MÅLARDALEN UNIVERSITY
SCHOOL OF INNOVATION, DESIGN AND ENGINEERING
VÄSTERÅS, SWEDEN

Advanced Project in Computer Science – DVA443 (30 ECTS)

Model-based approach for automatic generation of IEC-61025 standard compliant fault trees

Enrique Zornoza Moreno
ezo17001@student.mdh.se

Examiner: Barbara Gallina
Mälardalen University, Västerås, Sweden

Examiner: Radu Dobrin
Mälardalen University, Västerås, Sweden

Supervisor: Zulqarnain Haider
Mälardalen University, Västerås, Sweden

August 2018
Abstract

Reliability and safety of complex software-intensive systems are proved to be a crucial matter since most of these systems fulfil tasks, where a failure could lead to catastrophic consequences. For example, in space systems such as satellites, a failure could result in the loss of the satellite. Therefore, a certain level of reliability and safety must be assured for such systems to trust the services they provide. Standards set this level and put requirements for the analysis and assurance of these properties using documented evidence. In particular, European Cooperation for Space Standardization (ECSS) standards for space systems require Fault Tree Analysis (FTA) for identifying the causes of system failure and consequently safety hazards, as well as fault trees as evidence for the assurance of reliability and safety.

In this thesis, we present a tool supported model-based approach to generate fault tree automatically from an existing system modelling and analysis toolset. CHESS is a system and dependability modelling toolset and integrates Concerto-FLA to enable the support of failure logic analysis. We proposed a model-based transformation from Concerto-FLA to fault tree model and implemented it as an Eclipse plugin in CHESS toolset. A case study is performed in the aerospace domain; more specifically we modelled Attitude Control System (ACS) and automatically generated ECSS-compliant fault trees.
Table of Contents

1. Introduction .......................................................................................................................... 8
   1.1. Motivation ......................................................................................................................... 8
   1.2. Context ............................................................................................................................. 8
   1.3. Contribution .................................................................................................................... 8
   1.4. Document structure ........................................................................................................ 9

2. Background .......................................................................................................................... 10
   2.1. Complex Software Intensive Systems ............................................................................. 10
   2.2. Model-Driven Engineering ............................................................................................. 11
   2.2.1. Model-driven architecture (MDA) .............................................................................. 11
   2.2.2. Model-driven development (MDD) ............................................................................ 12
   2.2.3. MDE principles .......................................................................................................... 13
   2.2.4. Model transformations types ...................................................................................... 14
   2.2.5. Epsilon Transformation Language (ETL) .................................................................. 14
   2.3. Component-based software engineering ......................................................................... 15
   2.4. CHESS toolset ................................................................................................................ 16
   2.5. Dependability modelling and analysis approaches .......................................................... 17
   2.5.1. Dependability concepts ............................................................................................. 17
   2.5.2. Dependability analysis approaches ............................................................................ 18
   2.5.3. Concerto-FLA ............................................................................................................ 19
   2.6. Fault Tree Analysis ......................................................................................................... 20
   2.6.1. Graphical Elements ..................................................................................................... 21
   2.6.2. Fault tree generation process ....................................................................................... 22
   2.6.3. Cut set ........................................................................................................................ 22
   2.6.4. Quantitative vs qualitative FTA .................................................................................. 23

3. Related Work ....................................................................................................................... 24

4. Scientific Method ............................................................................................................... 26

5. Problem Formulation .......................................................................................................... 27
   5.1. Problem formulation ........................................................................................................ 27
   5.2. Problem analysis ............................................................................................................. 27

6. Automatic fault tree generation from system models ......................................................... 28
   6.1. Which existing failure analysis technique can be exploited to enable automatic generation of fault tree using model-based approaches according to the current state of the art? 28
   6.2. Which will be the structure of the meta-models required for automatic generation of fault trees? ......................................................................................................................... 29
   6.2.1. Source meta-model: FLAMM ..................................................................................... 29
   6.2.2. Target meta-model: EMFTA ...................................................................................... 30
   6.3. How to implement the automatic generation of fault tree in CHESS framework? ........ 32
   6.4. Proposed solution: Automatic FT generation ................................................................. 33
   6.4.1. M2M transformation algorithm .................................................................................. 33
   6.4.2. Eclipse plugin integrating the transformation ............................................................. 42
7. Case Study ........................................................................................................ 47
   7.1. Attitude Control System (ACS) ...................................................................... 47
   7.2. ECSS-compliant FT ....................................................................................... 48
   7.3. Failure logic analysis of ACS and fault tree generation ................................... 49
   7.4. Discussions .................................................................................................... 58

8. Conclusions and future work ............................................................................. 59
   8.1. Summary ........................................................................................................ 59
   8.2. Limitations and future work ......................................................................... 59

References ........................................................................................................... 61
List of figures

Fig. 1: Relation between MDA, MDD and MDE [30] ................................................................. 11
Fig. 2: MDA portability [1] ........................................................................................................ 12
Fig. 3: MDA transformations [2] ................................................................................................ 12
Fig. 4: OMG layered MDA [32] .................................................................................................. 13
Fig. 5: ETL abstract syntax overview [39] .................................................................................. 15
Fig. 6: ETL concrete syntax example [39] ................................................................................... 15
Fig. 7: ETL pre and post block concrete syntax [39] .................................................................. 15
Fig. 8: Relation between component and interfaces [5] .............................................................. 16
Fig. 9: Error propagation in a system [7] ..................................................................................... 17
Fig. 10: Dependability and security attributes [7] ...................................................................... 18
Fig. 11: Concerto-FLA process [13] ........................................................................................... 20
Fig. 12: FPTC rules syntax [5] ................................................................................................... 20
Fig. 13: FT generation procedure based on [19] ....................................................................... 22
Fig. 14: Scientific method used in this thesis based on [48] ......................................................... 26
Fig. 15: FLA Meta-model [40] .................................................................................................... 29
Fig. 16: EMFTA meta-model [43] .............................................................................................. 30
Fig. 17: EMFTA tool example of FT [43] ................................................................................... 31
Fig. 18: EMFTA representation of event types [42] ................................................................... 31
Fig. 19: CHESS component implementation example [4] .......................................................... 32
Fig. 20: FLA rules syntax example [40] ..................................................................................... 33
Fig. 21: Flowchart of FLAMM to EMFTA transformation ............................................................ 35
Fig. 22: ETL code for FTAModels generation ............................................................................. 40
Fig. 23: ETL code for top-event creation ................................................................................... 41
Fig. 24: Population algorithm ETL implementation for case 1 .................................................. 41
Fig. 25: Plugin appearance .......................................................................................................... 42
Fig. 26: Selection of source model implementation ..................................................................... 42
Fig. 27: UI for source file selection ............................................................................................ 43
Fig. 28: Creation of ETLModule and file parsing ...................................................................... 43
Fig. 29: EMFModels configuration Java method ....................................................................... 44
Fig. 30: Module configuration and execution ............................................................................. 44
Fig. 31: Clear of module and models .......................................................................................... 44
Fig. 32: Introduced changes in visualizer class (1) .................................................................... 45
Fig. 33: Introduced changes in visualizer class (2) .................................................................... 46
Fig. 34: Plugin visualizer implementation .................................................................................... 46
Fig. 35: ACS software functions [47] ......................................................................................... 48
Fig. 36: Structure of ECSS standards [52] ................................................................................ 49
Fig. 37: Methodology followed in the case study ........................................................................ 50
Fig. 38: ACS data-flow ............................................................................................................... 51
Fig. 39: ACS system modeled in CHESS ML ........................................................................... 51
Fig. 40: Scenario 1 failure behaviour for FeedforwController ................................................... 52
Fig. 41: Scenario 1 modelled system annotated with Concerto-FLA results ................................ 52
Fig. 42: Concerto-FLA results in CHESS .................................................................................. 52
Fig. 43: Scenario 1 result top section of FTAModel ................................................................... 53
Fig. 44: Scenario 1 result bottom section of FTAModel ............................................................. 54
Fig. 46: Rule for PD Controller .................................................................................................. 55
Fig. 47: Rule for FeedforwardController ................................................................................. 55
Fig. 48: Scenario 2 modelled system annotated with Concerto-FLA results .............................. 55
Fig. 49: Scenario 2 results top section FTAModel ...................................................................... 56
Fig. 49: Scenario 2 results bottom section FTAModel ............................................................... 57
List of tables

Table 1: FT standard events. ................................................................. 21
Table 2: FT standard static gates. ........................................................ 21
Table 3: FT standard dynamic gates. .................................................... 21
Table 4: Comparison of methods used to derive failure information. ........ 28
Table 5: Mapping between the source and target meta-model’s elements. ................................. 34
Table 6: Algorithm 1 for deriving FTModels........................................... 35
Table 7: Algorithm 2 for port transformation and FT population. ............. 36
Table 8: Algorithm 3 for FTPopulation. Case 1 ........................................ 37
Table 9: Algorithm 3 for FTPopulation. Case 2 ........................................ 38
Table 10: Algorithm 3 for FTPopulation. Case 3 ....................................... 39
## Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL</td>
<td>Architecture Analysis and Definition Language</td>
</tr>
<tr>
<td>ACS</td>
<td>Attitude Control System</td>
</tr>
<tr>
<td>CBSE</td>
<td>Component-Based Software Engineering</td>
</tr>
<tr>
<td>CHESS</td>
<td>Composition with Guarantees for High-integrity Embedded Software Components Assembly</td>
</tr>
<tr>
<td>CHESS ML</td>
<td>CHESS Modelling Language</td>
</tr>
<tr>
<td>DFT</td>
<td>Dynamic Fault Tree</td>
</tr>
<tr>
<td>ECSS</td>
<td>European Cooperation for Space Standardization</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>EMFTA</td>
<td>EMF-based Fault Tree Analysis</td>
</tr>
<tr>
<td>Epsilon</td>
<td>Extensible Platform of Integrated Languages for mOdel maNagement</td>
</tr>
<tr>
<td>ETLM</td>
<td>Epsilon Transformation Language</td>
</tr>
<tr>
<td>HAZOP</td>
<td>Hazard and Operability</td>
</tr>
<tr>
<td>FLA</td>
<td>Failure Logic Analysis</td>
</tr>
<tr>
<td>FLAMM</td>
<td>Failure Logic Analysis Meta-Model</td>
</tr>
<tr>
<td>FMEA</td>
<td>Failure Mode and Effect Analysis</td>
</tr>
<tr>
<td>FMECA</td>
<td>Failure Mode and Effect and Criticality Analysis</td>
</tr>
<tr>
<td>FPTC</td>
<td>Fault Propagation and Transformation Calculus</td>
</tr>
<tr>
<td>FT</td>
<td>Fault Tree</td>
</tr>
<tr>
<td>FTA</td>
<td>Fault Tree Analysis</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>M2M</td>
<td>Model-to-Model</td>
</tr>
<tr>
<td>MARTE</td>
<td>Modelling and Analysis of Real Time and Embedded systems</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Family</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>SASM</td>
<td>Sun Acquisition and Survival Mode</td>
</tr>
<tr>
<td>SFT</td>
<td>Static Fault Tree</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
</tbody>
</table>
1. Introduction

In this chapter, the content provided by this thesis is introduced. Section 1.1 discusses the primary motivation for the thesis. Section 1.2 presents the context in which this thesis is developed, and then Section 1.3 describes which are the main contributions provided by our work. Lastly, Section 1.4 shows the overall organisation that this thesis follows.

1.1. Motivation

High-integrity, cyber-physical embedded systems are becoming complex due to the growing demand for new functionalities and recent digitalisation trend. This digitalisation is targeting diverse domains, e.g., automotive industry, space systems, railway systems, avionics and process automation industry. Additionally, the development of such systems is regulated by the authorities, and compliance with the standards issued by them is an obligation; this also contributes to the complexity. Standards require a system to be analysed for different properties such as reliability, safety and security, to ensure the dependability of the system. Analysing and ensuring such properties requires an immense amount of effort and a deep understanding of the system. Traditionally, these tasks have been performed manually, which makes them time-consuming and error-prone. According to [62], such analyses “should also be done rapidly enough to keep consistent with design and […] integrated into the design process”. To address this, they state that a methodology for deriving this information automatically in a unified model-based toolset is needed. Thereby, both design and safety data would be contained in the same model which, according to [62], is not currently supported by similar solutions.

In particular, the European Cooperation for Space Standardization (ECSS) standards require Fault Tree Analysis (FTA) for identifying the causes of system failure, as well as fault trees as evidence for the assurance of reliability and safety. The automatic generation of such documents in a unified toolset could contribute to reducing the needed effort and cost, as well as aligning with the needs of the safety-critical systems development process.

