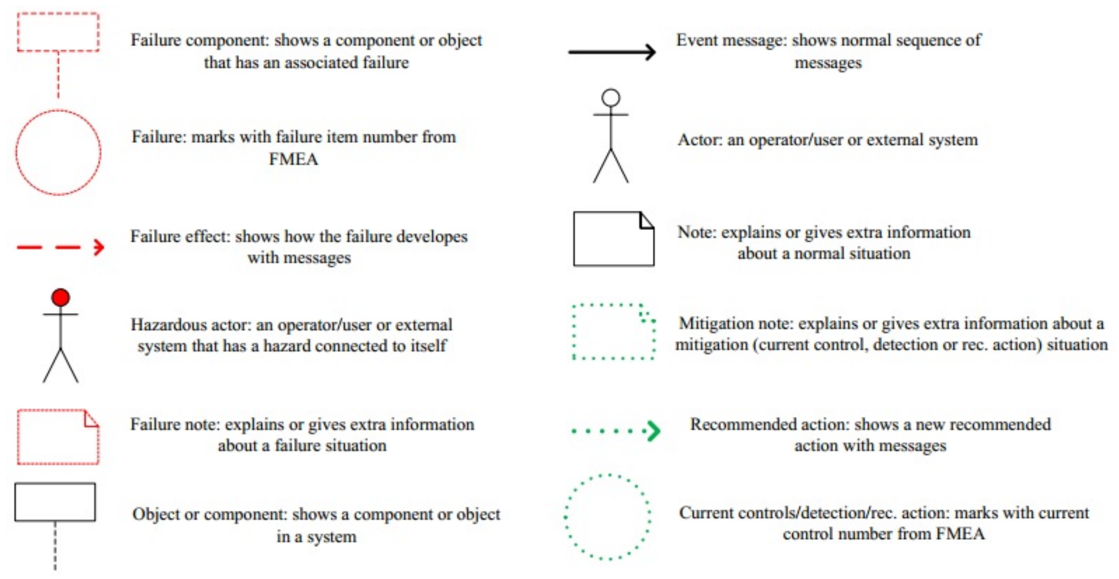

Failure component: shows a component or object that has an associated failure
Failure: marks with failure item number from FMEA
Failure effect: shows how the failure developes with messages
Hazardous actor: an operator/user or external system that has a hazard connected to itself
Failure note: explains or gives extra information about a failure situation
Object or component: shows a component or object in a system
Event message: shows normal sequence of messages
Actor: an operator/user or external system
Note: explains or gives extra information about a normal situation
Mitigation note: explains or gives extra information about a mitigation (current control, detection or rec. action) situation
Recommended action: shows a new recommended action with messages
Current controls/detection/rec. action: marks with current control number from FMEA

Figure 7.1: The first edition notation of FSD[6]

7.1.1 Actor and hazardous actor

Id	Requirement
FAH1	Hazardous actor shall be represented by a stick-figure with a filled red head (see Figure 7.1)
FAH2	Actor shall be represented by a stick-figure
FAH3	It should be possible to connect an actor to a component through a link
FAH4	It should be possible to connect an actor to a complete component failure through a link
FAH5	It should be possible to connect a hazardous actor to a component through a link
FAH6	It should be possible to connect a hazardous actor to a complete component failure through a link
FAH7	Actor should have a name
FAH8	Hazardous actor should have a name
FAH9	The name of an actor should be placed beneath the actor shape
FAH10	The name of a hazardous actor should be placed beneath the hazardous actor shape

Table 7.1: Requirements of actor and hazardous actor notation

As mentioned in 5.6.2, a stereotype extending a metaclass will inherit all the functionality it contains. The UML Annotated Link is one of Comment metaclass' associations. FAH3-FAH6 are therefore met by the means of this link facilitating a connection between a UML Comment and other components in the diagram. The extension of the Annotated link is located in Table 7.5. FAH10 and FAH11 are requirements the Papyrus tool covers (see section 7.5).

7.1.2 Marks and notes

Id	Requirements
FMN1	Failure mark shall be represented by a red dotted circle
FMN2	Current control shall be represented by a green dotted circle
FMN3	Failure mark shall have the option of a text field that may contain failure item number from FMEA
FMN4	Current control shall have the option of a text field that may contain current control number from FMEA
FMN5	Text field of current control shall be located inside the current control shape
FMN6	Text field of failure mark shall be located inside the failure mark shape

Table 7.2: Functional requirements of marks notation

As with actor and hazardous actor, current control, failure mark and the failure and mitigation note extends the UML comment metaclass. FMN1, FMN2, FMN7 and FMN8 are met by adding a shape and an icon attribute to each of the failure mark, current control, mitigation and failure note stereotypes. All four notations will require the functionality of a text field. These requirements, FMN3, FMN4, FMN9 and FMN10, are met by utilizing the text field attribute of UML Comment. FMN11 and FMN12 are, as with actor and hazardous actor, met through the Annotated link association. The placement of the text fields in the notation, FMN5, FMN6, FMN13 and FMN14, are covered in the Papyrus tool (see section 7.5).

Id	Requirements
FMN7	Mitigation note shall be represented by a green box with upper right corner folded down
FMN8	Failure note shall be represented by a red box with upper right corner folded down
FMN9	Mitigation note shall contain a text field
FMN10	Failure note shall contain a text field
FMN11	It should be possible to connect a mitigation note to a current control through a link
FMN12	It should be possible to connect a failure note to a failure mark through a link
FMN13	Text field for mitigation note shall be located inside the mitigation note shape
FMN14	Text field for failure note shall be located inside the failure note shape

Table 7.3: Functional requirements of notes

7.1.3 Components

Id	Requirement
FC1	Failure component shall be represented by a red square and a red tail (see 7.1)
FC2	Failure component shall have the option of a text field
FC3	The text field of the failure component shall be located inside the red square
FC4	It should be possible to connect a failure component with a UML Lifeline through a link
FC5	It should be possible to connect a failure component with another failure component through a link

Table 7.4: Functional requirements of failure component

The Component and Failure Component elements of FSD extends the UML::Lifeline metaclass. As previously, the graphical representation of the notation is covered by adding a shape and icon to the stereotype, fulfilling FC1. The option of a text field in the head of the Failure Component, FC2, is inherited from the UML Lifeline metaclass. Functionality connecting Failure Components together, or with a Lifeline, are also inherited feature from UML Lifeline[30], meeting FC4 and FC5. The placement of the Failure Component title is covered in the Papyrus tool (see 7.5).

7.1.4 Links

Id	Requirement
FL3	Link between actor and component shall be represented by a green dotted line
FL4	Link between actor and failure component shall be represented by a green dotted line
FL5	Link between hazardous actor and component shall be represented by a red dotted line
FL6	Link between hazardous actor and failure component shall be represented by a red dotted line
FL9	A green link should have actor as source and 1. component or 2. component failure as a target
FL10	A red link should have hazardous actor as source and 1. component or 2. component failure as a target

Table 7.5: Functional requirements of links connecting an actor/hazardous actor to a lifeline/component failure

The links included in FSD are the most complex elements in the notation list, due to their functionality to connect different notation in the editor window. Failure Effect and Recommended Action are both the result of an extension of the Message Async UML metaclass[30]. The reason for this is that other message types in SD (Message Synch/Lost/Found/Create/Delete and so on) require the presence of an Execution Specification on the Lifeline (see section **??**). The UML Message Async link has built in attributes that allows a user to place a message between a UML Lifeline and a Failure Component(FL7 and FL8). Remember that a Failure Component is an instance of the UML Lifeline, but with added functionality.

Adding arguments to a Failure Effect and a Recommended Action (FL11 and FL12) are fulfilled through one of Message Async's associations. FL1 and FL2 are met by adding shapes and icons to each of the Failure Effect and Recommended Action stereotypes. Table 7.5 displays requirements for Failure Effect and Recommended Action from FSD. In addition, a new notation not included in the notation for FSD in Figure 7.1 is included in the table. These links provides the means to link an Actor or a Hazardous Actor to a another figure in the diagram. FL8 and FL9 are therefore met by including two instances of an Annotated Link in the diagram palette: one green link connecting an Actor to a Lifeline/Component Failure (FL3 and FL4) and a red link connecting a Hazardous Actor to a Lifeline/Component Failure (FL5 and FL6). FL13-FL15 are fulfilled by utilizing an element called Enum when creating the FSD UML Profile.

Id	Requirement
FL1	Failure effect shall be represented by a red dotted arrow
FL2	Recommended action shall be represented by a greed dotted arrow
FL7	A failure effect should be able to have 1. a component or 2. a failure component as a source and 1. a component or 2. a failure component as a target
FL8	A recommended action should be able to have 1. a component or 2. a failure component as a source and 1. a component or 2. a failure component as a target
FL11	Failure effect should have an argument
FL12	Recommended action should have an argument

Table 7.6: Functional requirements of links creating a connection between a lifeline/component failure and a lifeline/component failure

7.2 First edition FSD UML profile

Based on the requirements presented in the previous section, a UML Profile defining the constructs of FSD is created. For each notation in Figure 7.1, and the additional links defined in Table 7.5, the appropriate metaclasses that are to be extended are created in the FSD UML Profile. Remembering that a metaclass may be extended by several stereotypes (see section 5.6.2), each metaclass need only be created once. A graphical illustration of this is seen in Figure 7.2, where <<metaclass>> Message occurs only once, but is extended twice: through the <<Stereotype>> Failure Effect and <<Stereotype>> Recommended Action. The figure also shows how a metaclass is connected to a stereotype through an extension-arrow.

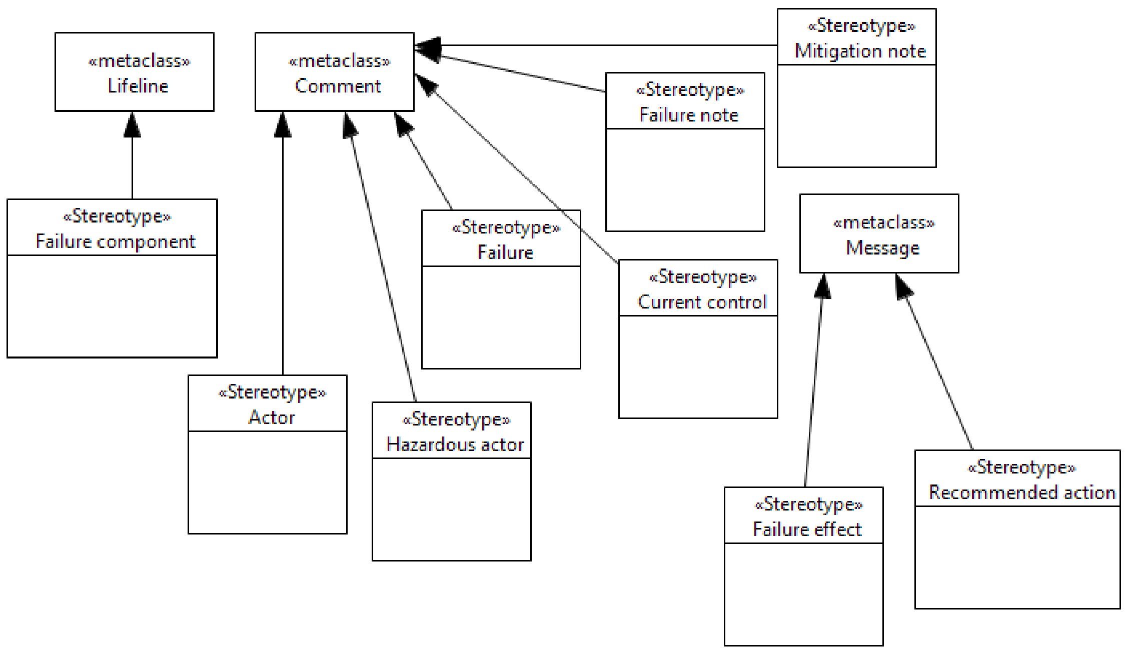
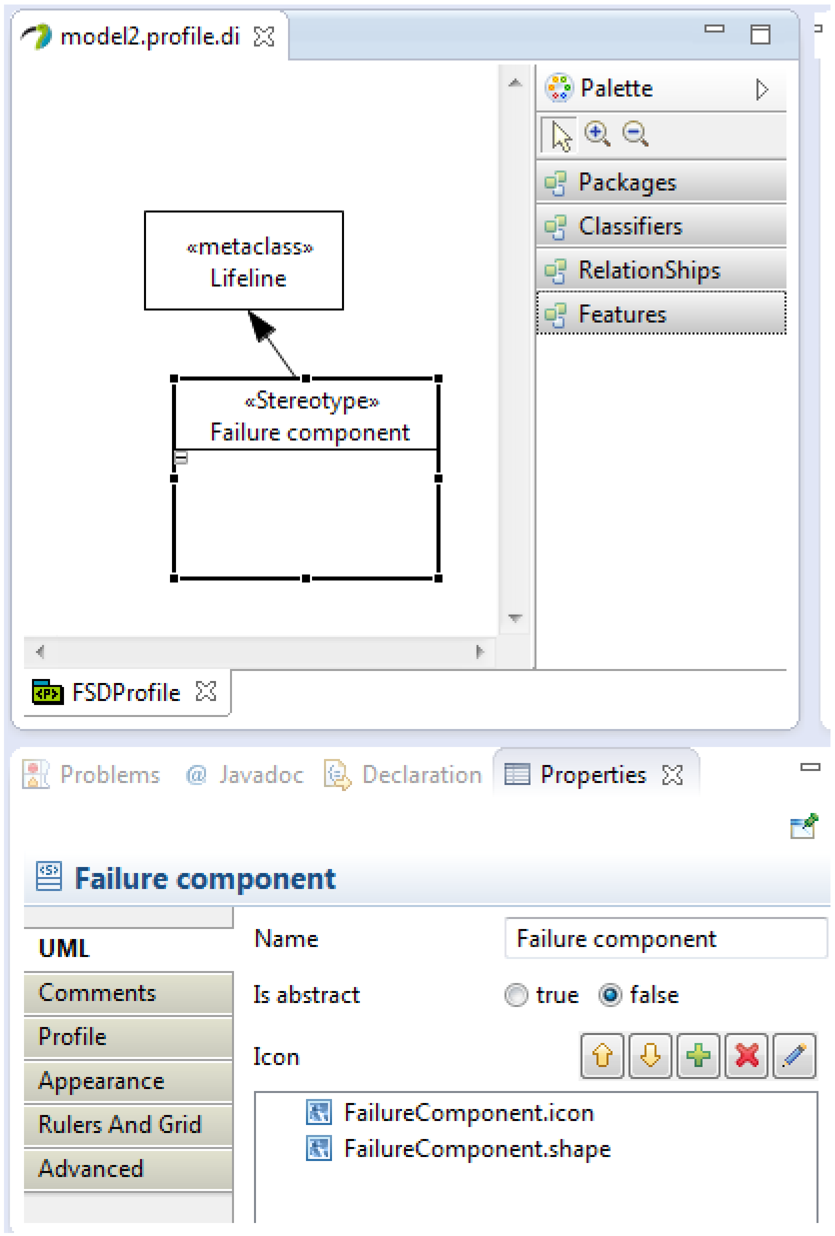