1.2. Context

This thesis is developed under the context of the AMASS project [65] and the CHESS toolset [38]. The former aims to develop an open tool platform for the certification of Cyber-Physical Systems (CPS). The latter is one of the baseline tools used by the AMASS project, and aims for addressing safety, reliability and extra-functional properties of the system, in addition to the functional aspects. To do so, this toolset, and its successor, CONCERTO [12], use Model Driven Engineering (MDE) as a solution to manage the complexity of these systems in an easy way for the user. CHESS Project has derived the CHESS toolset as a tool-supported methodology to support such modelling of systems. This toolset incorporates a set of Eclipse plugins to address the previously mentioned issues. One of such plugins is Concerto-FLA [64], which provides a dependability technique enabling failure behaviour and propagation analysis of a component-based system.

FTA is a deductive approach for analysing a system failure event by constructing a probable cause tree. Such fault trees could be constructed from the failure propagation paths produced by Concerto-FLA, and feasibility is demonstrated in [46].

1.3. Contribution

The main contribution this thesis provides is the development of a novel solution that generates automatically IEC-61025 standard compliant fault trees from model-based systems. To do so, the following elements have been derived:

- An analysis of current state-of-the-art solutions, in order to learn from their weakness and strengths. Additionally, a study regarding failure analysis techniques proposed by related studies, with a particular focus on Concerto-FLA, since the results generated from this analysis cornerstone for this thesis, as well as FTA. The work provided in [46] is used as the starting point of this thesis.

- The design and implementation of a novel algorithm that generates these fault trees from a modelled system. In addition, the automatization of this algorithm is developed in the form of an Eclipse plugin.

- The evaluation of the algorithm in the form of a case study, which presents an Attitude Controller System (ACS) as the system under study.

As an early development of this work, a paper [63] was submitted to the 3rd International Conference on System Reliability and Safety (ICSRS) 2018 and is in the progress of being accepted. Additionally, this work will be soon integrated into Polarsys Opencert tools platform [66].
1.4. Document structure

Chapter 2 presents the background knowledge needed both to develop the solution and for the reader to understand the work behind this thesis.

Chapter 3 presents how the problem undertaken with this work, has been accomplished by related studies and how our approach is going to differ from them.

Chapter 4 presents the scientific methodology followed to derive solutions for the proposed problem.

Chapter 5 presents the problem and a brief analysis to formulate research questions which are answered in this thesis.

Chapter 6 presents the proposed solution for the previous problem. Firstly, the formulated questions are answered, and then the approach is presented. The provided solution is based on designing and implementing an algorithm that performs a model-to-model transformation. Then, to automate this solution, the algorithm is implemented in the form of an Eclipse plugin.

Chapter 7 presents a case study of Attitude Control System (ACS) from the aerospace domain which is used to illustrate the proposed solution.

Chapter 8 presents the conclusions derived from this thesis as well as some limitations found. Then, some future lines of work related to this thesis are provided.
2. Background

In this chapter, the basic concepts needed to understand the main problem are explained in the following sections. Section 2.1 presents the concept of complex software-intensive systems, as well as how these systems must be compliant with standards. Since the approach is based on systems developed with Model-Driven Engineering (MDE), Section 2.2 explains the key concepts regarding MDE and the most common languages used to create these system models. Then, Section 2.3, presents another methodology to manage complexity called component-based software engineering and Section 2.4 is focused on presenting the AMASS project and the CHESS toolset and how they integrate both methodologies. Later, Section 2.5 explains the central concepts related to dependability and which are the different approaches made to perform dependability analysis in model-based systems. Finally, Section 2.6 presents the main concepts of one such analyses, i.e., Fault Tree Analysis (FTA). Since this work is built on top of several previous projects and concepts, some of the concepts explained during this Chapter 2 follow the same structure presented in [5] and in [64].

2.1. Complex Software Intensive Systems

Most of the services that are provided in our daily lives are performed by software systems, but such systems can be differentiated according to the service they provide and how critical is their interaction with the environment. These systems are referred to as software-intensive systems and are defined in [35] as "any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole". This definition was introduced in 2000, and since then the number of relations and interactions between these systems have grown, transforming them into complex software-intensive systems.

This complexity is increased when such systems are embedded systems, interacting with other systems and the real world through devices such as sensors and actuators. Such embedded systems can be considered as critical depending upon the services they provide. In [36], critical systems are characterized as systems "which must be highly reliable and retain this reliability as they evolve without incurring prohibitive costs" (the concept of reliability is discussed in Section 2.5.1). Their critical nature comes from the consequences of the malfunctioning of such systems. For example, a safety-critical system failing in providing the correct services can result in loss/injury to human life, while a failure of the mission-critical system could jeopardise the achievement of the objectives of system or project.

In the scope of this thesis, only the latter is considered, since the proposed case study is performed in an Attitude Control System (ACS). This system, defined in the aerospace domain, fits in this mission-critical systems group. The failure of this systems can result in catastrophic consequences for the mission and sometimes for the environment. Due to this, such systems must be developed with a certain level of quality and must comply to some level. This level is called standard, and the compliance with them has proven to be critical for their ability to deliver services.

As stated previously, there is a need to "set the bar" of safety in a proper level for safety-critical systems. If this level is set too high, no company could afford to follow the rules, and if the level were too low, quality would not be assured. Consequently, it is needed to establish a level where a system is said to be safe enough to operate in its intended context. The standard concept is presented as the solution to this issue. ISO (International Organization for Standardization) defines standards as "documents that provide requirements, specifications, guidelines or characteristics that can be used consistently to ensure that materials, products, processes, and services are fit for their purpose" [21].

A product is said to be compliant with a standard if it conforms to the specifications and requirements proposed by that standard. This concept is critical in the scope of this thesis since the proposed solution needs to generate documents compliant with the proposed standard. In this work, the case study presents ACS, which is the unit responsible for maintaining the orientation of a satellite once it is launched to space. Therefore, this system has mission-critical nature – thus, required to be engineered according to the ECSS (European Cooperation for Space Standardization) standards, and more specifically to ECSS-Q-ST-30C [54] and ECSS-Q-ST-40C [61] which deal with dependability and safety respectively. The main goal of this organisation is to improve standardisation within the European space sector.

Depending on the sector in which the project is developed, a different standard needs to be followed. Some of them are presented below:

- **Automotive industry**: ISO 26262 [49] is the international standard for functional safety of electrical and/or electronic systems in production automobiles defined by ISO in 2011. It is divided into ten parts and is based on the IEC 61508 [55] standard, which is the precursor for electrical/electronic systems standards.
• **Avionics industry:** DO-178C [56] is the primary document by which certification authorities approve software-based aerospace systems. It was introduced in 2012 replacing DO-178B [57], which was a de facto standard. This document introduces guidelines regarding the safety of software elements and how to determine this level by using Design Assurance Level (DAL).

• **Railway industry:** EN 50128 [58] is the European standard for railway control and protection systems, which international version is IEC 62279 [59]. It aims for being used in any area of the railway domain where there are safety implications.

• **Space industry:** ECSS-Q-ST-30C and ECSS-Q-ST-40C, deal with dependability and safety concepts at the system level.

As mentioned above, these systems are becoming each day more complex, which makes the development and certification of them a time-consuming task. Nevertheless, some methodologies to engineer these systems are appearing and establishing in the software industry. Two of the main methodologies are Model-driven engineering (MDE) and Component-based software engineering (CBSE), which are described in the following sections. Both appear as a solution to manage complexity and automatize some engineering tasks.

### 2.2. Model-Driven Engineering

One of the leading solutions proposed to reduce the complexity of the previously presented systems is Model-Driven Engineering (MDE). MDE is presented as a new paradigm focused on representing systems and their components as models, not only for software development but also for decision making and analysis. According to [29], “Model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively”.

Throughout history, several initiatives related to model-driven have been developed, such as Model-Driven Architecture (MDA) and Model-Driven Development (MDD). These two initiatives and MDE are related to each other since they inherit ideas and use similar concepts. To see the relation and how one initiative encompasses the others, Ameller in [30] presents the relation between MDA, MDD, and MDE in the form of a Venn diagram, presented in Fig. 1. Below, the evolution of this paradigm is presented through the different phases and approaches that have been developed (Section 2.2.1 and 2.2.2), as well as the fundamental concepts related to model-driven solutions (Section 2.2.3 and 2.2.4).

![Fig. 1: Relation between MDA, MDD and MDE [30].](image)

#### 2.2.1. Model-driven architecture (MDA)

Model Driven Architecture (MDA) is a standard proposed by the Object Management Group (OMG) [1] in 2000. OMG states that “the essential goal of MDA is to derive value from models and modeling that helps us deal with complexity and interdependence of complex systems” [1]. One of the key differences from this initiative to the rest is that the modelling language used to derive the models must follow the Meta Object Family (MOF) rules. MOF aims to be the primary standard for any modelled systems and is the main foundation of the layered architecture proposed by OMG. This layered architecture is later explained in detail in Section 2.2.3.

The main idea derived from MDA is that there is a clear separation from the specifications of how the system operates in general, to the specifications of a specific platform. To do so, three different models are presented in MDA, which represent the three steps followed when a system is developed. The first one is called Computational Independent Model (CIM). This model is closer to business concepts since it specifies the main goals of the developed systems as well as how it interacts with other systems. Once this is presented (usually with Use Case diagrams) a second model is developed called Platform Independent Model (PIM). In PIM, a more detailed
representation of system functionalities is defined but without determining a platform-dependent implementation. By using this PIM, and once the platform to be used has been agreed upon, a Platform Specific Model (PSM) can be defined, which represents a PIM adapted to the chosen platform. Therefore, MDA can produce PSM for a specific platform and technology from PIM. Fig. 2 shows the idea of how several PSM can be derived from one PIM.

Additionally, different transformations are defined to generate code and models. For example, in [2], seven different types of transformations are identified and illustrated in Fig. 3. Nevertheless, this thesis focuses on the first four, due to their relation to the concepts explained in the next sections. A brief description of these transformations is presented below.

1. **PIM to PIM**: This type of transformation is usually found during the development lifecycle when the models do not require any platform-specific model, which are related to refinement of these models until the platform is needed.

2. **PIM to PSM**: Once the PIM has been sufficiently refined, the transformation to a Platform Specific Model (PSM) is made. Depending on the target PSM, the transformation rules can change, but these rules can also be expressed in PSM.

3. **PSM to PSM**: This transformation is related to platform-dependent model refinement. As OMG explains, “this transformation is needed for component realization and deployment” [3].

4. **PSM to PIM**: The most common use of this transformation is for deriving an abstraction of an existing implementation. This is the most complex transformation to be automatized according to [3].

2.2.2. **Model-driven development (MDD)**

Later, in 2003, an article under the same name [27] is presented in IEEE Software, which defines MDD as: “simply the notion that we can construct a model of a system that we can then transform into the real thing.”
MDD is a development technique founded on MDA but with a critical difference: the models developed does not have to comply with any of the OMG standards. This implies that MDD benefits the development process since it adds more flexibility.

From MDA standard, MDE is derived as a development methodology which includes all modelling tasks and models needed to develop a complete software project. Some of the basic concepts presented by MDA, which are consequently used by MDE, are needed to understand this paradigm and are discussed in the following subsections.

2.2.3. MDE principles

System

“A system is defined as a collection of parts and relationships among these parts that may be organized to accomplish some purpose” [1]. Therefore, anything can be seen as a system, from hardware/software systems to business process and enterprises. The key advantage of system concept in MDE is ease of representing and understanding the complex real-world elements in comparison to the implementation-based techniques, such as software source code.

Model

In the context of MDA, a model is said to be “information selectively representing some aspect of a system based on a specific set of concerns” [1]. Thus, models are elements which fulfil a specific function inside a system. The combination of models, their relations and connections are what define a system.

For a model to be useful, it is necessary that the information is presented so that it can be interpreted by other parts of the development process (stakeholders). To do so, a language for defined models, relations and connections needs to be proposed. An extensive number of languages has been defined, under MDA initiative, to model systems. In this thesis SysML, MARTE and UML are discussed since our work is based on these languages. Several other known languages developed in MDE such as Matlab/Simulink, AADL and AltaRica based approaches are also mentioned in related work for the sake of completeness.

A model written in a specific modelling language is conformant to such modelling language. The structure of a modelling language is called meta-model and represents the base for all the models derived using this language.

Meta-Model

The collection of concepts, relations, and syntax used to define a model is known as a meta-model. Therefore, a model conforms to a meta-model.

Meta Meta-Model

According to Wendell et al. [32], “a meta-model is a model itself conforming to a meta-model, called the meta meta-model”. Therefore, if a meta-model conforms to itself, it is known as a meta meta-model. OMG introduced a layered architecture where systems, models, meta-models and meta meta-models are presented as layers, and each of them conforms to the above layer, except for the meta meta-model, which conforms to itself. For example, according to the MDA sub-section, MOF would always be in the M3 level of this architecture when MDA is followed. An overview of this architecture is shown in Fig. 4.

![Fig. 4: OMG layered MDA [32].](image-url)
2.2.4. Model transformations types

In the scope of MDE, a transformation is defined as a way of creating or updating models. In this context, a model is taken as an input to derive a new model or to refine the existing one. The primary goal of these transformations is to reduce the amount of work to system designers while they need to refactor or change their models. Depending on the type of inputs and outputs the transformation has, four types of transformations can be defined. Nevertheless, this thesis only discusses the three types that involve models, which are listed below along with few of the transformation languages used to implement them:

- **Model-to-model transformation (M2M):** According to Stephan and Stevenson [6], “M2M transformation refers to the process of modifying a model to create either a new model or update itself”. To follow this process, some rules have to be defined and followed to produce the targeted model. In [6], it is also discussed that these transformations go from PIM models to PIM models or either PSM, not in the code form. Some of the existing languages to perform M2M transformations are Atlas Transformation Language (ATL), Epsilon Transformation Languages (ETL) and Query View Transformation (QVT), which can be Operational Declarative (QVT-o). In this thesis, ETL is used to perform M2M and is discussed in detail in Section 2.2.5.

- **Model-to-text transformation (M2T):** This type of transformation is focused on receiving a model as an input and generating a textual artefact of this model as an output. This textual artefact is then represented as a stream of characters that have no explicit definition of the target meta-model. The main languages proposed to perform these are Jet, Acceleo, Xpand or Xtext.

- **Text-to-model transformation (T2M):** This is used to define a model from the grammar of a textual artefact. To perform this transformation, a parser of the input text needs to be developed in order to get the information needed from the text. Of the three presented transformation types, T2M is the most complex to accomplish according to [5]. The main problem is that the textual representation needs to be derived following a specific syntax definition language. Not too many techniques regarding this transformation are available.

2.2.5. Epsilon Transformation Language (ETL)

According to [39], Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) is “a platform for building consistent and interoperable task-specific languages for model management tasks such as model transformation, code generation, model comparison, merging, refactoring and validation”. This platform provides several languages, and for each language, Epsilon provides Eclipse tools and interpreters to execute and understand programs written in these languages. One of those languages is called Epsilon Transformation Language (ETL) and is used to develop model-to-model transformations (M2M).

Generally, there are three types of modelling languages, which are:

- **Declarative:** source and target meta-models are similar in their structure. Thus, the transformation consists of mapping the elements. The process that implies complex transformations does not fit with declarative types.

- **Imperative:** addresses a more extensive range of transformation scenarios but operating in a low level of abstraction.

- **Hybrid:** declarative rules for mapping and imperative features to handle complex scenarios. ETL belongs to the third type. This third group is derived from solving the issues that the previous languages cannot address.

The fundamental idea of model transformation is that an arbitrary number of input models can be converted into an arbitrary number of output models of different modelling languages. Most of the model-to-model transformation languages assume that there are only two types of models: source and target. Nevertheless, ETL adds other types, such as trace or configuration, to save/access models during transformations. Besides, [39] also defines a syntax for ETL, which can be presented in two different levels, which are explained below:

**Abstract syntax**

This syntax explains which elements are used in ETL and how they relate to each other. In [39], transformations are organized in modules, and each module can contain several transformations rules. For each rule, a source and target parameters are defined. Optionally, a rule can define guards, which can be derived as an EOL (Epsilon
Concrete syntax

A concrete syntax is provided to represent the previous abstract concepts and shall be used while programming. As in other programming languages, a set of keywords are used to code elements such as rules, transformations, and guards. An example of a piece of code, extracted from [39], is presented in Fig. 6.

As stated previously, ETL allows adding pre and post conditions to a transformation rule. The execution of these is either before or after the rule depending on the used condition. An example of how these statements would look like in the concrete syntax is presented in Fig. 7.

In addition to the usual transformations, ETL also provides another type called interactive. This allows the user to select some specific input manually in the source or target model for the transformation.

2.3. Component-based software engineering

Another proposed solution to solve the problem presented in Section 2.1 is component-based software engineering (CBSE). According to Verma [34], CBSE is defined as “an approach to developing software that
Model-based approach for automatic generation of IEC-61025 standard compliant fault trees

relies on the reuse of software”. This approach proposes that a system (or any application) can be developed by the use of a set of pre-built standardised software components, which are reusable. To develop the final system, the components are assembled instead of integrating different parts of conventional programming languages. This methodology allows the developer to upgrade or change the system with ease since they only need to change the component which needs to be modified. To understand better the CBSE paradigm, the key concepts related to this are explained below.

**Component**

According to Szyperski [34], “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by the third party”. Therefore, components need to be independent of each other so that they can communicate with other components through interfaces.

For a component to be suitable for this methodology, it needs to comply with some characteristics, which according to [5] are: independent, standardised, deployable, documented and composable. These characteristics are furthered explained in [5]. Once the component has been defined, CBSE presents component composition, which allows developers to combine two or more components. This combination creates a new component behaviour that can be suited to the needs of the developer.

**Interface**

Interfaces are presented as parts of the component which allows the communication with other components. Through them, components can send or receive the information needed to fulfil their tasks. Two types of interfaces are defined in CBSE:

- **Required interface**: services that are provided by the component to other components. It also represents the output of the component.
- **Provided interface**: services that are provided to the component by other components. It also represents the input of the component. An overview of the appearance of these two interfaces is shown in Fig. 8.

![Provided Interface](Component) ![Required Interface](Component)

Fig. 8: Relation between component and interfaces [5].

To summarise CBSE, four principles of this methodology are presented by Verma [34]:

1. Components are independent, so they do not interfere with each other.
2. Component implementation is hidden.
3. Communication is through well-defined interfaces.
4. Component platforms are shared and reduce development costs.

2.4. CHESS toolset

In previous sections, Model Driven Engineering (MDE) and Component-Based Software Engineering (CBSE) were presented as existing solutions to manage the increasing complexity of current software-intensive systems. The combination of these two approaches provides an increase of reusability and reduction of costs related to the development process. According to [37], current approaches that combine both methodologies fail to address extra-functional properties.

CHESS (Composition with Guarantees for High-integrity Embedded Software Components Assembly) project [38], appears as a solution that tries to accomplish previous limitations by providing a methodology to model and develop high integrity real-time component-based systems. Therefore, CHESS allows addressing safety, reliability and extra-functional properties of the system, in addition to the functional aspects of the system. In their own words: “CHESS project seeks to improve Model Driven Engineering practices and technologies to better address safety, reliability, performance, robustness and other extra-functional concerns while guaranteeing the correctness of component development and composition for embedded systems” [38].
CHESS also provides a framework or toolset, which implements the methods to assess above mentioned extra-functional properties of a system through several plugins integrated within the Eclipse IDE. Within this framework, CHESS provides a new modelling language called CHESS ML, which is defined as a “collection-extension of subsets of standard OMG languages (UML, MARTE, SysML)” [3]. As this language “imports” all the other three, these are grouped and treated them as a single language in Chapter 3.

Additionally, the AMASS project is also presented as a platform that aims to “create and consolidate the de-facto European-wide open tool platform, ecosystem, and self-sustainable community for assurance and certification of Cyber-Physical Systems (CPS) in the largest industrial vertical markets including automotive, railway, aerospace, space, and energy” [65]. This platform is developed on top of other tools such as Eclipse Process Framework (EPF) [67] or the own CHESS toolset. Therefore, the work developed in this thesis will ultimately be integrated into the AMASS project.

2.5. Dependability modelling and analysis approaches

In this section, we discuss existing approaches for dependability modelling and analysis as well as the main concepts on which it is based.

2.5.1. Dependability concepts

According to Avizienis et al. [7], dependability is defined as “the ability to deliver service that can justifiably be trusted” and is a non-functional property (NFP) of a system. In order to understand what this term means, there are some concepts that are needed. The main one is the system and is defined by Avizienis as “an entity that interacts with other entities” [7]. The frontier between these two entities is defined as system boundary. Avizienis also presents three components that define the proposed dependability tree: threats, attributes, and means. Threats are needed to understand what a fault and a failure are and how they can be defined to develop a fault tree. Attributes are also important in the scope of this section since the existing approaches presented in Section 2.5.2 are categorized according to dependability attributes. Finally, means are not discussed since they are out of the scope of this thesis.

Dependability threats

According to [7], three main threats to dependability are faults, errors, and failures. Faults are defined as “the adjudged or hypothesized cause of an error”, which can be internal or external to the system and also active (if they lead to an error) or dormant. Once the fault is active, an error occurs, which is defined as “the total state of the system that may lead to its subsequent service failure”. When the error reaches the system boundary and manifests as a deviation from the correct service, the failure takes place. Fig. 9 represents these concepts as well as how the error propagation affects the service status of a system.

Once the failure occurs, there can be several ways in which it manifests itself. These different ways are defined by [7] as failure modes, which could be of the following three types:
• **Value**, which is presented when a different value from the one expected is provided. It has two modes: value subtle, if the output deviates from the expected output value in an undetectable way, and value coarse if the output deviates from the expected output value in a detectable way.

• **Timing**, which is presented when the action takes place in a different moment from the intended one. It has two modes: early, if the output is provided before, and late if the output is provided after it is expected.

• **Provision**, which is presented when the action is performed in a different timing than the intended one. The difference with the timing mode is that the value is either provided or not. It has two modes, i.e., omission if no output is provided, and commission if the output is provided when it is not expected.

### Dependability attributes

Dependability is presented as an “umbrella term”, which encompasses the following key attributes:

- **Availability**: “readiness for correct service” [7].
- **Reliability**: “continuity of correct service” [7].
- **Safety**: “absence of catastrophic consequences on the user(s) and the environment” [7].
- **Integrity**: “absence of improper system alterations” [7].
- **Maintainability**: “ability to undergo modifications and repairs” [7].

In addition, Avizienis adds another attribute to the ones mentioned above which is confidentiality. This attribute is defined as the “absence of unauthorized disclosure of information” in [7]. Security is also defined as a system's ability and integrates some of the previous attributes: (1) confidentiality, (2) availability and (3) integrity. As it is observed in Fig. 10, there is a direct relation between dependability and security, since both of them are presented as ideal characteristics of a system.

![Fig. 10: Dependability and security attributes [7].](image)

#### 2.5.2. Dependability analysis approaches

The assessment of these attributes implies an analysis of the system under study, but depending upon the attribute of interest, the nature of the analysis approach could be different. In this section, we only discuss the safety, reliability and security as remaining attributes are out of the scope of this thesis work.

### Safety analysis

Safety analyses are focused on performing a qualitative/quantitative study in systems where failure can lead to a hazard. If this hazard appears in some operational conditions, it can lead to an accident. To prevent this, there are several approaches depending on the information that is derived from the system under study. According to [19], two types of safety analysis can be distinguished: inductive and deductive. The former is composed of those analyses where there is a “reasoning from individual cases to general conclusion” [19]. If this reasoning is applied to a system, then inductive approaches “postulate a particular fault or initiating condition and attempt to ascertain the effect of that fault or condition on system operation” [19]. Many inductive approaches to derive system safety analysis have been developed, but some of the most commonly used ones, according to [19] are Preliminary Hazards Analysis (PHA), Failure Mode and Effect /and Criticality Analysis (FMEA/FMECA) and Hazard and Operability (HAZOP) study. Since most of the related studies presented in Chapter 3 use some of these to model the system failure behaviour, these methods are explained below.

The primary objectives of PHA according to [19] are to identify potentially hazardous conditions in the system and to determine the criticality level associated with them. The difference between this analysis and the rest is that this study “should be conducted as early in the product development stage as possible” [19]. This allows that safety requirements are implemented in the early phases of design, which removes the need to modify the product in future stages, where costs are higher.
FMEA and FMECA are developed to identify both the potential failure modes of different parts of the system and the effect these may produce. Additionally, this analysis provides measures to avoid or mitigate these failures. The difference between FMEA and FMECA is the level at which the criticality of failure is studied, according to [19]. These methodologies also provide templates to be filled with the needed information, due to which they could be considered as consistent analyses.

Lastly, HAZOP is a structured analysis that identifies hazards in the system, but with a key difference. Instead of analysing the components, this methodology studies the deviations of interactions/interconnections between components. To define these deviations, HAZOP proposes the use of guide-words to create and standardise language to define them.

The second group of safety analysis, named deductive approaches, studies the system from a different point of view. In this case, and according to [19]: "In a deductive system analysis, we postulate that the system itself has failed in a certain way, and we attempt to find out what modes of system/component behaviour contribute to this failure". The most common example of deductive analysis is Fault Tree Analysis (FTA), and the automation of the Fault Trees (FT) generated by this analysis is the base of this thesis, so the central concepts of this analysis are presented in Section 2.6. Due to the nature of this analysis, FTA requires to start with a high-level concept, which in the case of this thesis is the system failure. In [19], it is discussed that some previous analysis, which informs how failures are propagated throughout the system is needed, and FMEA/FMECA or HAZOP analysis are proposed as means to derive this information.