«metaclass» Lifeline
«metaclass» Comment
«Stereotype» Mitigation note
«Stereotype» Failure note
«Stereotype» Failure component
«Stereotype» Failure
«Stereotype» Current control
«metaclass» Message
«Stereotype» Actor
«Stereotype» Hazardous actor
«Stereotype» Recommended action
«Stereotype» Failure effect

Figure 7.2: Graphical presentation of first edition of FSD Profile

All FSD notation will require the corresponding stereotype to include both a icon and a shape image of the notation. The icons support HL7 in Table 6.1, as they provide the means to present the notation in the palette (see section 7.4) in a graphical manner in addition to the textual one. FAH1-FAH2, FMN1-FMN2, FMN7-FMN8, FC1 and FL1-FL4 presented in Tables 7.1-7.5 are all met when the stereotypes are included with shapes. In the stereotypes, both images are added as Attributes, as can be seen in an example in Figure 7.3. When all the metaclasses are created and extended and necessary attributes have been added, the model is validated to ensure its syntactical correctness.



Failure component

Name: Failure component

Is abstract: true [✓ false]

Icon:
- FailureComponent.icon
- FailureComponent.shape

Figure 7.3: How icon and shape are added to a stereotype

7.3 Second edition FSD UML profile

As anticipated with a method as new as the CHASSIS method is, there was an update of the notation for the FSD; removing notation (the Actor and Hazardous Actor), adding new components (Message Lost, Message Found, Part Component Failure, Alternative and Parallel Failures and Mitigation and Failure Interactions) and editing existing notation (Failure Component, Failure Effect, Recommended Action, Failure and Current Control). The new notation is presented in Figures 7.4,7.5 and 7.6. The new notation meant a second round with establishing notation defining requirements, and translating these into UML Profile components. The modifications made to the first-edition profile described in section 7.2 are stated below. But first, a few comments on the existing collection of functional requirements (see section 7.1).

1. Failure Component is now called Complete Component Failure
2. Recommended Action is now called Message Mitigation
3. Failure Effect is now called Message Failure

Failure has changed name to failure indication and Current control to mitigation indication. In addition, these two notations no longer include a text field for failure item number and mitigation control number, respectively. Fmn3, FMN4, FMN5, and FMN6 in Table 7.2 are therefore no longer considered functional requirements.

Finally, FC4 and FC5 in Table 7.4 are no longer considered as the second version of the FSD UML Profile applies A Part Component Failure functionality by extending the UML Execution Specification.

7.3.1 Functional requirements - second version

Removing Actor and Hazardous Actor

When comparing the first edition notation of FSD to Figure 7.4, 7.5 and 7.6, you will notice the absence of both actor and hazardous actor. As a notation, actor and hazardous actor are the result of inspiration from Misuse Case (MUC) notation Misuser. In FSD, the purpose of the two actors where representing an operator/user or an external system affecting the system under analysis. However, after with Raspotnig, the CHASSIS domain expert, is was decided that the use of Mitigation and Failure Notes would be better at describing the external component effecting the system. As for the requirements in 7.1, removing these notations therefore invalidates Table 7.1 and 7.5.

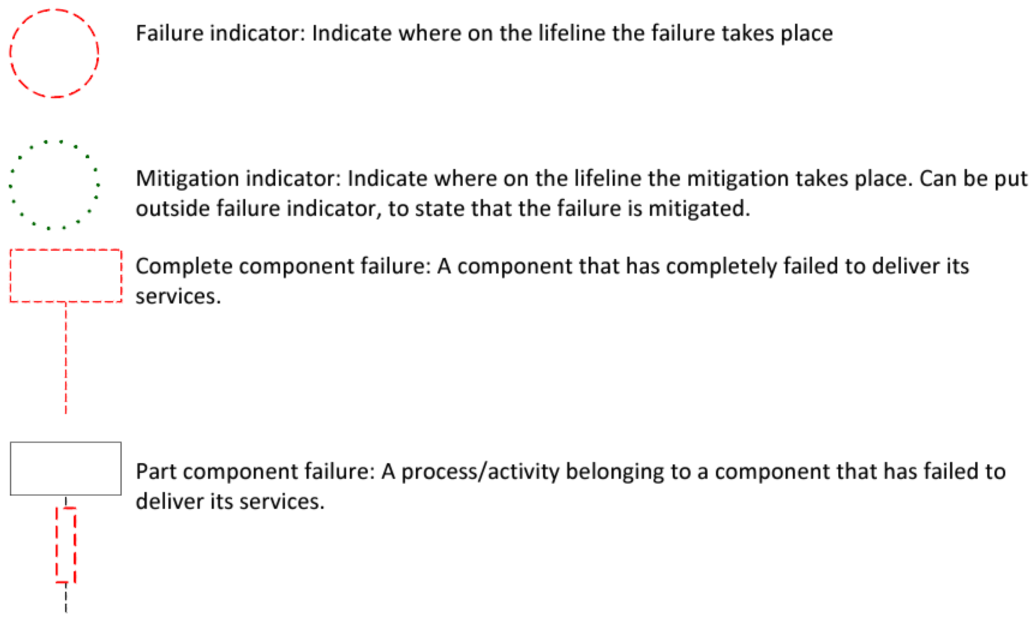

Failure indicator: Indicate where on the lifeline the failure takes place
Mitigation indicator: Indicate where on the lifeline the mitigation takes place. Can be put outside failure indicator, to state that the failure is mitigated.
Complete component failure: A component that has completely failed to deliver its services.
Part component failure: A process/activity belonging to a component that has failed to deliver its services.

Figure 7.4: Indicators and part/complete component failure in second version of FSD notation

Part Component failure

Part Component Failure is created by extending the UML Execution Specification, and thus letting the Part Component Failure inherit all its functionality. According to [30], this UML metaclass *is a specification of the execution of a unit of behavior or action within the Lifeline.* In other words, this notation is used to specify when a Lifeline or a Complete Component Failure is active. In Figure 6.1, you see two execution specifications, both represented by oblong squares on top of the tail of Lifeline A and B. In addition, the figure shows how messages linking the two Lifelines together go back and forth via the Execution Specification and not from Lifeline-tail to Lifeline-tail as shown in Figure 3.3.

Id	Requirement
FPC1	Part component failure shall be represented by a red dotted square
FPC2	Part component failure shape shall be placed on a lifeline tail
FPC3	A part component failure shall be the source of 1. a message failure 2. a message mitigation or 3. a message lost
FPC4	A part component failure shall be the target of 1. a message failure 2. a message mitigation or 3. a message found
FPC5	A part component failure should have eighter 1. none 2. one or 3. several messages as sources
FPC6	A part component failure should have eighter 1. none 2. one or 3. several messages as targets
FPC7	It should be possible to change a UML Execution Specification into a part component failure without having to delete the UML Execution Specification
FPC8	It should be possible to change a part component failure into a UML Execution Specification without having to delete the part component failure

Table 7.7: Functional requirements of part component failure

Similar to requirements related to graphical presentation of notation in section 7.1, FPC1 is fulfilled by adding a shape and an icon to the Part Component Failure stereotype. One of the associations related to an Execution Specification is a reference to which Lifeline the notation covers, meeting FPC2 through the extension. FPC3 and FPC4 are set in the associations of the message going between two Execution Specifications; *sendEvent* and *recieveEvent*. The number of sources and/or targets associated to a Part Component Failure (FPC5 and FPC6) is set in yet another association it possesses. FPC7 and FPC8 are met by including an Enum in the UML Profile diagram (see section 7.3.2).

Messages

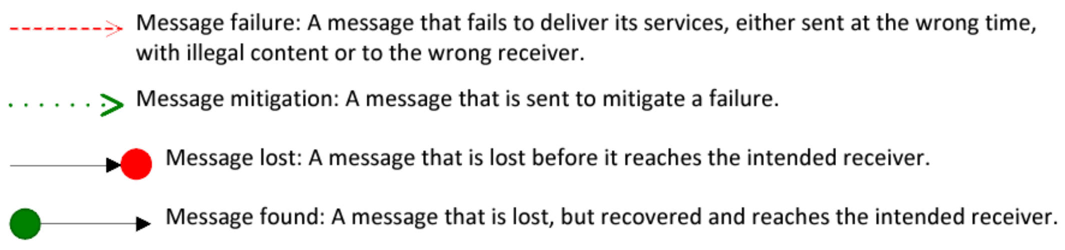

Message failure: A message that fails to deliver its services, either sent at the wrong time, with illegal content or to the wrong receiver.
Message mitigation: A message that is sent to mitigate a failure.
Message lost: A message that is lost before it reaches the intended receiver.
Message found: A message that is lost, but recovered and reaches the intended receiver.

Figure 7.5: Messages in second version of FSD notation

The Message Lost and Message Found stereotypes are the result of the extension of the UML Message Lost and UML Message Found, respectively. Both of the stereotypes contain icons and shapes so the links are fulfilling FM1 and FM2. As with the extension of UML Message Async in section 7.1, FM3-FM6 are met through associations included in the metaclass' functionality. FM3, as shown in Table 7.8, does not say anything about the target of Message Lost. As the message's name alludes, this message has no source as it never arrives at its destination. In the same way, the Message Found has no source as its origin is unknown (FM4). Allowing the editing of a message-types without deletion requirements, FM7-FM11, are met by introducing Enums (see section 7.3.2).

Id	Requirement
FM1	Message lost shall be represented as a black solid arrow with a red circle at the tip of the arrow
FM2	Message found shall be represented as a black solid arrow with a green circle at its tail
FM3	Message lost should have 1. Execution specification or 2. Part component failure as source
FM4	Message found should have 1. Execution specification or 2. Part component failure as target
FM5	Message lost should have an argument
FM6	Message found should have an argument
FM7	It should be possible to change a UML Message Async into a 1. Message Lost 2. Message found 3. Message failure or 4. Message mitigation without having to delete the UML Message Async
FM8	It should be possible to change a Message Lost into a 1. UML Message Async 2. Message found 3. Message failure or 4. Message mitigation without having to delete the Message Lost
FM9	It should be possible to change a Message found into a 1. UML Message Async 2. Message Lost into 3. Message failure or 4. Message mitigation without having to delete the Message found
FM10	It should be possible to change a Message failure into a 1. UML Message Async 2. Message Lost into 3. Message found or 4. Message mitigation without having to delete the Message failure
FM11	It should be possible to change a Message mitigation into a 1. UML Message Async 2. Message Lost into 3. Message found or 4. Message failure without having to delete the Message mitigation

Table 7.8: Functional requirements of message lost, found, failure and mitigation

Adding alternative/parallel failure and interaction

Through experiments investigating how FSD (see section 3.3) could be used to support FMEA, documented in [6, 2], a limitation in FMEA (see section 4.1.2), namely its inability to model multiple failures, was pointed out. As it turned out, FSD shared the same limitation; modeling multiple failures in a single FSD would result in the diagram becoming to complex. In an effort to try and resolve this issue, one of the experiments in [2] included a new notation: the Alt interactionOperator. This notation is a part of the semantics of the UML CombinedFragment metaclass, allowing the representation of a choice of behaviours/paths in a SD.[30] The use of the Alt operator showed promise in [2], and justifies adding the Alternative Failure-notation in the second edition of FSD. Parallel Failure, an extension of the Parallel interactionOperator, is added for the same reason[2], and provides the means to display graphically how multiple paths in SD can occur simultaneously[30] and how these may propagate differently through a system.

Yet another feature aiming at reducing the complexity, and hopefully increasing the readability, of a SD is discussed in [2, 5]. The feature, InteractionUse, allows the analyst to define certain interactions as separate diagrams, and then being able to reference them form other diagram instances.[2, 30] As a result, Mitigation and Failure Interaction notation are added to the second edition of FSD.

Alternative/Parallel failures and interaction

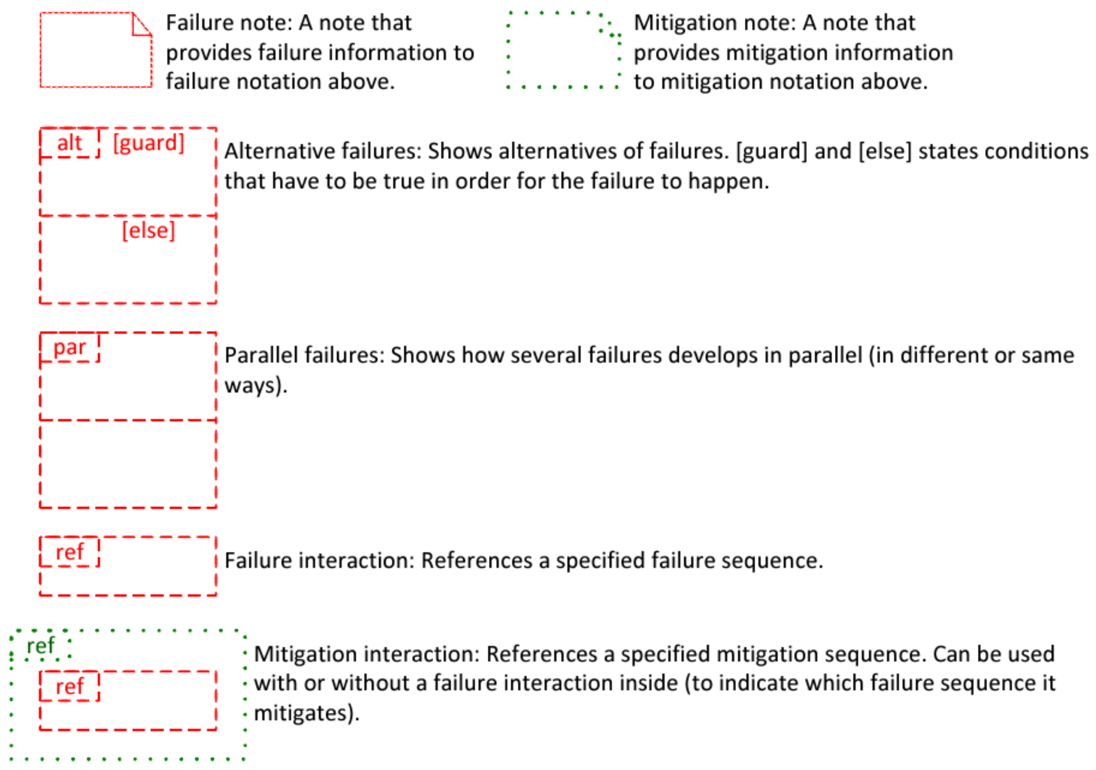

Failure note: A note that provides failure information to failure notation above.
Mitigation note: A note that provides mitigation information to mitigation notation above.
alt
[guard]
[else]
Alternative failures: Shows alternatives of failures. [guard] and [else] states conditions that have to be true in order for the failure to happen.
par
Parallel failures: Shows how several failures develops in parallel (in different or same ways).
ref
Failure interaction: References a specified failure sequence.
ref
ref
Mitigation interaction: References a specified mitigation sequence. Can be used with or without a failure interaction inside (to indicate which failure sequence it mitigates).

Figure 7.6: Notes, parallel/alternative failures and interaction in second version of FSD notation

Both the alternative and parallel failures presented in 7.6 are the result of extending two UML metaclasses: CombinedFragment (CF) and interaction-Operand (IO). The upper half of the figures extends CF and the lower half extends IO into corresponding Combined failure fragment and Interaction failure operand (see Figure 7.7). The graphical requirements presented in RAPF1 and RRAPF2 are both met through including icons and shapes. RAPF3, RAPF4, RAPF6, and RAPF8 are all covered by associations included within CF and IO metaclasses. Requirements stating the placement of text, RaAPF5 and RAPF7, are met through functionality in the Papyrus tool as the FSD profile is applied.

ID	Requirement
RAPF1	Combined failure fragment shall be represented by a red dotted square with an imbedded square in the upper left corner (see 7.6)
RAPF2	Interaction failure operand shall be represented by a red dotted square
RAPF3	It should be possible to connect an interaction failure operand to a combined failure fragment
RAPF4	Combined failure fragment shall have the option of a text field
RAPF5	The text field of the Combined failure fragment shall be located inside the red square
RAPF6	Interaction failure operand shall have the option of a text field
RAPF7	The text field of the Interaction failure operand shall be located inside the red square
RAPF8	The interaction failure operand should be an instance of 1. Alt or 2. Par

Table 7.9: Functional requirements for alternative and parallel failures

According to the OMG UML superstructure[30], an Interaction Use refers to an Interaction. Thus, by creating Failure and Mitigation interaction stereotypes extending the interaction use metaclass, Rfmi3 and Rfmi4 are met. Including icons and shapes in the stereotypes will cover RFMI1 and RFMI2.

ID	Requirement
Rfmi1	Failure interaction shall be represented by a red dotted square with an imbedded square in the upper left corner (see Figure 7.6)
Rfmi2	Mitigation interaction shall be represented by a green dotted square with an imbedded square in the upper left corner
Rfmi3	A failure interaction should be able to reference another interaction through a written link
Rfmi4	A mitigation interaction should be able to reference another interaction through a written link

Table 7.10: Functional requirements for failure and mitigation interactions

Complete Component Failure and Lifeline

As a means to aid in fulfilling HL7 in Table 6.1, FCCL1 and FCCL2 (see Table 7.11) allows the user to change a complete component failure into a UML Lifeline and vise versa without having to delete the shape from the diagram.

Id	Requirement
FCCL1	It should be possible to change a complete component failure into a UML Lifeline without having to delete the complete component failure
FCCL2	It should be possible to change a UML Lifeline into a complete component failure without having to delete the UML Lifeline

Table 7.11: Additional functional requirements for complete component failure

7.3.2 Implementing changes into the FSD profile

New in the second version of the FSD Profile, is the use of Enums. [54] presents an Enum as a special datatype, enabling the predefinition of a variable to a set constant. In this setting, the Enum allows the user to edit a shape instead of having to first delete, then regenerate it in the diagram.

Comparing the graphs representing the first (Figure 7.2) and second (Figure 7.7) FSD UML Profile, it is clear that applying the Enum datatype leads to significant changes in the structure of the profile. Equating the extension of UML::Message shows that the message extension in version two is replaced by a single stereotype (as compared to the <<Stereotype>> Failure effect and <<Stereotype>> Recommended action). This stereotype, namely <<Stereotype>> Messages is through the included Enum, permitted to have one of the set values (normal, lost, found, failure or mitigation). As the stereotypes of the profile is applied to a diagram, which for the <<Stereotype>> Messages for instance will occur five times (once for each value in the Enum), the correct constant will be selected for the appropriate notation. Applying a profile is described in further detail in section 7.5. But before a profile can be applied, the custom palette to which it is to be applied to must be defined.

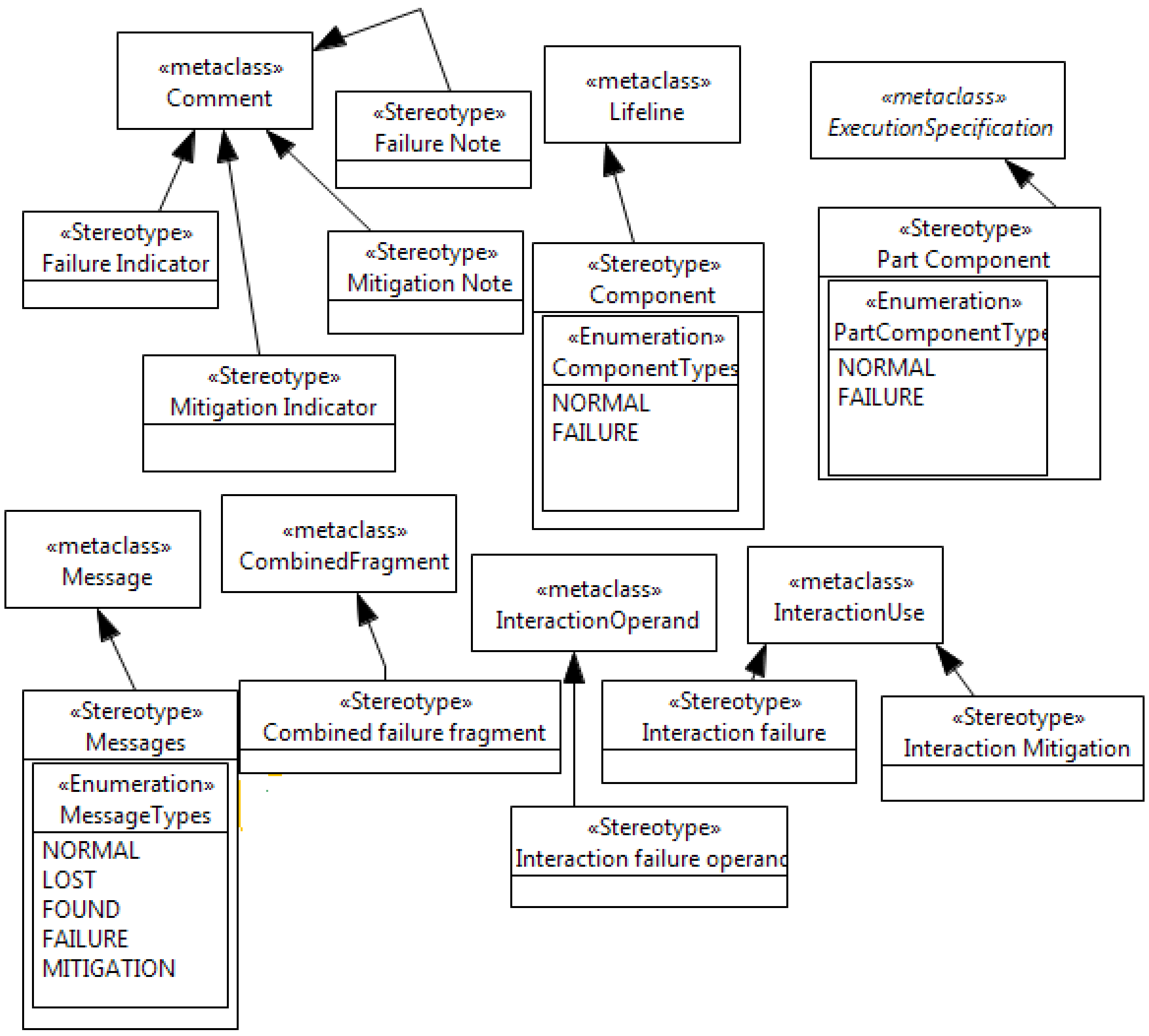
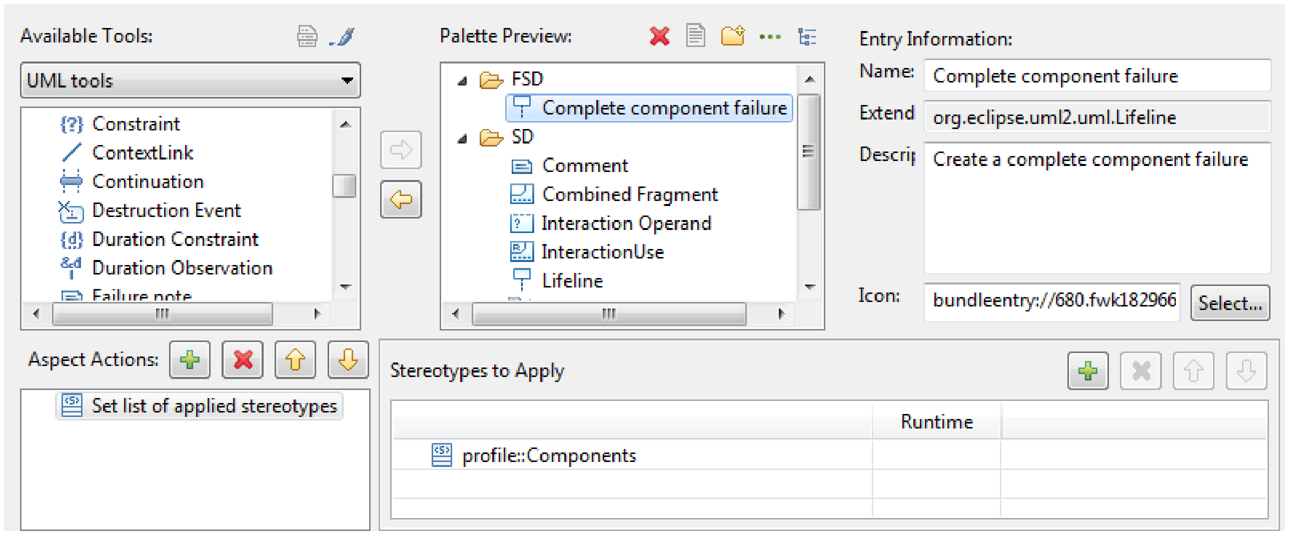

«metaclass» Comment
«Stereotype» Failure Note
«metaclass» Lifeline
«metaclass» ExecutionSpecification
«Stereotype» Failure Indicator
«Stereotype» Mitigation Note
«Stereotype» Component
«Enumeration» ComponentTypes
NORMAL
FAILURE
«Stereotype» Part Component
«Enumeration» PartComponentType
NORMAL
FAILURE
«Stereotype» Mitigation Indicator
«metaclass» Message
«metaclass» CombinedFragment
«metaclass» InteractionOperand
«metaclass» InteractionUse
«Stereotype» Messages
«Enumeration» MessageTypes
NORMAL
LOST
FOUND
FAILURE
MITIGATION
«Stereotype» Combined failure fragment
«Stereotype» Interaction failure
«Stereotype» Interaction Mitigation
«Stereotype» Interaction failure operand

Figure 7.7: Second version of the FSD profile

7.4 Creating a custom palette

In this setting, a palette is a section in the diagram editor itself where all the notation of FSD and SD (HL8 and HL9) are listed. In Papyrus, the palette is created by selecting notation from a complete list of UML notation. Since SD is the base when creating FSD (see section 3.3), the notation list is limited by Papyrus to SD notation only. Supporting HL7, the notation of SD and FSD are separated into two subfolders in the palette. Figure 7.8 displays the process of adding notation into the palette. The next step now is applying the FSD UML Profile and its stereotypes to the appropriate notation in the custom palette.