**Reliability analysis**

Even though some studies integrate safety analyses with reliability ones, for example in [11], a difference between both is made in this thesis since they assess different properties of the system. While safety deals with system failures which occurrence may result in harm to the user or the environment, reliability addresses system failures that do not have this effect. In systems where hardware components are involved, the most common approach to make reliability analyses is to derive the FT and calculate the reliability of the system. To do so, some metrics provided by the hardware manufacturer, such as Mean Time Between Failure (MTBF), are used to model the reliability of each component that composes the system. This provides the analyser with quantitative information of the system. On the other hand, software-based systems cannot perform such quantitative analysis since these values cannot be determined for software artefacts. To perform a reliability analysis in this type of systems, a qualitative FTA can be derived, as long as a failure of the analysed system does not lead to a hazard.

**Security analysis**

Lastly, security is also an important dependability-attribute to be assessed in modelled systems. To do so, attack trees [45] could be used. Attack trees are very similar to FTA since they are both often graphically represented using the same notation, but the events studied by attack trees, are related with security concerns of the system, which makes them a conventional method used to model threats against computer systems.

**2.5.3. Concerto-FLA**

As explained in Section 2.4, this thesis is developed under the scope of the CHESS toolset. This platform includes several plugins, used to derive different safety and reliability analysis. According to Section 2.5.2, a previous analysis must be conducted to develop FTA, which models the behaviour of the potential failures in the system and how they propagate. The analysis that performs this functionality is integrated into this toolset under the name Concerto-FLA (Failure Logic Analysis) and was proposed in [13] and in [64]. Concerto-FLA allows to introduce a failure on the system inputs and check if this failure produces failure in the output of the system. This analysis is based on FLA techniques such as Fault Propagation and Transformation Calculus (FPTC) [14] and Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures' Analysis (FIIFA) [15]. The Concerto-FLA technique follows four steps: (1) model the structure of a system, (2) model the behaviour of individual components through FPTC rules. These rules, defined using FPTC syntax, indicate the relation between the response of the output and the component's input. (3) Once the rules are established, a fault is injected in the input of the system. (4) execute the analysis. Lastly, a high-level overview Concerto-FLA flow, proposed by [13] is presented in the Fig. 11.
In Concerto-FLA the rules for specifying the failure behaviour are composed of two expressions separated by an arrow. The expression presented to the left of the arrow describes the behaviour of the input port/s. The port is specified for each expression, and then the failure/s related to that port is added (separated from the port by a dot). Both the left and the right side of the arrow can have multiple expressions for the same rule, which are separated by a coma. The right part of the arrow includes the expression regarding the output ports involved in the rule. The syntax for this expression is the same explained for the input expression. A summary of FPTC syntax is presented in Fig. 12, derived from [4].

\[
\begin{align*}
\text{behaviour} &= \text{expression} + \\
\text{expression} &= \text{LHS} \rightarrow \text{RHS} \\
\text{LHS} &= \text{portname}. \ bL | \text{portname}. \ bL (, \text{portname}. \ bL) + \\
\text{RHS} &= \text{\textquoteleft\textquoteright} \text{noFailure} | \text{\textquoteleft\textquoteright} \text{variable} | \text{\textquoteleft\textquoteright} \text{failure} + \text{\textquoteleft\textquoteright} \\
\text{bR} &= \text{\textquoteleft\textquoteright} \text{early} \text{\textquoteleft\textquoteright} \text{late} | \text{\textquoteleft\textquoteright} \text{commission} | \text{\textquoteleft\textquoteright} \text{omission} | \text{\textquoteleft\textquoteright} \text{valueSubtle} | \text{\textquoteleft\textquoteright} \text{valueCoarse} \\
\text{bL} &= \text{\textquoteleft\textquoteright} \text{ wildcard} | \text{\textquoteleft\textquoteright}
\end{align*}
\]

Fig. 12: FPTC rules syntax [5].

Depending on the failure behaviour modelled by these rules, the behaviour of components can be classified into:

- **Source** of failure, the component generated a failure when the input did not receive any.
- **Transformation** of the failure, the component transforms the input failure into a different type of failure.
- **Propagation** of the failure, the component propagates the failure to the output as it is in the input.
- **Sink** of the failure, the input of the component receives a failure, but the component stops it from propagating.

Gallina and Haider [46] proposed a methodology that can be implemented to generate FTs from CHESS ML system models automatically. In addition, they state that there is a need to develop FTs compliant with ECSS standards and provide the tool supported CHESS methodology to do so. This methodology, used in the Concerto project, incorporates the Concerto-FLA (Failure Logic Analysis) technology, which uses FPTC rules to model the failure behaviour at the component level. To derive the fault tree, this approach proposes to use the failure propagation paths, which are included in the results of the analysis. These paths represent the path followed (through the system components) by a failure from its original cause, to the moment it manifests outside the system boundaries. The approach proposed in this thesis follows the guidelines and methodology presented by [46] and tries to solve all its proposed goals.

### 2.6 Fault Tree Analysis

FTA is a top-down, deductive analysis in which a top undesired event is backtracked to its potential causes. The relations/connections of these causes, presented as events, are expressed graphically using Boolean logic. Initially, it was developed in 1962 at Bell Laboratories to analyse the Minuteman I Intercontinental Ballistic Missile Launch Control System [18]. From these days, this method has gained much popularity both as a standardised method of evaluating the safety of a safety-critical system and as a debugging method in the software industry. Some of the standards that require FTA to be conducted are ECSS-Q-ST-40 [51] or ISO 26262 [49].
2.6.1. Graphical Elements

In order to represent FT graphically, some standardised elements are used. These elements are divided between events and gates and all the graphical representations displayed below have been obtained from [19].

Events

According to [19], events are “faults that are associated with component hardware failures, human errors, or any other pertinent events that lead to the undesired event”. Table 1 summarises the main events defined for FT and a description of their purpose.

<table>
<thead>
<tr>
<th>Graphical representation</th>
<th>Element name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td>Basic event</td>
<td>Fault event that requires no further explanation. That is, the appropriate resolution is reached.</td>
</tr>
<tr>
<td><img src="image2.png" alt="Image" /></td>
<td>Conditioning event</td>
<td>A specific condition that applies to any logic gate.</td>
</tr>
<tr>
<td><img src="image3.png" alt="Image" /></td>
<td>Intermediate event</td>
<td>Fault event resulting from the combination of two or more events. These events can be either basic or intermediate.</td>
</tr>
</tbody>
</table>

Gates

Gates are particular elements used in FTs to show the connection between the faulty events (basic or intermediate) and how they relate to lead to the top undesired event. A differentiation between two types of gates is defined: static and dynamic. While static gates represent the connection between events, dynamic gates are used to model dependencies between events and spare components. The graphical representation of both is given in Table 2 and Table 3 respectively. Although this thesis implements only the static ones, both types are presented for the sake of completeness.

<table>
<thead>
<tr>
<th>Graphical representation</th>
<th>Element name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4.png" alt="Image" /></td>
<td>OR gate</td>
<td>The output of the gate produces a fault if any of the inputs produce a fault.</td>
</tr>
<tr>
<td><img src="image5.png" alt="Image" /></td>
<td>AND gate</td>
<td>The output of the gate produces a fault if all of the inputs produce a fault.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graphical representation</th>
<th>Element name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image6.png" alt="Image" /></td>
<td>Priority OR gate</td>
<td>The output of the gate produces a fault if precisely one of the inputs produces a fault.</td>
</tr>
<tr>
<td><img src="image7.png" alt="Image" /></td>
<td>Priority AND gate (PAND)</td>
<td>The output of the gate produces a fault if any of the inputs produce a fault in a specific ordered sequence.</td>
</tr>
</tbody>
</table>
2.6.2. Fault tree generation process

According to [19], some previous information about the possible deviations that lead to system failures is required to generate fault trees. This information can be obtained by some qualitative safety analysis, so FTA is usually presented in combination with FMEA/FMECA or HAZOP analyses. Then, the next steps need to be followed in order to generate an FT:

1. Define the top-event in a clear and unambiguous way.
2. For each event (only the top event in the first iteration) repeat the following:
   2.1. Define the immediate, closest in time and space, events and conditions causing the event.
   2.2. Connect these events through AND/OR gates.
3. Iterate step 2 until an appropriate level of basic events is reached, that is, either the event is a basic event or that additional refinement consumes more work than the benefits it produces.

Finally, a diagram that illustrates how this process needs to be performed is present in Fig. 13, which has been adapted from the procedure presented in [19].

![Fault Tree Diagram]

Fig. 13: FT generation procedure based on [19].

2.6.3. Cut set

A cut set is a description of the different ways in which the top undesired event can occur. Therefore, a cut set is defined as the combination of basic events whose occurrence at the same time can lead to the top event. After deriving the FT of a system, several cut sets are extracted, but to analyse the results of the FT qualitatively, it is necessary to find the minimal cut set. A minimal cut set is defined as the “smallest combination of component
failures which, if they all occur, will cause the top event to occur”[19]. Once this is derived, [19] also proposes that the following conclusions can be drawn depending on the number and size of the cut sets:

- If the size of the cut set is small (low number of basic events happening at the same time), the system presents a high vulnerability. The occurrence of a few events is more likely to happen than the occurrence of more events (big cut set).

- If the number of cut sets found in a system is high, the vulnerability of the system is high. The occurrence of one combination of events out of many is more likely to happen than a few combinations.

- If the minimal cut set consists of only one element, this is considered as a single point of failure. Therefore, there is a high risk of occurrence of the top event.

- If the event is repeated in different parts of the tree, this event is called a common cause failure. The occurrence of this event causes several failures and increases the complexity of applying countermeasures.

2.6.4. Quantitative vs qualitative FTA

To perform an FTA, two approaches have been presented over the years, i.e., qualitative and quantitative. The former is based on the concept explained before: cut sets. Once the FT is generated, the minimal cut set is derived and depending on the characteristics of this and the specifications of the system, countermeasures need to be taken or not. The latter is a mathematical approach that takes into account the failure rates of the hardware component or the software modules which compose the system. Once the FT is produced, this analysis calculates the overall probability of the top event to occur, depending on the events that compose the tree. To do so, gates are transformed to multiplications or additions using the main concepts of Boolean algebra (AND gates are multiplications and OR gates are adding). Once the overall probability is calculated, it is compared to the one that was required (by the standard or by the system specifications if no compliance with any standard is required).

Even though the quantitative approach can be used to make a more complete FTA if it is combined with the quantitative analysis, Concerto-FLA was not designed to provide quantitative data and is not intended to do so. Our approach generates these fault trees, related to the quantitative approach, but does not deal with the analytical part, which can be included as future development of our work.
3. Related Work

Deriving FT is a time-consuming activity and often prone to human errors due to its dependency on the expertise of the individuals deriving it. Additionally, the complexity of the system also contributes to the complexity of fault tree generation. In order to reduce the complexity of the system and analysis, researchers have focused on model-based analysis approaches. In this chapter, we provide the model-based fault tree analysis approaches classified according to the underlying modelling language used for system definition. In [11], the authors defined four different classes of modelling languages, and in this work, we also use the same classification.

The first group discussed uses the modelling languages conformant to Meta-Object Family (MOF) standard, which has been presented by OMG. Mhenni et al. [10] proposed an approach which integrates automatic generation of FMEA and FTA from a SysML system model. To do so, firstly they generate a preliminary version of the FMEA from the model and then manually complete the analysis by a human expert. FT is generated after this, using FMEA deviations and two crucial concepts, i.e., patterns and traversal algorithm. The patterns are used to divide the system into sub-fault trees. According to the elements these sub-fault trees have, four types of patterns are defined that encompass all possible situations in a fault tree. Once all sub-FT are generated, the traversal algorithm is used to combine all them and generate the final fault tree. The work carried out in this thesis also uses SysML constructs for system definition, provided by CHESS ML, which integrates SysML and other modelling languages. However, the main difference is the definition of failure behaviour information for the analysis used to derive the fault trees; Mhenni et al. [10] used FMEA, while our approach uses Concerto-FLA in order to define the failure propagation paths.

Xiang et al. [20] proposed a SysML based approach to generate Dynamic Fault Trees (DFT). To do so, they model dynamic gates (PAND and FDEP) as static ones and use a novel Reliability Configuration Model (RCM) to identify components and relations in the system without using any previous safety analysis. To transform the SysML model into the FT they used the Internal Block Diagram (IBD), for the structure of the system, and the Sequence Diagram (SD) to identify the flow between components. Lastly, failures are modelled as the states of components with four predefined types which are down, functionless, disables and failed

Yakymets et al. [53] proposed a SysML based approach to generate and analyse fault trees. To implement this solution, they model the system using SysML blocks and internal block diagram and then annotate this model with failure behaviour information. This behaviour is represented “as a set of analytical expressions showing how deviations in the block outputs can be caused by internal failures of the block” [53]. After this, the model is transformed into AltaRica language so that FTs can be generated and analysed by external tools. This transformation considers both the information of the component's ports and the output and input flows to derive events, which is similar to the approach proposed by this thesis.

Pai and Dugan [16], proposed a UML based approach to model failure propagation paths between components and generated both SFT and DFT. Even though their FTs generation works automatically, the modelling of the failure behaviour is carried out manually by the designer, which requires a significant amount of time and effort. To model, the failures at the system level both hardware and software components are considered, and a set of stereotypes are proposed for modelling dependencies relations between these components. While this work models both hardware and software components, our work is only focused on modelling the software ones.

The second group of approaches are based on Matlab/Simulink system models. Papadopoulos and Maruhn [22], proposed a Simulink based approach that uses a computer-based HAZOP analysis to derive FTs automatically. This procedure studies all deviations for each component in the system and uses the failure modes, presented in Section 2.5.1, to model the failure behaviour. In addition, they propose a methodology that begins with a local analysis of each component and then propagates failures through the system. Once this analysis is performed, a traversal algorithm is used to derive the FT. Although this approach improves the automatization process, Roth et al. [11] discuss the overall effort needed to develop the analysis, a profound knowledge of each component is required. Tajarrod and Latif-Shahbazi [23] present a Simulink based solution that removes the use of a previous computer-based HAZOP. The FT is automatically generated from a first system model which contains preliminary information of the system. Once this is provided, the user must refine this model by generating an extended one. This refined model should include information about the impact of the components in the top event and how they cooperate. Then, depending on the provided information, components are categorised as "usual" or "redundant". The former is modelled as an OR gate and the latter as an AND gate. However, Roth et al. [11] state that this approach is not recommended for complex systems since the amount of information required to the user for refining the analysis can be excessive.
The third group of approaches are based on a non-conventional modelling language, proposed by Roth et al. [11]. In this, the concepts Domain Structure Matrices (DSM) and Domain Mapping Matrix (DMM) are introduced to model the system. The combination of these previous concepts results in a Multiple Domain Matrix (MDM), which is used to model functions, flows, and failures of the system. To derive such MDM, a five-step process that “allows to analyse and optimise system structures” [11] is proposed. Then, the generated MDM contains the failure behaviour in the system, and an OR gate is inserted for each failure found in it. In addition, two concepts are introduced to analyse the obtained results, namely distance and occurrence. Even though this approach allows the use of abstract and functional concepts, the metrics proposed to determine the safety of the system are not formal. Besides, this approach does not implement automatically AND gates, so the user needs to add them manually. Roth et al. [24] also propose a refinement of the solution presented in [11]. The previous work is further developed by using a hierarchical structure of the system, which is now divided into system, sub-systems, collections, and components. Despite this improvement, the problem concerning, AND gates persist, and they still need to be added manually to the fault tree.

Finally, other modelling languages, which are not as commonly used in literature as the previous ones, are analysed. Some approaches based on AADL (Architecture Analysis and Definition Language) model systems are provided to generate FTs automatically, such as the ones presented by Joshi et al. [25], and by Dehlinger and Dugan [26]. The main issue with this modelling language is that failures need to be modelled using the Error Model, which fails to capture some failure modes which need to be considered separately. The approach proposed by [25] generates SFTs from AADL modelled systems using this Error Model. From the results provided in this previous approach, [26] follows this work and includes DFTs in their implementation. Besides, Feiler and Delange [41] also present an AADL model-based approach, similar to the previous ones. This approach also presents the issue regarding the Error Model but introduces the EMF-based Fault Tree Analysis (EMFTA) open source tool, which is used in this thesis. This tool includes a meta-model to defined fault trees like models and a visualizer functionality that allows the user to generate models based on industry standards. Majdara and Wakabayashi [27] propose an approach based on the use of function tables, decision tables and states diagrams to model the behaviour of the components, and a directed graph to model their relations and dependencies. Once all the previous information is derived, a trace-back algorithm, which looks for the components and its implemented function in the system, is used to generate FTs. Even though the analysis is very detailed, [11] states that the amount of effort required is significant. Besides, [11] also discusses that this method is not compatible with any of the conventional modelling languages.

In contrast to all the above mentioned, our work is based on automatically generating fault trees from systems modelled in CHESS ML. This language is implemented in the CHESS platform as an extension of other OMG standard languages such as UML, SysML and MARTE. The use of this platform allows our solution to benefit from some of the functionalities which are already implemented. Instead of modelling the failure behaviour with traditional techniques, such as FMEA/FMECA or HAZOP, our approach proposes to use one of CHESS platform’s analysis, i.e., Concerto-FLA. This analysis provides all the failure propagation information (along with the failure propagation paths) without requiring the user to refine the failure behaviour of each component in the system once the analysis is performed, which is one of the main issues previously mentioned related to the traditional techniques. Besides, Concerto-FLA uses the traditional failure modes (presented in Section 2.5.1), allowing to model the failure behaviour in a standardised way, which some other approaches, as [20] or [11], are failing to accomplish. In addition, the use of the CHESS toolset permits to store both the design and safety information in a unique model. Thereby, there is no possible loss of data when the design model is used to derive the safety one, as other approaches such as [53] may face, and the M2M transformation process is reduced. Finally, according to [62], there is a need for a methodology that can be used in a unified toolset to derive such fault trees. By including the design, failure modelling, and fault tree generation in CHESS platform, our solution aligns with the needs of the development process of safety-critical systems, which, to the best of our knowledge, is not accomplished by most of the related approaches.
4. Scientific Method

This chapter describes the research methodology followed to find the solutions that address the problems proposed in this thesis. The procedure used is adapted from the work proposed by Leedy and Omrod in [48] to the needs of this thesis. The final process is shown in Fig. 14 and is explained below.

According to [48], “research originates with a question or problem”. Based on this, the first step is to identify and understand which is the problem that we aim to solve. After this, a review of related literature is performed by studying both the concepts needed to understand the problem and previous related works. The later provides the knowledge needed to understand what scientific gaps our work is trying to fill. Then, the proposed problem is divided into manageable units or subproblems, which combined solutions must resolve the main problem. Next, according to such decomposition, research questions are formulated in order to address them.

Once all these steps are performed, a solution that solves the problem in a scientific manner is proposed and is later implemented and tested to determine its success. This evaluation process is done by applying the implemented solution into a real scenario, which in this thesis is done through a case study, presented in Chapter 7. With the knowledge gained from the related literature study, we can determine if our solution meets the proposed goals, or if a different approach is needed. As [48] proposes, “research is, by its nature, cyclical or, more exactly, helical”, that is, the research process may end up providing a valid solution to the proposed problem, but also can arise one or more new problems. Due to this, the process at this point is defined as iterative, since for each new problem or question, the methodology needs to be repeated until the problems of interest are solved.

After determining if the proposed solution answers the questions and solves the problem, conclusions are drawn from the evaluation process. Finally, all the findings are reported so other researches can benefit from this work.

Fig. 14: Scientific method used in this thesis based on [48].
5. **Problem Formulation**

This chapter presents and analyses the problem addressed by this thesis and is organised as follows: Section 5.1 presents the problem and motivates why it needs to be solved. Then, Section 5.2 analyses the problem and extracts the formulated questions needed to address the previous problem.

5.1. **Problem formulation**

High-integrity, cyber-physical embedded systems are becoming complex due to the growing demand for new functionalities and recent digitalisation trend. This digitalisation is targeting diverse domains, e.g., automotive industry, space systems, railway systems, avionics and process automation industry. Depending on how critical the interaction of these systems with the environment is, a failure in providing the service may result in catastrophic consequences. Due to this, the development of such systems is regulated by the authorities, and compliance with the set of rules is mandatory; this also contributes to the complexity, negatively.

These rules are provided as requirements in the form of a standard document – the development of such critical systems needs to be compliant with these requirements for certification and qualification purposes. Standards require that evidence need to be presented in order to claim that a system is safe and reliable. One such evidence is in the form of fault trees, which reasons about the safety and reliability of a system. In the space domain, ECSS-Q-ST-30C [54] and ECSS-Q-ST-40-12C [51] require such evidence and refer to IEC-61025 [50] for guidelines on how to conduct this analysis.

These fault trees represent how a single failure can lead to a system level failure which can potentially cause harm, also called hazard. This hazardous event, combined with the proper environmental and operational conditions, can lead to an accident. Due to this, this evidence can unveil issues undetected during the development of such systems. The main problem is that software-intensive systems are growing in complexity, which makes this process a time-consuming task if it is done manually.

This thesis presents a solution to reduce the time and costs associated with this process, in the form of an automatic implementation that generates the FT, compliant to the standards requirements, associated with a given modelled system. The approach is implemented as an Eclipse plugin and integrated into the CHESS toolset. CHESS, which provides a modelling and dependability analysis language, could benefit from this plugin concerning the automatic generation of the certifiable evidence from one of the available analysis, i.e., Concerto-FLA.

5.2. **Problem analysis**

Based on the above discussion we formulated the following three research questions:

- **Which existing failure analysis technique can be exploited to enable automatic generation of fault tree using model-based approaches according to the current state of the art?**

  As Chapter 3 proposed, several techniques can be used to model the failure behaviour of the system components. Each technique offers different advantages and disadvantages since some of them cannot be fully automated while others cannot model all the elements. When considering a suitable technique for the proposed solution, the choice made needs to be motivated. Due to this, a comparison between the techniques discussed in related studies and the one implemented in our approach needs to be done. Then, if the selected approach presents some limitations to address the presented problems, modifications need to be proposed.

- **Which will be the structure of the meta-models required for automatic generation of fault trees?**

  The solution proposed in this thesis implements a model-to-model (M2M) transformation, which derives FTs from the results provided by the previous failure behaviour analysis. Understanding the meta-models in which these models are based, can provide us with the needed knowledge, so this M2M transformation algorithm can be later designed and implemented.

- **How to implement the automatic generation of fault tree in CHESS framework?**

  Since this thesis is developed under the scope of CHESS toolset, we need to determine how our solution fits in this framework. Understanding which functionalities from this platform are used and what elements are derived are critical to the success of this approach. A methodology of how the solution is implemented in this platform needs to be derived so that the process can be done systematically.
6. Automatic fault tree generation from system models

In this chapter, the research questions proposed in Section 5.2 are answered, and the proposed solution is presented. This chapter is organised as follows: Section 6.1 answers the first research question by analysing all the techniques used in related studies to model failure behaviour and motivates why Concerto-FLA is used. Section 6.2 presents and describes the meta-models involved in the transformation process. Section 6.3 presents how the solution is integrated under the CHESS toolset and the interaction with the existing solutions of this platform. Lastly, Section 6.4 presents the proposed solution that addresses the problem presented in Section 5.1.

6.1. Which existing failure analysis technique can be exploited to enable automatic generation of fault tree using model-based approaches according to the current state of the art?

As presented in Chapter 3, the automatic generation of FTs from model-based systems needs from a previous analysis to be carried out, which is aligned with the guidelines proposed in [19]. This analysis is used to model the failure behaviour of each component in the system and is critical for the development of FTs. Some of the presented approaches use traditional safety analysis techniques, such as FMEA/FMECA or HAZOP, which have been automatized to derive this behaviour information. The main issue regarding the use of these techniques to model failure behaviour is their lack of flexibility since they require filling a template with concrete information. Therefore, these analyses cannot be fully automated and require from the user to complete the results once they are generated.

However, other approaches use novel techniques to derive the failure behaviour information and overcome the previous problem. While these techniques allow the full automatization of the process, they fail both to model failures using the traditional modes and to derive all the needed elements. In addition, the effort required from the user to decorate the components with failure behaviour in some of these approaches ([11], [25], [27]) is significant, as Chapter 3 discusses.

Since this thesis is developed under the scope of the CHESS toolset, the analysis used to model the failure behaviour is Concerto-FLA, presented in Section 2.5.3. To further motivate our choice, a comparison of Concerto-FLA is carried out with the techniques used in related studies to model the failure behaviour, presented in Chapter 3. Table 4 provides the comparison of these techniques according to the characteristics essential to the solution and to achieve substantial improvement over the existing researches. These characteristics present the main issues identified for the existing solutions, which are eliminated in our approach. Next to the used method, a reference to the related solution that implements it is included.