Name: Complete component failure
Extend: org.eclipse.uml2.uml.Lifeline
Descrip: Create a complete component failure
Icon: bundleentry://680.fwk182966

	Runtime	
profile::Components		

Figure 7.8: Creating a custom palette, showing the complete component failure and its related stereotype

7.5 Applying the profile

Applying a stereotype to a notation will, as mentioned previously (see 5.6.2) cause the notation to incorporate the additional functionality stated in the stereotype. Using the Papyrus tool, stereotypes are applied in the same window as the custom palette is created. Figure 7.8 shows the Complete Component Failure-notation and that it is created using a Lifeline extended with the Component stereotype.

In order to meet requirements FPC7 and FPC8, the execution specification, located in the SD subfolder in the palette, apply the Part Component stereotype (see Figure 7.7), tagged with the NORMAL value of the stereotypes Enum. There are no additional functionality connected to the NORMAL constant; only the indication that the execution specification is an instance of the Part Component. The part component failure notation in the palette will also apply the same stereotype, but here the new functionality required are added and FAILURE constant is selected. FM7-FM11, FCCL1 and FCCL2 are realised using the same technique.

According to [31], the official tutorial for profiling with Papyrus, creating the profile and custom palette and applying the profile should result in a customized tool. Here however, that turned out not to be the case. Only about half of the notation, both with and without applied stereotypes, where visible in the palette. After several days of troubleshooting and extensive Papyrus forum searches a sad fact had to be realised: Several parts of the Papyrus program logic code for extending SDs where still under development, not to be made available to the public until late June 2014. The shortfalls would result in Tables 7.2, 7.3, 7.6, 7.8 and parts of 7.9 all being void. Suddenly and quite unexpected, the project had to take a new direction.

Inspired by the approximations to a CHASSIS tool presented by the safety experts from Avinor (see section 5.1), the a second tool approach to CHASSIS would, instead of focusing on a syntactical complete diagram editor, focus on a more low level, mostly syntactic-free tool aiding the CHASSIS method during brainstorming activities. For this second approach, the need for a different development tool arose. As described in section 5.6.2, a stereotype extending a metaclass is forced to inherit all of the functionality of that metaclass. This solution would impose to much restrictions on the tool, and as . As a result the Graphical Modeling Framework (GMF) was chosen as the preferred development tool for the second CHASSIS tool approach.

Details about GMF, the requirements of the second tool approximation, how they are realised with GMF and the implementation of the tool is described in section 7.6.

7.6 The CHASSIS artefact - GMF approach

7.6.1 Graphical Modeling Framework

Like Papyrus, the Graphical Modeling Framework (GMF) is a component in the Eclipse Modeling Project (EMP).[69] The means of GMF is providing a bridge between two other projects in EMP: the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). Here, the EMF facilitates the creation of a structured tool data model and the generation of executable model code presenting the syntax of the tool.[70] The graphical editor, including the visual content of the palette, placed on top of the model code, and thus creating a tool with backhand logic and a visual front, is realised using GEF.[71] The bridge, GMF, facilitates connecting the correct parts of model code with the appropriate visual representation figures in the editors palette.

Table 6.3 (see page 49) compares GMF with Papyrus and VSU. Here, GMF was presented as a weaker choice than Papyrus when the task was creating a strict UML based tool: GMF has no built in model for UML. The developer is therefore forced to create the complete domain model, including how all components in the model relates to each other and constraints, from scratch. When creating a syntactically complex tool such as the CHASSIS tool in 7.3, using GMF as a development tool puts tremendous pressure on the developers domain knowledge. All the details in the will need to be implemented correctly in order to call the tool a full UML editor for SD and FSD (HL1, HL8, Hl9). For a tool such as another approximation suggested by safety experts at Avinor (see section 5.1), the tool this section describes, defines and implements, however, GMF is the strongest development tool candidate. The limitations of GMF not implementing the formal UML model is no longer an issue, as the new approximation to a CHASSIS tool involves developing a nearly constriction-free brainstorming tool aiding the CHASSIS method.

7.6.2 Functional requirements

As with the first and second version of the profile diagram for FSD, the functional requirements (FR) constituting the domain model for the GMF version of the second CHASSIS tool approximation are based on the same set of information sources. (see Chapter 5) In addition, the functional requirements of the first and second version of the CHASSIS approach inspired the tables containing the FRs for the second approximation to the CHASSIS tool. Table 7.12 and 7.13 present the requirements inherited from the first and second version of the CHASSIS approximation using Papyrus and the new requirements the existing ones did not cover.

Id	Requirement
FMN1	Failure indicator shall be represented by a red dotted circle
FMN2	Current indicator shall be represented by a green dotted circle
FMN9	Mitigation note shall contain a text field
FMN10	Failure note shall contain a text field
FC1	Complete component failure shall be represented by a red square and a red tail
FC2	Complete component failure shall have the option of a text field
FPC1	Part component failure shall be represented by a red dotted square
FPC5	A part component failure should have eighter 1. none 2. one or 3. several messages as sources
FPC6	A part component failure should have eighter 1. none 2. one or 3. several messages as targets
FPC7	It should be possible to change a execution specification into a part component failure without having to delete the execution specification
FPC8	It should be possible to change a part component failure into a execution specification without having to delete the part component failure
FCCL1	It should be possible to change a complete component failure into a component without having to delete the complete component failure
FCCL2	It should be possible to change a component into a complete component failure without having to delete the component

Table 7.12: Functional requirements inherited from of the CHASSIS tool - Papyrus version

Comparing Table 7.12 and 7.13 to the numerous tables presenting the functional requirements describing the first and second version of the CHASSIS approach in Chapter 7, there is an obvious differentiation between the syntax-level of the two. As this second CHASSIS approximation is environed as a tool excluding all no crucial syntax, high level requirements HL2, HL3, HL6, HL8 and HL9 in Table 6.1 are considered no longer valid in order to meet the goal of tool creation.

Id	Requirement
FGMF1	Mitigation note shall be represented by a green box
FGMF2	Failure note shall be represented by a red box
FGMF3	It should be possible to attach text to a Message
FGMF4	Component shall be represented by a black square and a black tail
FGMF5	A message shall be represented by a black arrow
FGMF6	A message should be able to have a 1. execution specification or 2. part component failure as a source and a 1. execution specification or 2. part component failure as a target
FGMF7	A 1. part component failure or 2. Execution specification shall be the source of a message
FGMF8	A 1. part component failure or 2. Execution specification shall be the target of a message
FGMF9	An execution specification should have eighter 1. none 2. one or 3. several messages as sources
FGMF10	An execution specification should have eighter 1. none 2. one or 3. several messages as sources
FGMF11	Failure fragment shall be represented by a red box
FGMF12	Failure fragment should have a text field in the upper left corner
FGMF13	Failure operand shall be represented by a red box
FGMF14	Failure operand should have a text field in the upper left corner
FGMF15	Failure interaction shall be represented by a red square
FGMF16	Mitigation interaction shall be represented by a green square
FGMF17	It should be possible to attach text to a failure interaction
FGMF18	It should be possible to attach text to a mitigation interaction

Table 7.13: Additional functional requirements - GMF version

The first obvious difference between the CHASSIS approach using Papyrus and the approximation using GMF is message types. The first-mentioned presents four different message types (see Figure 7.5). [72] presents brainstorming as a spontaneous and unsecured activity. The second approach therefore only contains one message-type as the means to simplify the tool palette and further promote unhindered creative thinking. The remaining notation in FSD as presented in section 7.3 are visually the same in the second tool approximation despite a simplification of their former constraints.

7.6.3 Implementation

Implementing a tool using GMF is preformed by going through the steps presented in Figure 7.9. The CHASSIS tool editor creation begins with the construction of a model that will cover all the functional requirements stated for the second approximation to the CHASSIS tool: the domain model.

In GMF, the domain model is created using a core construct at the heart of EMF called Ecore, providing class, attribute and reference elements.[73] At the center of the domain model is a class named Interaction. This class has no attributes, and serves mainly as a class connecting all the surrounding classes describing the actual notation of the CHASSIS tool together. Of the surrounding classes, the definition of a component and a complete component failure deserves some extra attention. In order to mimic a real life sketching-tool, both component and complete component failure are created using two classes in the domain model: a component head (containing the title) and a tail (having messages as in- and output). In order to fulfill FGMF3, FGMF17 and FGMF18, the Comment is created in the domain model, thus allowing the placement of a text field anywhere in the diagram editor. Additionally, FM9, FM10, FC2, FGMF12 and FGMF14, leads to the failure and mitigation note, component and complete failure component head and the failure fragment and operand are equipped with a text field attribute automatically. Facilitating the creation of connections between figures in the diagram editor, FPC5, FPC6 and FGMF6-FGMF10 are met by drawing reference elements between the appropriate classes in the domain model. In section 7.3, the Enum construct was implemented in order to meet requirements stating that the user should be able to change certain notation in the editor window without having to remove the notation first. (FP7, FP8, FM7-FM11 and FCCL1-FCCL2) In the second tool approach, the still relevant requirements stating this type of functionality was met in a different way. The tool permitted stacking figures in the editor window, so that placing a figure directly on top of an existing one would meet the requirements.

Now the construction of the domain model is complete and the model is validated and model code is generated (Domain Gen Model in Figure 7.9). This concludes the EMF part of the tool creation. The next step is describing the functionality of the figures and the layout of the palette: GEF.

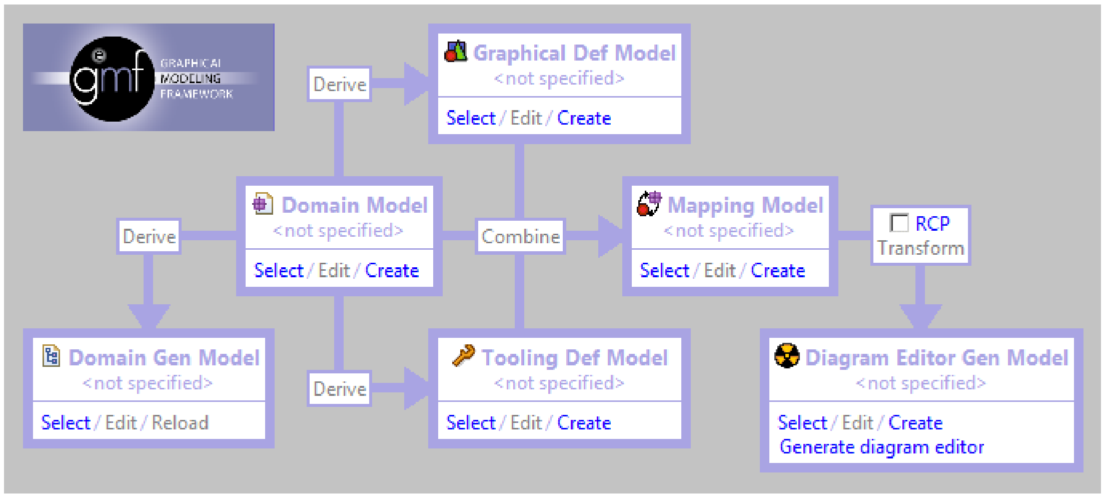


Figure 7.9: The steps taken in the development of a GMF project[73]

The palette of the CHASSIS approximation will contain the means to create the notation presented in Appendix C. Each of these are individually defined in a graphical definition model (see Figure 7.9). This decision contradicts a statement made by one of the safety experts during the Avinor meeting. (see section 5.1) The expert commented that three figures, a simple link, circle and square would be more than sufficient when drawing a diagram during a brainstorming session. With only three figures in the palette whilst needing to draw a total of sixteen shapes (see the graphical notation for the tool in Appendix C) could lead to unclear affordance. Here, affordance entails the implying context of use and functionality of an object by observing its sensory characteristics.[74] Having one entry for each of the figures in the palette will help clarify the affordance, the intended use, of each shape.