<table>
<thead>
<tr>
<th>Method used to model failure behaviour</th>
<th>Fully automated</th>
<th>Deals with traditional failure modes</th>
<th>Ability to integrate all static gates</th>
<th>Low effort required</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMEA/FMECA [10]</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Error Model [25, 26]</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability Configuration Model (RCM) [20]</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HAZOP [22, 23]</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DSM [8, 24]</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Concerto-FLA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Comparison of methods used to derive failure information.

The primary goal of the thesis is to generate fault trees automatically. Therefore the support of full automation is included in Table 4 as an important characteristic of the analysed approaches. That is, once the system and the failure behaviour are modelled, no additional information is required from the user. Another key aspect is that the failures modes used to model failure behaviour are the traditional ones (presented in Section 2.5.1). These can define any specific and domain dependent failure in an abstract manner. Hence, they can provide support for the analysis of heterogeneous systems and independent of the domain they belong to. The ability to integrate all the static gates (i.e., AND, OR) is also a critical functionality since the lack of these would condition the results, and the solution would be incomplete. As previously discussed, one of the primary goals is that the effort required to use the solution is reduced. Although this looks similar to the fully automated functionality, as explained in Chapter 3, some fully automated solutions presented in [11], [24], [25] or [26] require a refinement of the modelled failure
6.2. Which will be the structure of the meta-models required for automatic generation of fault trees?

Due to the fact that this thesis is based on model-based systems, it can be stated that an FT form like model needs to be generated from the Concerto-FLA results. Hence, the core work of the thesis is to develop an M2M transformation. As Section 2.2 presented, models are implementations of a given meta-model, which defines a standard structure for all the models which conform to it. Therefore, a critical issue for the success of the solution is to identify how many models are involved in the transformation process. To implement such transformation, a mapping between the elements of these models needs to be carried out.

In our approach, the source model is represented by the results obtained from Concerto-FLA and the target one is the FT form like that is derived. However, no previous information about this transformation exists in CHESS toolset, and there is no fault tree meta-model included in this platform. Therefore, a target meta-model needs to be implemented, or an existing meta-model could be reused if it serves the purposes.

6.2.1. Source meta-model: FLAMM

In the case of the Concerto-FLA results model, the meta-model is presented by the Concerto project in one of its published deliverables [40], under the name of FLAMM (FLA Meta-model). In their own words, FLAMM is defined as the meta-model that “has the elements needed to support the hierarchies of components, connections and failure logic behaviour needed to perform the Concerto-FLA analysis”. This meta-model is presented in Fig. 15 and its elements are explained in detail below.

![FLA Meta-model](image)

Fig. 15: FLA Meta-model [40].

Four main elements can be found in this meta-model, which are used in our proposed transformation:

- **Component**: Is the primary unit of the system. As Fig.15 shows, two types of components are defined; composite and simple. According to this meta-model, the composite component is composed either by simple components or other composite components. However, we limit this in our approach and only address the situation where a composite component is composed of simple components. Therefore, the proposed approach considers that the composite component represents the modelled system. A key attribute named parent is defined to differentiate this composite type from the simple one. If a component has no parent component, then it is a composite one, while in the other case, it is a simple component contained in the composite. Both simple and composite components have input and output ports, used to provide or require information to other components, but only the simple component can have rules. These rules are used to model the failure behaviour of individual components, and the combination of all the modelled components within the system provides the failure behaviour of the system. Therefore, rules are not needed for composite components.
• **Port**: Components have ports, so they can communicate and pass information to other components. There are two types of ports: `inputPorts`, which receive the information components need, and `outputPorts`, which send the information processed to the connected component. One component can have many input and output ports. Ports have one key attribute titled as `connectedPorts`, which defines all the ports that are directly connected to this specific port. In our approach, we used this attribute to reason on the backpropagation of the failures. Besides, a port also has an `owner` attribute, which represents the owner component of that specific port.

• **Rule**: Components (in our approach only simple components) are composed by `rules`. These are used to model the behaviour of the component for a given failure. As we presented in Section 2.5.3, Concerto-FLA rules are defined using FPTC syntax, which means that each rule is composed of an `inputExpression` and an `outputExpression`. Both expressions have the same attributes: port and failure. By studying the rules for the simple components and the ports, we can determine the propagation paths that one failure follows through the whole system. One component can have many rules, and each rule can have many input and output Expressions.

• **Failure**: Failure is one of the main elements of interest in our work and has attributes such as `id` and `previousFailures`. The former is used to identify the failure modes, which were defined in Section 2.5.1. The latter one is used to determine preceding failures, which are causes of this specific failure. While the `id` is a unique value, the `previousFailures` attribute may have many causes.

### 6.2.2. Target meta-model: EMFTA

In order to support the fault events and their relation to each other in the form of logic gates, a meta-model containing different elements as the representation of these events and gates is required. The implementation of such meta-model enables the support of containing the required information for the generation of fault trees. Instead of developing a novel meta-model, an open source relevant meta-model is found in the literature [42] and is referred to as EMFTA tool. In addition to the meta-model, the tool provides an editor for visualising the fault trees. This meta-model was introduced in [41], which was lately integrated under the EMFTA tool [42] and is used by other studies such as [43]. EMFTA tool provides a meta-model under the same name, which is presented in Fig. 16.

In addition, EMFTA tool includes a set of characteristics that motivated its choice and are as follow:

- The meta-model is conformant to the MOF standard and thus part of the OMG MDA initiative, which is the main feature we are searching.
- The meta-model is developed on top of Eclipse Modelling Framework (EMF). This feature is needed since both the source meta-model and CHESS ML are also developed using EMF. Besides, ETL requires that the involved meta-models need to be developed using this framework.
- The meta-model is included as part of an open source software. Since we want to integrate the solution under the CHESS toolset, one of its principles is that all software involved needs to be open source.

Due to all the aforementioned, EMFTA is selected as the tool to integrate FTs in our transformation process.

![Fig. 16: EMFTA meta-model [43.]](image)
In this target model there are three main elements, which are of most interest in the context of our approach, and are discussed in detail as follows:

- **FTAModel**: This element represents an FT and contains all its events and gates. An EMFTA model can have many FTAModels, where FTAModel has two attributes, i.e., name and events. The former refers to the unique identifier of the FT, while the latter represents the containing events. Fig 17 shows a sample FTAModel represented in the EMFTA visualizer tool.

![FTAModel Example](image)

- **Event**: Each failure of each port is defined as an Event in our approach, and has two attributes, i.e., type and name. Even though the latter is a unique identifier of the element, there can be repeated events names in the FTAModel, since one failure can be the cause of different failures. Regarding the Event type, it is included using the Enumeration style, that is, it only accepts the values presented in the meta-model. To this end, our approach only incorporates the following two types of events: Intermediate and Basic. Lastly, another attribute that events own is the gate, which is used to connect the event to its child events. EMFTA visualizer provides different visualisations for these events, depending on the type, which are illustrated in Fig. 18.

![Event Types](image)

- **Gate**: This element is used to determine the relationship between the parent and the child event. Similar to the other elements, it also has a name attribute to identify different gates uniquely. The events attribute provides all related child events to this gate. The attribute titled as Gate Type provides the types of gates. To this end, our approach incorporates only the static types of gates, i.e., AND and OR gates.

Once both meta-models, i.e., the source and target, have been defined, the next step is to establish the transformation from the source model to the target model. This transformation takes all the required information from the FLAMM to populate the EMFTA, hence generates a corresponding FT. The proposed transformation and its implementation are discussed next in the following sections.
6.3. How to implement the automatic generation of fault tree in CHESS framework?

The proposed automatic fault tree generation solution is also implemented as an Eclipse plugin and integrated into the CHESS toolset. In order to generate FT automatically using CHESS toolset, three main steps are required to be performed. The first two steps are already supported in the toolset, and the third step is provided by the implementation of our approach in the context of this toolset. In the following list we provide a detail description of each of the step, to show the integration of fault tree generation in CHESS toolset.

1. **Modelling the system:** The first step required is to model the system under study using CHESS ML language. CHESS platform defines this language as “a collection-extension of subsets of standard OMG languages (UML, MARTE, SysML)” [38]. The system (as stated in Section 6.2.1) is represented as a composite component, composed by a set of simple components. These simple components are connected to each other so that the information can be transmitted through the system. Each component is provided with a set of ports, that enable the connections to other components. These ports provide or require the information to the interfaces, which act as a link between connected components and transmit the information through some operations. In addition, all simple components have an implementation type, which inherits all the specifications from the component, so these can be instantiated and connected. An example of these connections is provided in Fig. 19.

![CHESS component implementation example](image)

Fig. 19: CHESS component implementation example [3].

2. **Failure behaviour decoration and Concerto-FLA:** After modelling the system, the next step is to model the failure behaviour of each component following the guidelines proposed in Section 2.5.3. To do so, each component needs to be categorised depending on how it behaves for a given failure. The component either propagates, transforms, sinks or creates the failure. This behaviour relies on the internal implementation of the component, which at this level is abstracted, so a user can basically think of any implementation and define the rules accordingly, depending upon the need. Since this analysis can be performed several times, our solution enables to use this type of analysis in the early phases of design, when the information the user has about the system is reduced.

To define this behaviour, Concerto-FLA includes FPTC rules, which enable to model failure behaviour in a simple syntax (presented in Section 2.5.3), by using an editor, which shown in the Fig. 20. As this figure shows, FPTC rules are composed of two parts separated by an arrow. The left side of the arrow represents the input port or ports which participate in the rule and the failure value that is happening in this port. In the example provided in Fig. 20, only one input port and failure are defined, but several ports and failures can be involved by using a coma to separate them. This same process is used for the right side of the arrow, with the difference that it is used to define the behaviour of the output port involved in the given rule.

Even though rules are also applicable for the connections in the system, our approach only considers that rules are defined for components, but not for connections. Therefore, a failure sent from one component to another through a connection is always propagated, that is, the connection itself never modifies the failure.

Once this process is followed for the existing components, the input ports of the composite component (i.e., system) need to be decorated with the failure modes that want to be analysed. Then, Concerto-FLA
Enrique Zornoza Moreno  
Model-based approach for automatic generation of IEC-61025 standard compliant fault trees

3. Automatic FT generation: With the previously generated model, the proposed solution generates FTs in the form of a plugin integrated into the CHESS toolset. To do so, an M2M transformation is performed using Epsilon Transformation Language (ETL). The process followed to derive our approach started with the selection of a target meta-model to model fault trees. In addition to this, a deep understanding of such meta-model (presented in Section 6.2) was required for designing the mapping between the models. Then, this mapping algorithm is developed in ETL to perform the transformation. Finally, this algorithm is implemented into an Eclipse plugin and integrated with EMFTA visualizer so that the FTs can be derived automatically and exported by the user. The design and implementation of this concepts into our solution are explained in detail in the following section.

6.4. Proposed solution: Automatic FT generation

In this subsection, the proposed solution to automatically generate ECSS compliant fault trees is presented. To do so, the approach is divided into two steps. The former describes the proposed M2M transformation algorithm and its implementation in ETL, and the latter presents the integration of this transformation algorithm under an Eclipse plugin.

6.4.1. M2M transformation algorithm

The proposed transformation between FLAMM and EMFTA is described at two different abstractions since the transformation itself is independent of the platform opted for its implementation. Therefore, this subsection provides in-depth reasoning about the designed algorithm that maps the elements of both meta-models. Next, we provide the implementation of these algorithms using ETL.

Algorithm for mapping the elements of source and target meta-models

To perform the proposed mapping between the elements of both models, we need to identify which elements from the source model need to be transformed so that the target model can be generated. The designed mapping is presented in Table 5 below. Nevertheless, it is essential to remark that there is no element from the source model that has all the needed information to create an element of the target meta-model. Therefore, the elements of the source meta-model collaborate with each other to derive a target model element, which implies that the proposed mapping is not an exact equivalence. One example that proves this statement is that failures are translated into events (there are as many events in the target model as failures in the source model), but this does not imply that,
with the information provided by the failure, an event can be created. Since events need to be uniquely identified, the events are identified not only with the failure id but also with the name of the port that causes the failure and the name of the component owner of such port.

Table 5: Mapping between the source and target meta-model’s elements.

<table>
<thead>
<tr>
<th>FLAMM</th>
<th>Attribute</th>
<th>EMFTA</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>name</td>
<td>Event</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>owner</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>connectedPorts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>previousFailures</td>
<td>Event</td>
<td>Event Type</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td></td>
<td>name</td>
</tr>
<tr>
<td>Rule</td>
<td>Expression</td>
<td>Gate</td>
<td>Gate Type</td>
</tr>
<tr>
<td>Port (outputPort of composite component)</td>
<td>failures</td>
<td>FTAModel</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Owner</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another critical functionality provided in our solution is how the Gate Type is determined. As we stated in Section 2.6.1, our approach only considers static gates, which is a representation of the connection between the two elements. Considering the source model, the elements which are connected in this model are the ports, but since there are two types of ports, input and output, the connection is different depending on the case. Four different types of connections exist in Concerto-FLA, depending on the involved ports, but the provided mapping considers three of these cases as equivalent. The reasoning followed to determine the gate type is the following:

- Connection between inputPort of Composite Component and inputPort of Simple Component/ outputPort of Simple Component and inputPort of a different Simple Component/ outputPort of Simple Component and outputPort of Composite Component: Since these connections are direct and, as it was mentioned in Section 6.3, rules are not included in the connection element for our approach, the failures are propagated. Therefore, this situation is always transformed into an OR gate in the target model.