In the graphical model, the definition of each notation includes amongst others the size, line thickness and colour of the figure. Moving on to the actual palette, the notation can be sorted in a sensible order and given appropriate names. (HL7) The completion of the graphical and tooling definition, these two along with the former created domain model are all mapped together. Final adjustments, assuring that each domain class, graphical definition and palette entry are correctly connected. The mapping model is then used to generate the diagram editor code. The resulting code base contains an executable program file of the FSD editor.

The second phase of the design science research method, construction, is now completed. Chapter 8 describes the next step: artifact analysis, here through usability testing.

Chapter 8

Artefact analysis - student experiment

This chapter presents the plan, conduction and results of the analysis phase: student experiment.

8.1 Scope and purpose

The purpose of the student experiment is testing how the CHASSIS tool described in 7.6 functions in a realistic setting. As mentioned earlier, the typical setting where such a tool is used will be in brainstorming-meetings. With this as a basis, the test will elucidate and hopefully fulfill the functional requirements listed in 7.6 and the non-functional requirements listed in Table 8.1.
The main focus was not on how the participants interpreted CHASSIS as a method, but rather how the CHASSIS tool was utilized in a brainstorming-setting.

Id	Requirement
NF1	It should be easy for a user to draw a figure from the palette
NF2	A user should easily be able to separate notations from each other when notation is presented graphicly
NF3	A user should find it easy to locate a specific notation in the palette
NF4	It should be easy for a user to edit existing figures in a diagram
NF5	The visualizations should act as a motivator for the brainstorming

Table 8.1: Non-functional requirements for the CHASSIS tool

8.2 Location, equipment and process

The student experiment was conducted in a meeting room at Norwegian University of Technology and Science, campus Gløshaugen. Here, participants where seated next to each other, facing a 21" computer screen with a accompanying data-mouse and keyboard. The test leader was seated next to them, facing both students. The screen was attached to a laptop, facing the test leaders way. This setup permitted the leader to take screenshots of the editor window during the sessions without having to disturb the participants' workflow. Audio recordings of of each session was made so that participant-feedback could be reviewed at a later time.

In order to conduct a structured experiment, and to ensure that all participants have the same starting point, an agenda for the testing was created:

1. Introduction to CHASSIS, FSD and the CHASSIS tool
2. Brief test-scope presentation
3. Presenting the scenarios
4. Test session - participants carries out scenarios using the CHASSIS tool
5. Discussion and Retrospective probing (RP)

Starting with an introduction, the CHASSIS method and it's purpose was explained to the participants. Continuing with FSD, the participants was introduced to the notation and provided with a sheet of paper containing the names of the FSD-notation and a graphical representation of the figures. Moving on to the CHASSIS tool, it's intended use as a supporting tool when brainstorming as an activity to elucidate and extract functional requirements in a system was explained.

Presenting the scope of the test included giving participants an overview of the non-functional requirements that their attendance was meant to cover. In addition, the participants where informed that only the CHASSIS tool, and not the participants performance, where being tested. Next, the participants was asked, to the extent of their ability, to utilize the concurrent thinking aloud (CTA) technique, thus encourage them to keep a constant stream of consciousness and think out loud during the experiment.[52] They where in addition informed that they could ask questions during the experiment should anything be unclear.

The next step on the agenda was presenting the scenarios (see 8.3). Here, the participants where handed a sheet of paper containing a short story presenting the tasks at hand, and the participants began working through the story. Additionally, the participants where given a sheet of paper displaying the FSD notation graphically.(See Appendix C)

After completing the testing session, the participants where encouraged to discuss the scenarios and how they had chosen to implement them using the tool. Using RP, a technique where participants are asked about their thoughts and actions during the session, provides the opportunity for asking participants follow-up questions. Each session lasted 1,5 hours.

8.3 Scenarios

[53] present three different approaches to scenario-creation. One of the tree, Goal- or Task-Based Scenarios (GTBS), only states what the user wants to do. During a brainstorming, GTBS gives the participants a reason and a goal for the activity, but at the same time leaving dictions regarding how a task is to be performed up to the participants. During an experiment, this gives valuable information as to the intended use as well as patterns for use. The following numbered list represents an English version of the scenarios handed out to the participants. For the original Norwegian version, see Appendix E.

1. A company, Duck AS, has all it's company data stored in a server. This server can distribute data through a router to two different clients, two computers located in the company's facilities. Primarily, the router communicates with the clients via cable, but may switch to a wireless option should a routing error occur.

2. Lately, Duck AS has had issues with the router; it fails to send messages/communicate with the clients. In an effort to try and resolve the issue, the company decides to take a closer look at how the router-component functions. The router consists of a send-, receive- and package-management-component.

3. Following a discussion, the employees in Duck AS agrees to introduce a watch-dog on the package-management-component. It's task is to send out heartbeats to the send- and receive-component. When the watch-dog notices that the send-component is hung up, it will send a mitigating message to the send-component, telling it to reset. Now, the package-management-component will switch to a wireless connection and resend the last messages should an error occur.

8.4 Participants

All participants in the experiment where students between the age 21-25. In order to mimic a realistic brainstorming situation, participants where paired up and asked to carry out the scenarios together. Even though CHASSIS' main application area is in aviator industry brainstorming situations[3], drawing SD and FSD does not require the user to be a safety domain expert, as basic knowledge about the concepts of UML SDs and FSD notation. Student pairs where therefore selected based on these premisses, and knowledge about safety-modeling was not stated as a requirement. Each of the pairs in the sessions knew each other already, aiming at creating a relaxed atmosphere during the experiment. Table 8.2 presents basic information about the participants in each of the three sessions.

Session ID	Participant ID	Gender	Age
S1	P1	Female	21
	P2	Female	21
S2	P3	Female	24
	P4	Male	25
S3	P5	Male	25
	P6	Male	23

Table 8.2: Basic information about participants in the sessions

8.5 Results

Participants in all sessions drew a router and two clients as a part of the sequence diagram in scenario 1. They also added part components and messages going between the components. (see Figure 8.1) In one session, the participants included components for a server from the beginning. Participants in one of the sessions did not draw a server as a component in the diagram. One pair did draw the server, but did not draw any links between the Server component and the Router until after the completion of the second task.

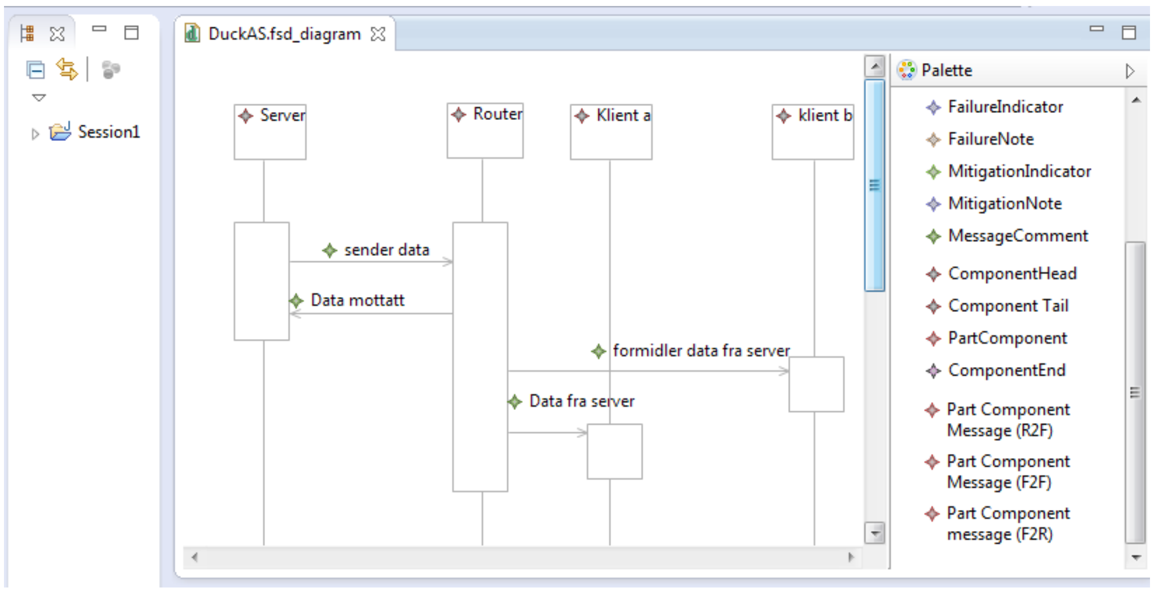


Figure 8.1: Resulting sequence diagram after completing the first scenario during session 1

In all three sessions the participants drew messages indicating messages going from a router to either one or both of the clients. Two pairs followed up scenario 1 by drawing the situation where the router fails to deliver a message (first part of scenario 2) by editing the title of the message and including a failure indicator over the head of the message. The last pair placed failure indicators around the heads of the messages and a failure note next to the failure indicators and wrote that the router failed to deliver a message to the client (see Figure 8.2). None of the pairs marked the Router-component as a failing component in any way.

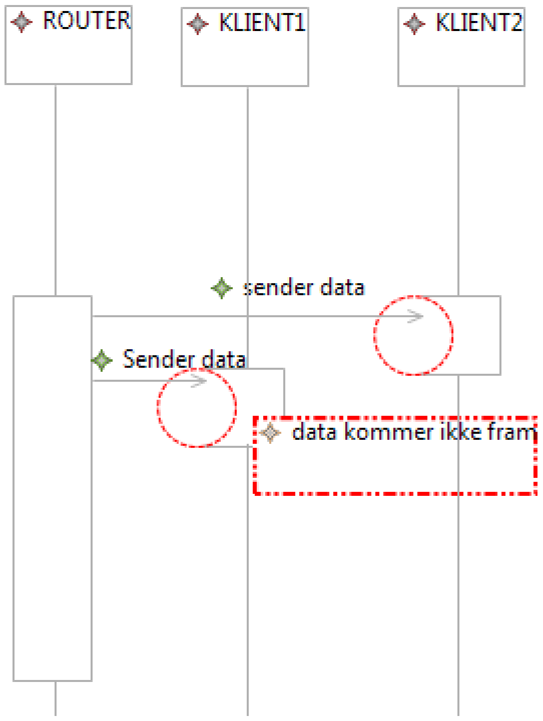


Figure 8.2: A version of how the router failing was visualized during one of the sessions

One of the pairs immediately created a new diagram-file when they where going to decompose the router component (second half of scenario 2). Another team started decomposing the router directly in the first diagram-file, but created a new file and moved the router components once they where tipped about the failure interaction-figure in the palette.

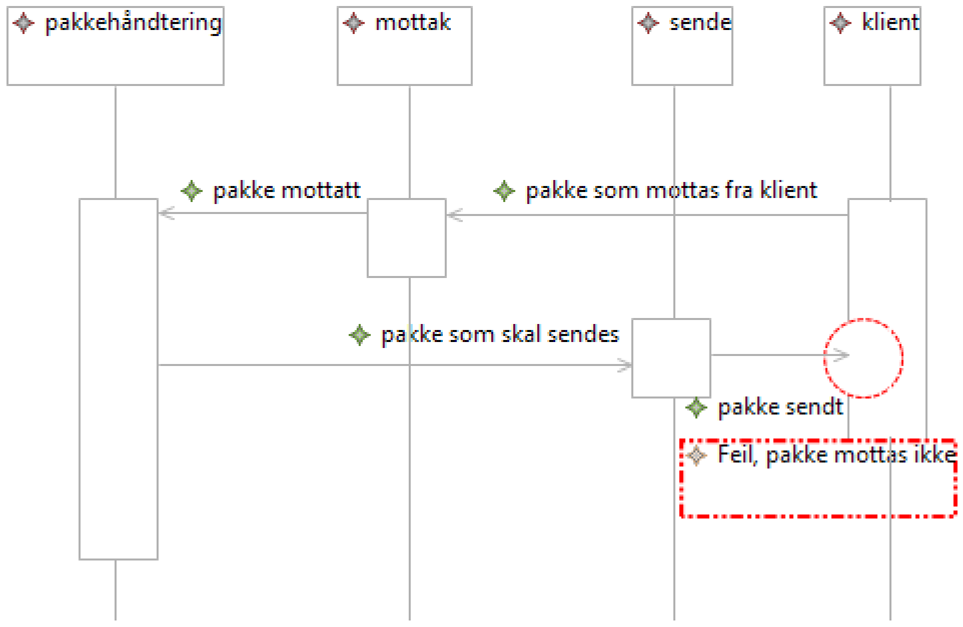


Figure 8.3: Decomposition of the router component

None of the participants in the sessions suggested stacking a part component failure on top of a part component in order to change the one to the other without removing the part component first (FPC7 in Table 7.12). Instead, one session applied a failure indicator covering the execution specification and a failure note explaining the error of that part of the component. During the RP, this pair said they where aware of the part component failure notation, but that they did not want to delete the execution specification and all its connected messages in order to insert a part component failure. Further they stated that marking the specification with the red circle and a note was sufficient. Another session marked the specification with a failure indicator and added a freestanding message next to it, explaining the error occurring. The last session created a part component failure on the lifeline-tail, moved all messages connected to the execution specification over to the part component failure figure and then deleted the execution specification.