- Connection between inputPort of Simple Component and outputPort of the same Simple Component: This case is the most complex one to deal with, and whenever this situation appears, the configuration of the rules needs to be checked for this specific component. If the rule has more than one inputExpression, all the inputExpressions need to occur at the same time, which is translated into an AND gate. If the number of inputExpressions in the rule is one, the element is mapped to an OR gate.

In addition to the mapping, it is also essential to determine which is the order that the proposed algorithm uses to map the elements and derive the FT model. To do so, Fig. 21 presents the steps taken by the mapping algorithm, as well as the order in which they are performed, in the form of a flowchart. Three main tasks are developed by this algorithm and are presented in Fig. 21 as steps 1, 2 and 3. The functionalities each step performs, are explained below.
1. Identify and create all FTAModels: As previously mentioned, the target model needs to have as many FTAModels, as failures there are in the source model. Besides, only those for which the id is different from “noFailure” value are included. Since composite components are considered as the system in our approach, output ports of the composite component represent the boundary of the system. Therefore, a failure in one of these ports leads to a failure of the whole system. Then, an FTAModel for each failure in the output port of the composite component is generated and stored with the other created FTAModels. To implement this functionality, all the ports from the source model (flammModel in Table 6) are identified and stored (line 4 in Table 6). Next, those that belong to a composite component and which are of output type are selected (line 6 in Table 6). From these ports, all the failures they contain are used to create FTAModels. Table 6 presents the algorithm for this step.

Table 6: Algorithm 1 for deriving FTAModels.

<table>
<thead>
<tr>
<th>Algorithm 1 Identify and create all FTAModels (flammModel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. procedure: Identify and create all FTAModels (flammModel)</td>
</tr>
<tr>
<td>2. P ← ∅ : P is a variable that will store all ports existing in flammModel</td>
</tr>
<tr>
<td>3. F ← ∅ : F is a variable that will store all generated FTAModels</td>
</tr>
<tr>
<td>4. P ← getAllPorts(flammModel)</td>
</tr>
<tr>
<td>5. for all p ∈ P do</td>
</tr>
<tr>
<td>6. if (isOutputPort(p) and belongsToComposite(p)) then</td>
</tr>
<tr>
<td>7. fa ← getFailures(p)</td>
</tr>
<tr>
<td>8. for all f ∈ fa do</td>
</tr>
<tr>
<td>9. if (getId(f) ≠ “noFailure”) then</td>
</tr>
</tbody>
</table>
2. **Create top-event and related OR gate:** The following step in the FT construction process, according to the procedure explained in Section 2.6.2, is to identify the top-event. Once, this top-event is located, the process of populating the fault tree begins. As stated in step 1, the outputPort of the composite component symbolises the boundary of the modelled system. Therefore, the top-event for each FT is the failure for the specific outputPort used to derive the FT. To create this event, from all the existing ports in the source model, we need to identify those which are of output type and which owner is a composite component (line 9 in Table 7). From those ports, we also need to know which failures they are causing (line 10 in Table 7). Once the failure id is identified, as well as the port and the owner component, we can determine to which FTAModel this top-event belongs, since the name of the FTAModel already contains this information.

After this, the top-event is created and added to the owning FTAModel (lines 16-19 in Table 7). Since the top-event is always derived from an outputPort of a composite, this port is connected to an outputPort of a simple component. Therefore, the gate type for this situation is of type OR. Once the top-event has been defined, the next analysed port needs to be determined in order to populate the FTAModel with all its children events. To perform this task, an external method that iteratively creates the events is used until all the basic events are identified. Since the connection between the outputPort of the composite component and the connectedPort is one to one (no rules are involved), this populateFT method is invoked for all the connected ports to this port (line 25 in Table 7). Table 7 below presents the algorithm for this second step.

### Table 7: Algorithm 2 for port transformation and FT population.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><strong>procedure:</strong> Create top-event and related OR gate (flammPort)</td>
</tr>
<tr>
<td>2.</td>
<td>Procedure: Create top-event and related OR gate (flammPort)</td>
</tr>
<tr>
<td>3.</td>
<td>F ← Generate FTAModels(flammModel)</td>
</tr>
<tr>
<td>4.</td>
<td>fe ← true : fe is a variable that is used to check if the port has one or many failures</td>
</tr>
<tr>
<td>5.</td>
<td>C ← ∅ : C is a variable that will store all connectedPorts to the flammPort</td>
</tr>
<tr>
<td>6.</td>
<td>C ← getAllConnectedPorts(flammPort)</td>
</tr>
<tr>
<td>7.</td>
<td>if (isOutputPort(flammPort) and belongToComposite(flammPort)) then</td>
</tr>
<tr>
<td>8.</td>
<td>for all p ∈ P do</td>
</tr>
<tr>
<td>9.</td>
<td>if (isOutputPort(p) and belongToComposite(p)) then</td>
</tr>
<tr>
<td>10.</td>
<td>fa ← getFailures(p)</td>
</tr>
<tr>
<td>11.</td>
<td>for all f ∈ fa do</td>
</tr>
<tr>
<td>12.</td>
<td>for all ft ∈ F do</td>
</tr>
<tr>
<td>13.</td>
<td>if (getName(ft) = id(f) + getName(flammPort) + getOwnerName(flammPort)) then</td>
</tr>
<tr>
<td>14.</td>
<td>if (fe = true) then</td>
</tr>
<tr>
<td>15.</td>
<td>if (id(f) ≠ “noFailure”) then</td>
</tr>
<tr>
<td>16.</td>
<td>e ← create a new Event with the proper name and Intermediate Type</td>
</tr>
<tr>
<td>17.</td>
<td>g ← create a new Gate of type OR</td>
</tr>
<tr>
<td>18.</td>
<td>e.gate ← g</td>
</tr>
<tr>
<td>19.</td>
<td>ft.events ← ft.events ∪ e</td>
</tr>
<tr>
<td>20.</td>
<td>fe ← false</td>
</tr>
<tr>
<td>21.</td>
<td>for all con ∈ C do</td>
</tr>
<tr>
<td>22.</td>
<td>cfa ← getFailures(con)</td>
</tr>
<tr>
<td>23.</td>
<td>for all cf ∈ cfa do</td>
</tr>
<tr>
<td>24.</td>
<td>if (getId(cf) = getId(f)) then</td>
</tr>
<tr>
<td>25.</td>
<td>populateFT (con, cf, e) – Algorithm 3</td>
</tr>
<tr>
<td>26.</td>
<td>end if</td>
</tr>
<tr>
<td>27.</td>
<td>end for</td>
</tr>
</tbody>
</table>
3. **Invoke FTAModel population algorithm:** To implement this method, we use an iterative algorithm. Following the procedure discussed in Section 2.6.2, from the given top-event, the algorithm needs to look for the related events backwards until an inputPort of a composite component (boundary of the system) is found. Then, the connectedPorts and rules attributes are used to follow the propagation path of the failure, so that the event can be created, and its children events can be found.

As Table 7 shows in line 25, this method is invoked using four attributes derived from the parent event, which are (following Table 7 notation):

- **con:** Analysed port which contains the failures, the owner and the connectedPorts.
- **cf:** Analysed failure which appears in the con port.
- **ft:** FTAModel which needs to be populated.
- **e:** Parent event, for which its child events needs to be found.

This method is repeated iteratively until the final condition (the analysed port is an inputPort of a composite component) is reached. The algorithm determines the failure events iterating over the Port element, and three possible cases can arise. These cases are due to the existence of different types of ports, i.e., inputPort of the composite component, inputPort of the simple component and outputPort of the simple component. The case outputPort of composite component is already handled by the algorithm described in Table 6 and is not included here. Next, we discuss in detail these three cases.

**Case 1: OutputPort of a simple component**

This case is always the first one used by our solution since the port that invokes this method is always an outputPort of a composite component, which is connected to an outputPort of a simple component (line 2 in Table 8). Therefore, the failure id related to the parent event is going to be the same one, since direct connections always propagate failures. Then, a new event is created in the fault tree (line 4 in Table 8), with the port, owner and id information previously obtained. Next, the attribute previousFailures related to the analysed failure is used for determining the Event Type (lines 5 and 9 in Table 8). If this attribute is empty, the failure is the root cause and therefore, for the corresponding event, the Event Type shall be set as Basic (line 6 in Table 8). In any other case, this failure is not the root cause, and the Event Type is set as Intermediate (line 10 in Table 8). When the event type is Basic, no gate is needed since this branch of the FTAModel does not need to be further developed. On the other hand, to determine the type of the gate, the rule which contains an outputExpression, with the same port and failure, is selected (lines 23-26 in Table 8). In this situation, the number of inputExpression of this rule is checked, and if this value is higher than one, an AND gate is added (lines 14-15 in Table 8), but for all other cases, the Gate Type is set as OR (line 17 in Table 8).

After this, the next port to be analysed is determined. To do so, the FTPopulation method is invoked for each inputPort contained in the previous rule’s inputExpression (line 27 in Table 8). Table 8 below, shows the algorithm developed for this case.

**Table 8: Algorithm 3 for FTPopulation. Case 1.**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.</td>
<td>end for</td>
</tr>
<tr>
<td>29.</td>
<td>end if</td>
</tr>
<tr>
<td>30.</td>
<td>end if</td>
</tr>
<tr>
<td>31.</td>
<td>end if</td>
</tr>
<tr>
<td>32.</td>
<td>end for</td>
</tr>
<tr>
<td>33.</td>
<td>end for</td>
</tr>
<tr>
<td>34.</td>
<td>end if</td>
</tr>
<tr>
<td>35.</td>
<td>end for</td>
</tr>
<tr>
<td>36.</td>
<td>end if</td>
</tr>
<tr>
<td>37.</td>
<td>end procedure</td>
</tr>
</tbody>
</table>

**Algorithm 3 FTPopulation(flammPort,flammFailure,emftaFTAModel,emftaEvent)**

1. procedure: FTPopulation(flammPort, flammFailure, emftaFTAModel, emftaEvent)
2. if (isOutputPort(flammPort) and belongToSimple(flammPort)) then
3. if ( id(flammFailure) ≠ “noFailure”) then
4. evchild ← create new event with proper name
5. if (previousFailures.isEmpty(flammFailure)) then
Enrique Zornoza Moreno  
Model-based approach for automatic generation of IEC-61025 standard compliant fault trees

6. \( \text{evchild.type} \leftarrow \text{Basic} \)
7. \( \text{emftaFTAModel.events} \leftarrow \text{emftaFTAModel.events} \cup \text{evchild} \)
8. \( \text{emftaEvent.gate.events} \leftarrow \text{emftaEvent.gate.events} \cup \text{evchild} \)
9. \( \text{else} \)
10. \( \text{evchild.type} \leftarrow \text{Intermediate} \)
11. \( R \leftarrow \text{getRules}(	ext{getOwner}(	ext{flammPort})) \)
12. \( \text{for all } r \in R \text{ do} \)
13. \( \text{if } (\text{getOutPutExpressionPort}(r) = \text{flammPort}) \text{ then} \)
14. \( \text{if } (\text{getInputExpression}(r).\text{size}() > 1) \text{ then} \)
15. \( \text{evchild.gate.type} \leftarrow \text{AND} \)
16. \( \text{else} \)
17. \( \text{evchild.gate.type} \leftarrow \text{OR} \)
18. \( \text{end if} \)
19. \( \text{end if} \)
20. \( \text{end for} \)
21. \( \text{emftaFTAModel.events} \leftarrow \text{emftaFTAModel.events} \cup \text{evchild} \)
22. \( \text{emftaEvent.gate.events} \leftarrow \text{emftaEvent.gate.events} \cup \text{evchild} \)
23. \( \text{for all } r \in R \text{ do} \)
24. \( \text{if } (\text{getOutPutExpressionPort}(r) = \text{flammPort}) \text{ then} \)
25. \( \text{for all } \text{inp} \in \text{getInputExpressionPorts}(r) \text{ do} \)
26. \( \text{for all } \text{fail} \in \text{getFailures}(	ext{inp}) \text{ do} \)
27. \( \text{FTPopulation(inp, fail, emftaFTAModel, evchild)} \)
28. \( \text{end for} \)
29. \( \text{end for} \)
30. \( \text{end if} \)
31. \( \text{end for} \)
32. \( \text{end if} \)
33. \( \text{end if} \)
34. \( \text{end if} \)
35. \( \text{end procedure} \)

**Case 2: InputPort of a simple component**

This second case is used when the method receives a port of type input owned by a simple component (line 2 in Table 9). In this situation the connection between this inputPort and the following port is one to one, so the failure is propagated. Then, the event and the Event Type are created using the failure and port information, following the same structure used for Case 1. If the Event Type is defined as Intermediate, the type for the created gate is OR (line 11 in Table 9), and then, the FTPopulation method is invoked for each connectedPort, with the previous failure value (line 17 in Table 9). Table 9 presents the developed algorithm for this situation.