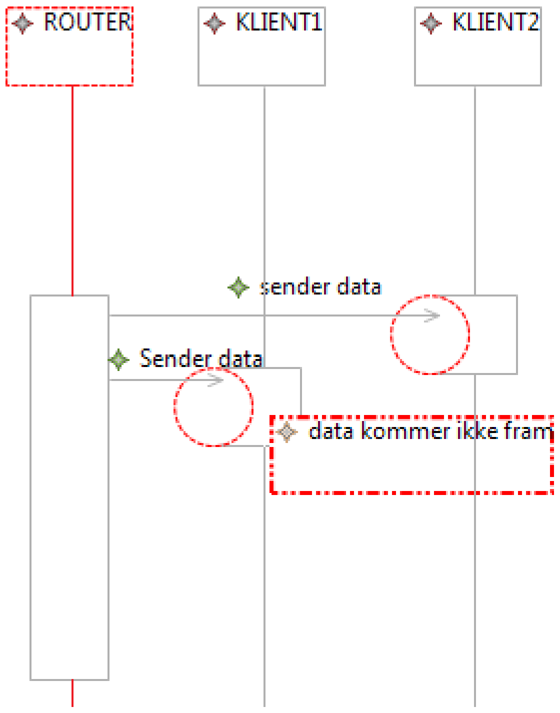


Figure 8.4: Showing Router as a complete component failure

A similar issue as when execution specification needed to become a part component failure was apparent as one session decided that the Router component was an instance of a complete component failure. Here, they created a complete component failure next to the existing router component, moved the execution specification from the old router component to the new, and deleted the old router component. (see Figure 8.4)

The inclusion of the Watch-dog (scenario 3.) was implemented fairly similar in all three sessions, even though only two sessions had a separate diagram for the decomposition of the Router component. Heartbeat messages where drawn between the package-management-component and send-/receive-component. An additional message was placed from the package-management-component to the send-component encouraging it to reset if the component needs to be reset. (see Figure 8.5) Additionally, one pair added mitigation indicators to both heartbeat messages and included a mitigation message.

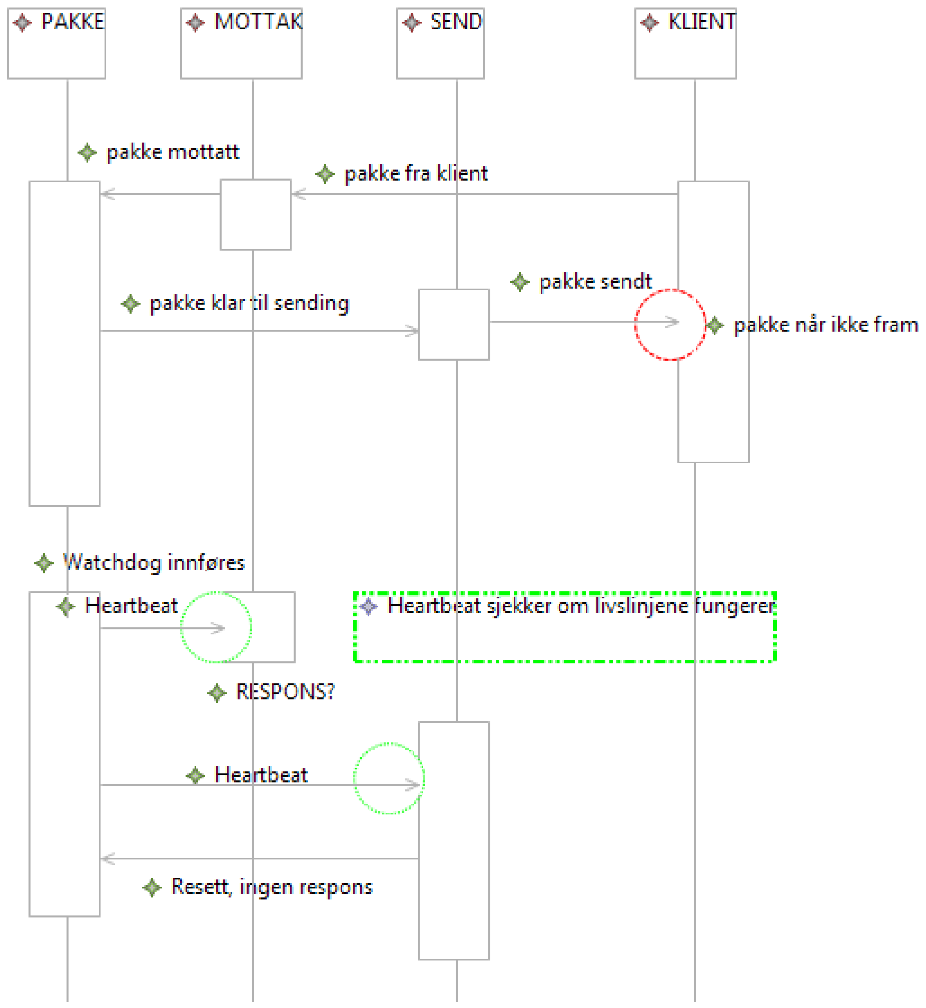


Figure 8.5: Adding a watch-dog to the decomposed Router component

During the RP, the participants gave several statements about the tool. Both pairs in S1 and S2 commented that they found the creation of a component or a complete component failure unnecessary complicated as they had to first draw the head then the tail. The participants suggested a figure containing both the head and tail together as a better solution. On the other hand, all participants stated it was easy to drag a figure to the palette, as well as repositioning, resizing and otherwise edit the shapes. Additionally, participants in all sessions stated that finding a specific notation in the palette was easy. Here, participants in S1 and S3 suggested that including small icons of the figure next to its instance in the palette would aid in visualizing the relevant shape.

A reoccurring issue in S2 was trouble separating failure- and mitigation notation from mitigation- and failure interactions when drawn in the editor window. When asked, they had no suggestions how to improve this issue. S1 and S3 showed similar issues, but used the Comment-notation (see section 7.6.3) diligently during the brainstorming to assure common understanding of the diagram they where creating. When asked, they did not mention the similarity of the notation as an issue. With this, the analysis of the too approach is concluded. Chapter 9 evaluates the CHASSIS tool, as well as the supporting techniques and methods utilized in the report.

Chapter 9

Artefact evaluation and validation

This chapter present the results the tool analysis (see Chapter 8), answers research questions stated in the introduction and evaluates main and supporting techniques and methods used throughout the report. In addition, threats to validation are discussed.

9.1 DSR and Iterative development

The main research method utilized in this report is design science research (DSR). Having a new artefact as its final result, DSR provided a well-structured process supporting the CHASSIS tool approximation: defining the problem and describing a proposal to a problem solution, the construction of the CHASSIS tool, investigating if the requirements of the tool are met and finishing with an evaluation of the results from the artefact investigation. Supporting the DSR method, this project has applied a supporting development method: iterative development method. Despite overlapping with the DSR, the iterative development method enables further detailing of each DSR phase. As a result, the two methods together provided the necessary motivation and structure when conducting software development such as the approximation to a CHASSIS tool. In addition, the iterative process permitted decomposition of each development phase into structured and manageable tasks to be performed several times during the project life cycle.

9.2 Gathering tool requirements

Whereas many techniques used to acquire requirements for a new system development project to great extent utilizes future system users, this project has applied a mostly theoretical approach. The functional requirements related to the CHASSIS tool approximation was based on exploring CHASSIS as a method, mainly through written articles, but also through other written sources: the OMG UML superstructure and other relevant tools and techniques. In retrospect, this theoretical approach to requirement elicitation was more than sufficient as FSDs notation and realistic experiments using FSD in practise was well defined in the articles. Furthermore, the OMG UML superstructure and an expert on the CHASSIS domain, Raspotnig, provided any needed clarifications when unclarities arose.

The answer to the first research question however, *What approximations are there to a CHASSIS tool?*, emerged during a meeting with three safety experts from Avinor. Here, one of the safety experts explained a brainstorming process with first utilizing a low-level sketching-tool during the brainstorming activity and then using a syntax strict tool to clean up the collected data after the conclusion of the brainstorming. Both of these are further investigated in this report and wholly or partly developed and implemented. Together these two suggestions lead to a third CHASSIS tool approximation: a twofold tool allowing the creation of both low- and high-level diagrams. This version is not covered in this report, but is presented as a direction for further work.(see Chapter 10)

9.3 MDSD

Model Driven Software Development (MDSD) present yet another development method supporting the DSR method. One of its key components is metamodeling. One of the tool approximations in this report is a high-level tool including all the syntax of both UML sequence diagram (SD) and CHASSIS' Failure Sequence Diagram (FSD). MDSD, through metamodeling, aids in the creation of a domain specific language (DSL) for FSD by the means of a model extension. Here, existing metaclasses of the UML metamodel is extended through stereotypes, adding the functionality required by the FSD syntax. The extensions are collected in a UML Profile model. MDSD was, for the creation of the high-level approach of a CHASSIS tool extremely valuable. Making the UML metamodel and all its syntax available meant not having to construct all the details FSD directly inherited from the UML SD. In addition, the extension of the UML metamodel would guarantee that all the inherited functionality in the DSL was syntactically correct.

9.4 Development platforms - Papyrus and GMF

As the first tool approximation to CHASSIS was to be implemented, several development platforms where evaluated as possible candidates. Eclipse and the Papyrus tool was chosen based on high level requirements, stated in Table 6.1, where Papyrus, in theory was able to fulfill all. As the FSD UML Profile was to be applied to a UML SD, however, several program core limitations where revealed. The limitations where severe to the degree of over half of the functional requirements of the FSD being void. As a result, the first approach to the CHASSIS tool was abandoned in favor a second approach: a CHASSIS tool supporting brainstorming. For this approach, the Papyrus tool would become to advanced as it forced all functionality and constraints through the metaclass onto the extending stereotype. Using Papyrus to solve the new problem; the low-level brainstorming aid tool would be counterproductive as the approximation to a CHASSIS tool no longer depended on strict UML syntax. The Graphical Modeling Framework (GMF) was instead used as it had no compulsory implemented metamodel. Instead, the domain as a whole is created from scratch, allowing for the creation of a more low-level CHASSIS tool. Evaluating the two up against the tasks they where set to aid, both where fitting solutions for the approximation that was to be constructed.

9.5 Student experiment

The conducted student experiment provides information that answers the second research question: *Does such an approximation work in a realistic setting?* Supporting the answering of this questions are the non-functional requirements states in Table 8.1.

All participants in the three session understood quickly how to drag-and-drop a figure from the palette onto the editor window (NF1). However, S1 and S2 stated that they found the two-part creation of a component or a complete component failure unnecessary complicated. Here, they stated that a combination of head and tail into a single figure would be a better solution.

NF2, stating that *A user should easily be able to separate notations from each other when notation is presented graphicly* turned out to be an issue for the participants during the sessions. Participants in S2 stated they had trouble separating failure- and mitigation notation from mitigation- and failure interactions when drawn in the editor window. When asked, they had no improvement suggestions. The same issue was observed during two other sessions, but here the participants utilized the Comment notation as a means to clarify the the common diagram understanding and separating notation though textual statements. During the RP, the participants in S1 and S3 did not state this observed situation as an issue.

When asked, all participants stated they found it easy to locate specific notation in the editor palette. (NF3) As a mean to further visualize the different notation in the palette, S1 and S3 suggested adding small icons next to the figure instances in the palette.

Regarding repositioning and resizing of figures in the editor window (NF4), all participants commented they found this easy. As an opposition, all the sessions presented issues when scenarios called for editing execution behaviour into a part component failure and a component into a complete component failure. None of the participants in any of the three sessions attempted to edit existing notation in the editor window by stacking notation. Instead, they either created a new notation in the window and continued manually moving notation before removing the now unused figure or refrained from editing the figures at all. These are serious issues that affect the activity flow during the brainstorming and should be a main priority during further development.

Several syntactical errors where performed during the sessions. However, this is outside the scope of the student experiment and is therefore accepted here. With all five non-functional requirements mostly met, it is concluded that this approximation to a CHASSIS tool does work in a brainstorming setting.

9.6 Threats to validity

This section covers validity threats that may occur during research. [75] describes different types of validity which are covered in the remainder of this section.

Construct validity may be threatened if measures are not sufficiently defined for the concepts subjected to the study. During the collection of background knowledge the construct validity may be threatened if the presentation of this data makes use of unclear and insufficiently understood vocabulary. In this report, the threat to the collected data was handled by applying only qualified sources, submitted papers, when acquiring the information. The conducted experiment could also become a victim for threats to construct should the language used during the test be unclear. At the end of the experiment, participants where subjects to retrospect probing, aiming at resolving any unclarities. In addition, the participants where opted to ask questions during the experiments if anything was unclear.

Internal validity are subject to threats that may arise when conclusions form the data are drawn. During the student experiments, the validity may be threatened if actions, comments and feedback made by participants are wrongly interpreted by the test leader. Here, the internal validity threats where handled by encouraging the participants to utilize the concurrent thinking aloud (CTA), opting the participants to think out loud while working with the scenarios. In addition, audio tapes of the sessions, screen shots and the test leaders notes where used as the data from the experiment where analysed.

External validity states whether or not the conducted study can be generalized. The external validity of the experiment could be exposed to threats. The issues with visually separating figures as well as the substitution of a component into a complete component failure, both stated feedbacks and observations during the student experiments, is a matter of affordance within each of the participants and is not related to their understanding of either SD or FSD, but the tool itself. Therefore there is a possibility that this affordance will also apply if an experiment using safety and security expert participants are conducted, thus supporting the idea a generalisation of experiment findings.

Reliability concerns whether if the same data collection may be performed and achieve the same result. As a mean to avoid threats to reliability, a test plan was prepared in advance of the student experiments, thus facilitating a well structured procedure for the execution of the usability testing.

Chapter 10

Conclusions and further work

This report has presented two approximations to a CHASSIS tool allowing for the generation of Failure Sequence Diagrams, see Chapters 5-9. As there does not exist any previous attempts at approaching a tool for CHASSIS, the first research question aimed at discovering if there existed several options to a tool approach. This question was answered during a meeting with safety experts at Avinor. Here, two approaches to a tool for CHASSIS was suggested: a low-level, nearly syntactic free brainstorming tool and a formal and syntactically strict tool utilizing the constrictions of the UML Sequence Diagram. These two proposals lead to a third option: a two-fold tool containing both a high- and a low-level tool option.

Of the three approximations to the CHASSIS tool, the two first are further investigated in this report. The strict high-level approximation was first attempted to be implemented, but unforseen and unexpected instabilities with the program core of Papyrus, the chosen development platform, lead to a change of direction. Instead, the Graphical Modeling Framework was used to implement a low-level brainstorming approximation to the tool.

As a means to answer the second research question, asking if the approximation would work if placed in a realistic brainstorming setting, a student experiment was conducted. Despite some erroneous application of the Failure Sequence Diagram notation and a slight affordance issue with some of the notation, the experiment was considered a success and concluded that the tool did indeed work in a realistic setting.

Computer tools may aid users in a wide spectrum of tasks. As this report only present a first time approximation to a tool for CHASSIS, the list below suggest possible areas that could be a part of further work with a tool for CHASSIS:

- Implementing the first high-level approximation to the CHASSIS tool.
- Run experiments with the first tool approximation.
- Correct issues that where apparent during the student experiment with the second tool approach.
- Investigating the third approximation alternative by attempting to combine the two approximations presented in this report into a single tool.
- Conducting experiments with safety experts form the industry.

Bibliography

[1] Böhm, P and Gruber, T. (2010), *A Novel HAZOP Study Approach in the RAMS Analysis of a Therapeutic Robot for Disabled Children*, in Schoitsch, E., *Computer Safety, Reliability, and Security*, Springer Berlin Heidelberg, 15-27

[2] Opdahl, A. L. and Raspotnig, C. (2012), *Improving Security and Safety Modelling with Failure Sequence Diagrams.*, IJSSE 3 (1) , 20-36.

[3] Raspotnig, C. (2014), *Requirements for safe and secure information systems*, philosophiae doctor (ph.d), University of Bergen, Norway

[4] Raspotnig, C., Karpati, P., and Katta, V. (2012), *A Combined Process for Elicitation and Analysis of Safety and Security Requirements*, in Bider, I., Halpin, T., Krogstie, J. Nurcan, S., Proper, E., Schmidt, R., Soffer, P. and Wrycza, S., *Enterprise, Business-Process and Information Systems Modeling*, Volume 113, Springer Berlin Heidelberg, 347-361

[5] Raspotnig, C., Karpati, P. and Opdahl, A. (2013), *An Evaluation if CHASSIS with Two Air Traffic Management Suppliers*, Manuscript to be submitted

[6] Raspotnig, C. and Opdahl, A. (2012), *Supporting Faliure Mode and Effect Analysis: A Case Study with Faliure Sequence Diagrams*, in Regnell, B. and Damian, D., *Requirements Engineering: Foundation for Software Quality*, Springer Berlin Heidelberg, 117-131

[7] Stahl, T. and Völter, M. (2006) *Model-Driven Software Development - Technology, Engineering, Management*, John Wiley & Sons, Ltd., 85-108, 55-70, 223-238

[8] Meland, P. H., Spampinato, D. G., Hagen, E., Baadshaug, E. T., Krister, K. M., Velle, K. S. *SeaMonster: Providing tool support for security modeling*, ResearchGate

[9] http://www.eclipse.org/papyrus/, downloaded 02.04.2014

[10] http://www.ehow.com/list_7202158_advantages-using-software.html, downloaded 28.03.2014

[11] http://www.oxforddictionaries.com/definition/english/safety?q=safety, downloaded 25.03.2014

[12] http://www.oxforddictionaries.com/definition/english/security?q=security, downloaded 25.03.2014

[13] http://projects.eclipse.org/projects/modeling.mdt.papyrus, downloaded 02.04.2014

[14] https://www.microsoft.com/security/sdl/adopt/threatmodeling.aspx, downloaded 23.04.2014

[15] 10http://www.uml-diagrams.org/profile-diagrams.html, downloaded 24.04.2014

[16] http://www.uml-diagrams.org/uml-25-diagrams.html#structure-diagram, downloaded 29.04.2014

[17] http://wiki.eclipse.org/Papyrus, downloaded 01.05.2014

[18] http://sourceforge.net/apps/mediawiki/seamonster/index.php?title=Main_Page, downloaded 05.05.2014

[19] http://www.sintef.no/home/About-us/, downloaded 05.05.2014

[20] http://en.wikipedia.org/wiki/Eclipse_(software), downloaded 05.05.2014

[21] http://en.wikipedia.org/wiki/Compiler, downloaded 05.05.2014

[22] http://en.wikipedia.org/wiki/Interpreter_(computing), downloaded 05.05.2014

[23] http://en.wikipedia.org/wiki/Workspace, downloaded 05.05.2014

[24] http://en.wikipedia.org/wiki/Eclipse_(software)#Modeling_platform, downloaded 05.05.2014

[25] http://en.wikipedia.org/wiki/Papyrus_(software), downloaded 05.05.2014

[26] http://www.sparxsystems.com/resources/uml_datamodel.html, downloaded 06.05.2014

[27] http://www.dpi.ufv.br/projetos/geoprofile/tutoriais/Visual_Paradigm_for_UML_Tutorial_english.pdf, downloaded 06.05.2014

[28] http://msdn.microsoft.com/en-us/library/dd465143.aspx, downloaded 06.05.2014

[29] http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html, downloaded 07.05.2014

[30] http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/, downloaded 09.05.2014

[31] http://www.eclipse.org/papyrus/usersTutorials/resources/PapyrusUserGuideSeries_AboutUMLProfile_v1.0.0_d20120606.pdf, downloaded 10.05.2014

[32] http://www.upedu.org/references/bestprac/im_bp1.htm, downloaded 10.05.2014

[33] Guideline for applying CHASSIS, draft. Made avaliable by Raspotnig, C. at request.

[34] http://en.wikipedia.org/wiki/Requirement, downloaded 10.05.2014

[35] http://www.uml-diagrams.org/use-case-diagrams.html, downloaded 10.05.2014

[36] http://www.uml-diagrams.org/sequence-diagrams.html, downloaded 10.05.2014

[37] http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf, downloaded 12.05.2014

[38] http://www.oxforddictionaries.com/definition/english/problem?q=problem, downloaded 12.05.2014

[39] http://www.techrepublic.com/blog/10-things/10-techniques-for-gathering-requirements/, downloaded 12.05.2014

[40] http://searchsoftwarequality.techtarget.com/tutorial/Software-requirements-gathering-techniques, downloaded 12.05.2014

[41] http://www.evalperiod.com/services/10-strategies-for-software-requirements-gathering/, downloaded 12.05.2014

[42] http://www.cse.msu.edu/c̃se870/Lectures/Notes/02b-RequirementsElicitation-bhc-notes.pdf, downloaded 12.05.2014

[43] S. Kriaa, C. Raspotnig, M. Bouissou, L. Pietre-Cambacedes, P. Karpati, Y. Halgand, V. Katta (2013),*Comparing Two Approaches to Safety and Security Modeling: BDMP Technique and CHASSIS method* in *Proc. of the 37th Enlarged Halden Programme Group (EHPG) meeting*

[44] Raspotnig, C., Katta, V., Karpati, P. and Opdahl, A. (2013), *Enhancing CHASSIS: A method for Combined safety and Security Assessments* in *2013 Eighth International Conference on Availability, Reliability and Security (ARES)*, IEEE, 766-773

[45] http://en.wikipedia.org/wiki/Software_development_methodology, downloaded 13.05.2014

[46] http://en.wikipedia.org/wiki/Block_diagram, downloaded 14.05.2014

[47] http://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis#Functional_analysis, downloaded 14.05.2014

[48] http://www.hq.nasa.gov/office/codeq/risk/docs/ftacourse.pdf, downloaded 14.05.2014

[49] http://www.tdi.texas.gov/pubs/videoresource/stpfaulttree.pdf, downloaded 14.05.2014

[50] http://www.usability.gov/how-to-and-tools/methods/planning-usability-testing.html, downloaded 19.05.2014

[51] http://www.usability.gov/how-to-and-tools/methods/usability-testing.html, downloaded 19.05.2014

[52] http://www.usability.gov/how-to-and-tools/methods/running-usability-tests.html, downloaded 19.05.2014

[53] http://www.usability.gov/how-to-and-tools/methods/scenarios.html, downloaded 20.05.2014

[54] http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html, downloaded 20.05.2014

[55] Wilkinson, P. J and Kelly, T. P. (1998), *FUNCTIONAL HAZARD ANALYSIS FOR HIGHLY INTEGRATED AEROSPACE SYSTEMS*

[56] Pavlidis, M., Islam, S. and Mouratidis, H. (2012), *A CASE Tool to Support Automated Modelling and Analysis of Security Requirements, Based on Secure Tropos*, in *IS Olympics: Information Systems in a Diverse World*, Springer Berlin Heidelberg, 95-109

[57] Kiyavitskaya, N., and Zannone, N. (2008), *Requirements model generation to support requirements elicitation: the Secure Tropos experience*, in *Automated Software Engineering*, Springer US, 149-173

[58] Fabian, B., Gürses, S., Heisel, M., Santen, T. and Schmidt, H. (2008),*A comparison of security requirements engineering methods*, in *Requirements Engineering*, Springer-Verlag volume 15 - issure 1, 7-40

[59] Lund, M. S., Solhaug, B. and Stølen, Ketil, (2011) *Model-Driven Risk Analysis- The CORAS Tool*, Springer Berlin Heidelberg, 5-8

[60] Eames, D.,P. and Moffett, J., (1999) *The Integration of Safety and Security Requirements*, in *Computer Safety, Reliability and Security*, Springer Berlin Heidelberg, 468-480

[61] Srivatanakul, T., Clark, J., A., and Polack, F. (2004), *Effective Security Requirements Analysis: HAZOP and Use Cases*, in *Information Security*, Springer Berlin Heidelberg, 416-427

[62] http://www.utdallas.edu/s̃upakkul/tools/RE-Tools/creating-kaos.html, downloaded 22.05.2014

[63] Piètre-Cambacédès, L. and Bouissou, M. (2010), *Attack and Defense Modeling with BDMP* in *Computer Network Security - 5th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2010, St. Petersburg, Russia, September 8-10, 2010. Proceedings*, Springer Berlin Heidelberg, 86-101

[64] http://www.usability.gov/how-to-and-tools /methods/requirements.html, downloaded 22.05.2014

[65] http://sdedit.sourceforge.net/index.html, downloaded 26.05.2014

[66] http://www.all4tec.net/index.php/en/model -based-safety-analysis, downloaded 26.05.2014

[67] http://www.uml-diagrams.org/constraint.html, downloaded 26.05.2014

[68] http://www.visualstudio.com/en-us/products/how-to-buy-vs, downloaded 27.05.2014

[69] https://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1#Tutorial, downloaded 27.05.2014

[70] https://wiki.eclipse.org/EMF, downloaded 27.05.2014

[71] https://wiki.eclipse.org/GEF, downlaoded, 27.05.2014

[72] Marshall, T. (2008), *Brainstorming* in *Design Dictionary - Perspectives on Design Terminology*, Birkhäuser Basel, 49

[73] http://en.wikipedia.org/wiki/Eclipse_Modeling_Framework, downloaded 28.05.2014

[74] http://www.usabilityfirst.com/glossary/affordance/, downloaded 27.05.2014

[75] Wohlin, C., Runeson, P., Höst, M., Ohlsson, C., M., Regnell, B. and Wesslén, A. (2012), *Experimentation in Software Engineering*, Springer Berlin Heidelberg, 104-110

[76] http://en.wikipedia.org/wiki/Iterative_and_incremental_development, downloaded 29.05.2014

[77] http://www.edrawsoft.com/fault-tree-diagram-software.php, downloaded 30.05.2014

[78] http://what-when-how.com/information-science-and-technology/integrating-security-in-the-development-process-with-uml/, downloaded 30.05.2014

Appendix A

High level specification of CHASSIS tool

High level specification of CHASSIS tool

High-level requirements

1. The tool shall implement FSD and MUSD of CHASSIS
2. The tool should integrate with other development, safety or security analysis tools by using XML
3. It should be possible to extend the tool to connect to other diagrams, especially MUC diagrams
4. The system shall be user-friendly

Functional requirements

1. The tool shall implement all the rules and notation from UML sequence diagrams
2. The tool shall support the FSD
    - a. The tool shall implement all the notation of FSD
        - a.i. Failure and hazard of a component shall be represented by a red dashed circle on a lifeline of a component
        - a.ii. Failure of interaction shall either be:
            - a.ii.1. Red colored variable in message arguments
            - a.ii.2. Red colored arrow and text
        - a.iii. Error propagation through interaction shall be red arrows or red variable (on an arrow) that are connected to each other
        - a.iv. A mitigation against a failure shall be represented by a green dotted circle
    - b. The tool shall allow decomposition of FSD
3. The tool shall support the MUSD
    - a. The tool shall implement all the notation of MUSD
        - a.i. Vulnerability and threat of a component shall be represented by a red dashed circle on a lifeline of a component
        - a.ii. Vulnerability of interaction shall either be:
            - a.ii.1. Red colored variable in message arguments
            - a.ii.2. Red colored arrow and text
        - a.iii. Attacker sequence shall be red arrows or red variable (on an arrow) that are connected to each other
        - a.iv. A mitigation against a vulnerability shall be represented by a green dotted circle
    - b. The tool shall allow decomposition of MUSD

GUI requirements

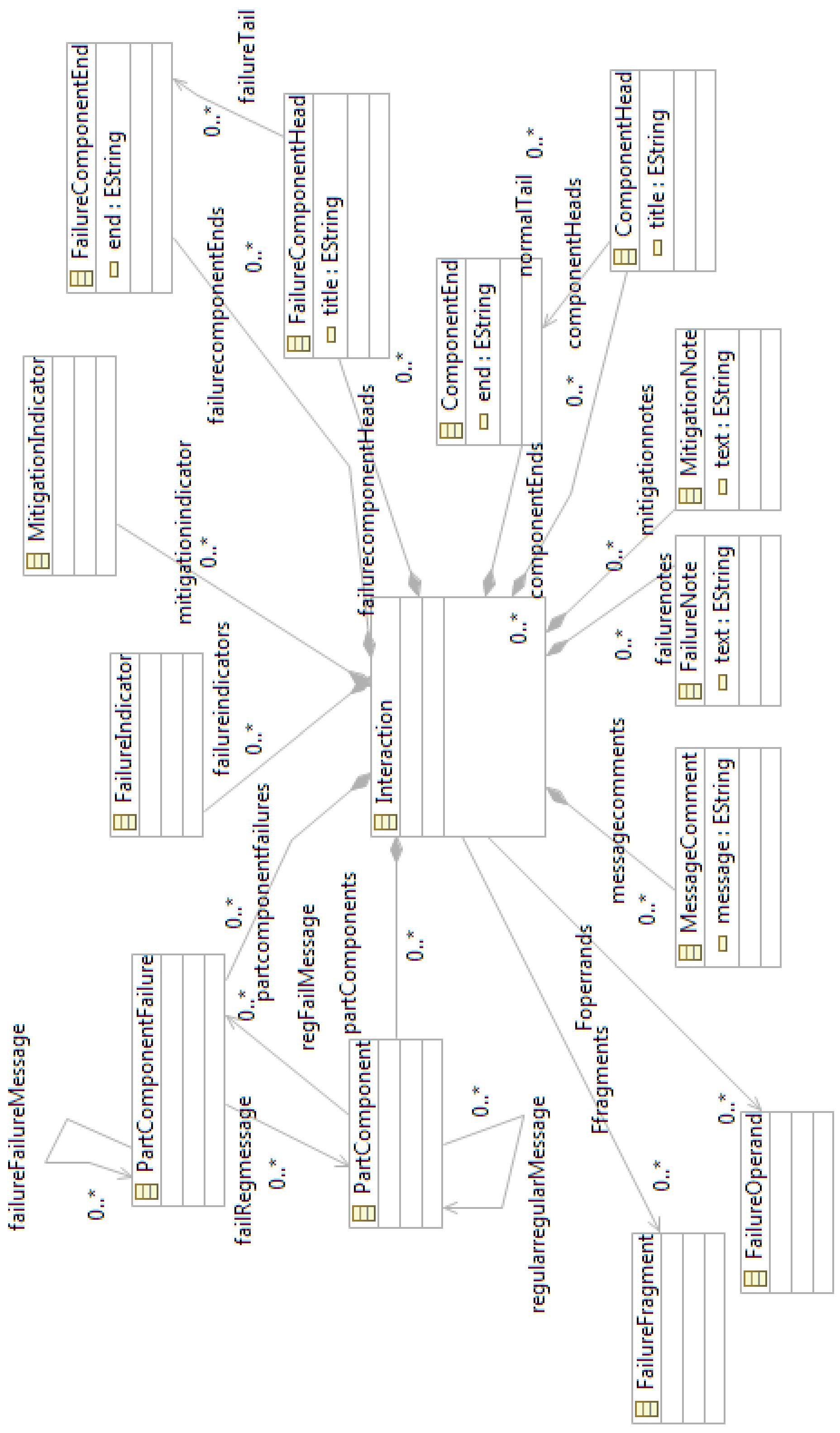
1. The tool shall support additional GUI features
    a. Save and save as
    b. Print/documentation – jpeg, xml-based reports
    c. ￼Search - data types
2. The user should be constrained from specifying elements that are syntactically wrong, e.g., failures in a MUSD or vulnerabilities in FSD

Usability requirements

1. The tool shall be used as a discussion medium in hazard, threat, failure and vulnerability analysis meetings
    a. The tool shall be able to display the FSD/MUSD with failures/vulnerabilities and mitigations to be visible to all stakeholders in a brainstorming meeting
2. The tool shall be used to document the hazard, threat, failure and vulnerability analysis
3. The tool should allow easy placement and changes to symbols in the FSD/MUSD
4. The tool should allow to hide elements (e.g., notes) in the FSD/MUSD

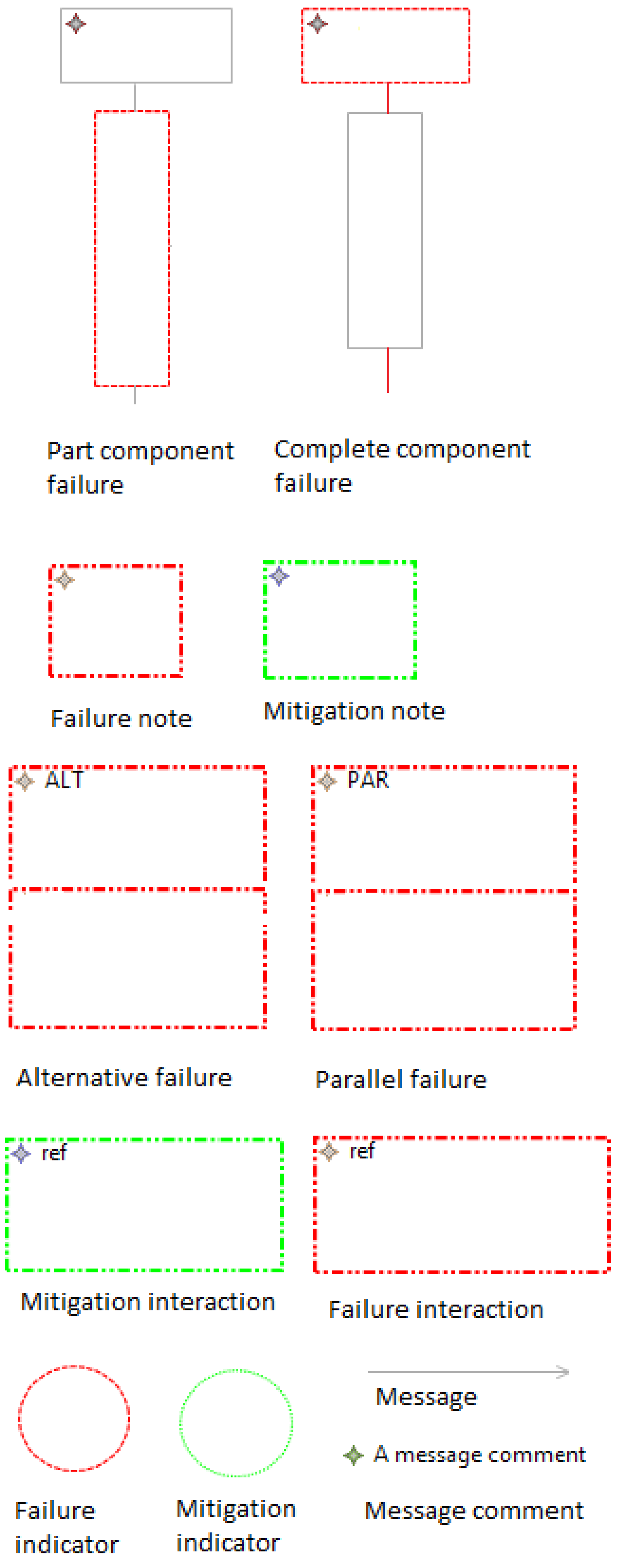
Appendix B

Domain model - GMF tool


FailureComponentEnd
end : EString
failureTail
0..*
FailureComponentHead
title : EString
failurecomponentEnds
0..*
ComponentHead
title : EString
normalTail
0..*
componentHeads
ComponentEnd
end : EString
0..*
MitigationIndicator
mitigationindicator
0..*
failurecomponentHeads
componentEnds
MitigationNote
text : EString
mitigationnotes
0..*
FailureIndicator
failureindicators
0..*
Interaction
0..*
failurenotes
FailureNote
text : EString
0..*
messagecomments
MessageComment
message : EString
0..*
partcomponentfailures
0..*
regFailMessage
partComponents
0..*
Foperrands
Ffragments
PartComponentFailure
failureFailureMessage
0..*
failRegmessage
0..*
PartComponent
0..*
regularregularMessage
FailureOperand
0..*
FailureFragment
0..*

Appendix C

Notation for CHASSIS tool - GMF


Part component failure
Complete component failure
Failure note
Mitigation note
ALT
PAR
Alternative failure
Parallel failure
ref
ref
Mitigation interaction
Failure interaction
Message
A message comment
Failure indicator
Mitigation indicator
Message comment

Appendix D

Proposed case for usability testing

Case for brukertesting av FSD verktøy

De skal modellere en server som distribuerer data gjennom en router til to forskjellige klienter. Routeren kommuniserer primært via kable med klientene, men kan svitsje til trådløs løsningen om det oppstår en feil med rutingen via kablene.

I første omgang skal de modellere server, router og to klienter med vanlige sekvensdiagrammer.

Så skal de med dette sekvensdiagrammet modellere at router feiler å sende meldinger til klientene. De markerer feil på routeren og at meldinger forsvinner.

Videre skal de dekomponere router komponenten. Her skal de modellere at routeren består av en mottaks-komponent, en pakkehåndterings-komponent og en sende-komponent. De skal vise hvordan feilen skjer ved at prosessen for å sende meldinger ut henger seg i sende-komponenten. De skal kunne relatere denne feilen (en under-feil) til feilen på router i det overordnede sekvensdiagrammet. De skal også bruke ”parallel failures” her, ved at to melding skal sendes til hver sin klient, og at begge feiler å bli sendt pga. feil i sende-komponenten.

Så skal de innføre en watch-dog på pakkehåndterings-komponenten, som sender ut heartbeats til mottaks- og sende-komponenten. Når denne oppdager at sende-komponenten har hengt seg, sender den en mitigerende melding til sende-komponenten for å få den til å resette seg.

Videre skal de modellere at pakkehåndterings-komponenten svitsjer til trådløs forbindelse og re-sender de siste meldingene (de er blitt bufret). Disse meldingen skal vises som ”message found”.

Til slutt skal de bruke ”failure interaction” for feilsekvensen med prosessen som henger seg på sende-komponenten og som fører til ”lost messages”. De skal videre bruke ”mitigation interaction” utenpå ”failure interaction” for watch-dog, heartbeat og ”message found”.

Appendix E

Case used during artefact analysis

Case for brukertesting-session

Firmaet Duck AS har all sin firmatada lagret på en server. Denne kan distribuere data gjennom en router til to forskjellige klienter; de to datamaskinene firmaet har i sine lokaler. Routeren kommuniserer primært via kabel med klientene, men kan svitsje til trådløs løsningen om det oppstår en feil med rutingen via kablene.

Duck AS har for tiden hatt problemer med at at routeren feiler å sende meldinger til klientene. For å prøve å finne ut av feilen velger de å ta en nærmere titt på hvordan router-komponenten fungerer. Ruteren består av en sende-, mottaks- og pakkehåndterings-komponent.

Etter en diskusjon blir de ansatte i Duck AS enig om å innføre en watch-dog på pakkehåndterings-komponenten, som sender ut heartbeats til mottaks- og sende-komponenten. Når denne oppdager at sende-komponenten har hengt seg, sender den en mitigerende melding til sende-komponenten for å få den til å resette seg. Nå vil pakkehåndterings-komponenten svitsje til trådløs forbindelse og re-sender de siste meldingene dersom noe skjer galt (de er blitt bufret).