**Table 9: Algorithm 3 for FTPopulation. Case 2.**

<table>
<thead>
<tr>
<th>Algorithm 3 FTPopulation(flammPort, flammFailure, emftaFTAModel, emftaEvent)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case 2</strong></td>
</tr>
<tr>
<td>1. procedure: FTPopulation(flammPort, flammFailure, emftaFTAModel, emftaEvent)</td>
</tr>
<tr>
<td>2. if (isInputPort(flammPort) and belongToSimple(flammPort)) then</td>
</tr>
<tr>
<td>3. if (id(flammFailure) ≠ “noFailure”) then</td>
</tr>
<tr>
<td>4. evchild ← create new event with proper name</td>
</tr>
<tr>
<td>5. if (previousFailures.isEmpty(flammFailure)) then</td>
</tr>
<tr>
<td>6. evchild.type ← Basic</td>
</tr>
<tr>
<td>7. emftaFTAModel.events ← emftaFTAModel.events \cup evchild</td>
</tr>
<tr>
<td>8. emftaEvent.gate.events ← emftaEvent.gate.events \cup evchild</td>
</tr>
<tr>
<td>9. else</td>
</tr>
<tr>
<td>10. evchild.type ← Intermediate</td>
</tr>
<tr>
<td>11. evchild.gate.type ← OR</td>
</tr>
<tr>
<td>12. emftaFTAModel.events ← emftaFTAModel.events \cup evchild</td>
</tr>
<tr>
<td>13. emftaEvent.gate.events ← emftaEvent.gate.events \cup evchild</td>
</tr>
<tr>
<td>14. for all cp ∈ getConnectedPorts(flammPort) do</td>
</tr>
<tr>
<td>15. for all cf ∈ getFailures(cp) do</td>
</tr>
<tr>
<td>16. if (getId(cf) = getId(flammFailure)) then } then</td>
</tr>
</tbody>
</table>
Case 3: InputPort of a composite component

Lastly, Case 3 represents the final condition for this FTPopulation method, which is reached whenever the received port is an inputPort of a composite component (line 2 in Table 10). This port represents the boundary of the modelled system in our approach, so no further decomposition of the failures can be made. Due to this, the created events do not have an associated gate since they are always the root cause, and the Event Type is set as Basic (line 5 in Table 10). Next, the FT is completed, and the algorithm is repeated for the remaining top-events created in step 1. Table 10 presents the algorithm developed for Case 3.

Table 10: Algorithm 3 for FTPopulation. Case 3.

<table>
<thead>
<tr>
<th>Algorithm 3 FTPopulation(flammPort,flammFailure,emftaFTAModel,emftaEvent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. procedure: FTPopulation(flammPort,flammFailure,emftaFTAModel,emftaEvent)</td>
</tr>
<tr>
<td>2. if (isInputPort(flammPort) and belongToComposite(flammPort)) then</td>
</tr>
<tr>
<td>3. if (id(flammFailure) ≠ &quot;noFailure&quot;) then</td>
</tr>
<tr>
<td>4. evchild ← create new event with proper name</td>
</tr>
<tr>
<td>5. evchild.type ← Basic</td>
</tr>
<tr>
<td>6. emftaFTAModel.events ← emftaFTAModel.events ∪ evchild</td>
</tr>
<tr>
<td>7. emftaEvent.gate.events ← emftaEvent.gate.events ∪ evchild</td>
</tr>
<tr>
<td>8. end if</td>
</tr>
<tr>
<td>9. end if</td>
</tr>
<tr>
<td>10. end procedure</td>
</tr>
</tbody>
</table>

ETL implementation

Once the structure that the algorithm needs to follow to generate FTs automatically is presented, this is implemented in a specific M2M transformation language. As it was explained and justified in Section 2.2.5, the selected language is Epsilon Transformation Language (ETL). In our solution, two models are involved in this transformation, which are:

- **Source model**: Referred to in the transformation code as "flamm" and based on Concerto-FLA meta-model.
- **Target model**: Referred to in the transformation code as "emfta" and based on EMFTA meta-model.

In order to realise this mapping and implement the algorithms mentioned above using Epsilon Transformation Language, one rule and one operation are developed. We start with the composite component, which is composed of other simple components. For each output port of the composite component and each failure on that specific port, a different fault tree is generated.

One notable feature is that all failures of type noFailure need to be removed since this is a possibility presented in Concerto-FLA when a component behaviour is to sink the received failure. To do so, before making any transformation, the pre-statement provided by ETL is used as explained in Section 2.2.5. Fig. 22 presents how all FTAModels are created according to the previous condition. This part of the code is the implementation of step 1 explained in the algorithm design sub-section.

As Fig. 22 shows, a variable called mapping (lines 3 and 8 in Fig. 22) is used to store all the ports contained in the source model. Then, the ports which are owned by a composite component and which are of type outputPorts are selected (line 11 in Fig. 22). For these ports, an FTAModel is created in the target model for each failure they contain. The name of this FTAModel is derived using the id of the selected failure, the name of the port and the name of the owning component (line 15 of Fig. 22). Lastly, this FTAModel is added to a variable called ftas, which stores all created FTAModels, so they can be used later when the transformation process is carried out (line 16 of Fig. 22). This implementation corresponds to Algorithm 1 as explained in the previous sub-section.
Next, it is necessary to identify and create the top-event for each FTAModel. To do so, Algorithm 2 is implemented in the form of an ETL rule. These rules, as explained in Section 2.2.5, transform one element of the source model into one (or many) elements of the target model. Even though we need to transform failures into events, these elements do not have enough information to create an event by itself. Due to this, the Port element from the source model (which contains information about the Failures) is used to derive events.

As Algorithm 2 proposed, the top-event of each FTAModel is defined as the failure which appears in the outputPort of the composite component. To limit the applicability of this rule to those specific type of ports, ETL provides a feature named guards. These guards are defined by [39] as an element “which limits the applicability of a rule to a narrower subset of instances of its specified type”. The implementation of the guard functionality is provided in line 5 of Fig. 23, where the subset of ports is limited to those of outputPort type and which are owned by a composite component.

Next, the following step is to create as many events as failures included in the analysed port. In ETL every time a rule is triggered, it generates an entity of the target metamodel. To avoid the generation of duplicate entries, we used conditions to generate the new events when they are corresponding to a unique port failure combination. This has been implemented by using a Boolean variable named firstelement, which determines if the event for the analysed port has been already created. If the value of this variable is true (line 11 of Fig. 23), the default event in the heading of the rule is used to generate the event, since no event for that specific port has been previously created. Then, the value of the firstelement variable is set to false to inform the next iteration (line 19 in Fig. 23). The remaining failures associated with the Port are used to create new events, different from the default one.

Once the top-event is created, and the type has been defined as Intermediate, the gate is derived (lines 16-17 in Fig. 23). Then, this event is added to the owner FTAModel, as a part of its events attribute. This FT is identified through the name since this attribute contains the analysed port, failure id, and owner component of the port (line 10 in Fig. 23). Finally, the rule invokes the ETL operation createRoot() (line 24 in Fig. 23) that populates the FTAModels. Fig. 23 presents the implementation of this ETL rule when the firstelement variable is true.

The createRoot() operation (which corresponds to the previously presented FTPopulate algorithm) is divided into the three cases presented in Algorithm 3 design sub-section, which are implemented in ETL using three if-else statements. As it was previously mentioned, the algorithm is invoked by the transformation rule and receives four variables: the analysed Port, the associated Failure, the parent Event and the owning FTAModel. To determine the type of the received port, the functionalities isDefined() and includes(), provided by ETL, are used. If the parent component of the owner port is defined, then the component is of simple type. On the other hand, if the component parent is not defined, the component is a composite one. The includes() statement works similarly, that is, if the outputPorts attribute of the owning component of the port includes the value of the port, then this is of output type, and in any other case, it is of input type. When the results provided by both statements are combined, we can determine to which of the three proposed cases, the port belongs (line 6 in Fig. 24). Fig. 24 shows the ETL implementation for the Case 1 of Algorithm 3.
Case 1 presents the situation where the port is of type output and owned by a simple component. Firstly, for each failure in the analysed port, a new event is created (line 9 in Fig. 24). To determine the event type, the code checks if previousFailures for that specific failure exist (line 11 in Fig. 24). If this is the situation, then the event type is set to Intermediate, but if there are no previousFailures defined, then the type is set to Basic. The latter case does not require a gate, since it is the root cause, and then the next failure is analysed. On the other hand, the Intermediate type needs a gate to be defined, and the type of such gate is determined based on the guidelines provided by the mapping done at the beginning of this section. Finally, the following port to be analysed is defined. In case 1 this is done through the rules attribute (line 39 in Fig. 24). For each rule that contains the port and failure under study as outputExpressions, each port appearing in the inputExpressions of the same rule are the connected ones. Therefore, the createRoot() operation is invoked iteratively for these parameters (line 45 in Fig. 24).

Once this operation reaches its final condition and creates all the Basic events, the FTAModel is populated. If all the candidate ports have been transformed, the transformation process ends.
6.4.2. Eclipse plugin integrating the transformation

Once the ETL transformation derives FTs from Concerto-FLA results, this functionality needs to be generated automatically. Since this work will be available as part of CHESS toolset, the solution is implemented as an Eclipse plugin. To develop this plugin, the implementation process is divided into three steps, which are needed to deploy our solution adequately. These are the following:

- Plugin appearance and architecture
- Integration of ETL transformation with plugin
- Fault tree visualisation and editor

The implementation of each of these three features is explained below and provides us with the information needed to understand how the plugin is developed to reach our final goal.

**Plugin appearance and architecture**

Since Eclipse defines several ways in which a plugin can contribute, we need to decide which one is better for our approach. Once all the provided templates have been analysed, we conclude that the best one for our solution is the menu contribution bar, since most of the CHESS functionalities are added using this plugin style. This template creates a new button in the top menu of Eclipse which, when clicked, triggers the transformation process. Since it is implemented in the Plugin Development Environment (PDE) as a template, the creation is a simple process which relies on following the guidelines proposed by the PDE assistance. Once this is created, the template provides an empty Java class which is executed when an event from the user is received. This event represents the click action over the menu bar.

The name of the bar that includes our approach is set to “Dependability Analysis”, which contains a submenu under the name “Generate Fault Trees”. Once this submenu button is clicked, the plugin is invoked. Fig. 25 below shows the final appearance of the plugin.

![Fig. 25: Plugin appearance.](image)

The first feature that the plugin needs to implement is to allow the user to select the Concerto-FLA result file from which he/she needs to derive FTs. To enable this, we use the JFileChooser class provided by Java, which allows to select a file or folder and store its content so that this can be used or invoked later. Figure 26 shows how this implementation is integrated into our solution.

```java
FileChooser chooser = new JFileChooser();
chooser.setCurrentDirectory(folder);
chooser.setDialogTitle("Please select your FLA results file");
chooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
chooser.setAcceptAllFileFilterUsed(false);
if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
    System.out.println("getCurrentDirectory() : " + chooser.getCurrentDirectory());
    System.out.println("getSelectedFile() : " + chooser.getSelectedFile());
}
```

**Fig. 26: Selection of source model implementation.**

Some critical parameters related to the appearance, need to be configured in the JFileChooser, such as the title of the dialogue or the default folder in which it is opened. Even though the user can navigate through the folders to select the .flamm file (containing Concerto-FLA results), the default route (named folder in the Fig. 26 code) is set to be the location of the current workspace used by the user. Once all the options are configured, the user only needs to select the file to be transformed and press the Open button. Then, the transformation process begins, using as source model the file selected by the user. The appearance of this file selector is shown in Fig. 27 below.
Integration of ETL transformation with plugin

After this, no further information from the user is required, since the next step is to configure the transformation parameters automatically. To do so, the guidelines proposed by Epsilon in the examples provided in their git repository [44] are followed. From these, four steps that are needed to configure the transformation are extracted and explained in detail below along with some code snippets.

1. Creation of an ETLModule and reading transformation file: Firstly, an ETL module needs to be created. This ETL module represents a container that stores all the context and parameters the transformation uses. Once this ETL module is configured programmatically, it can be executed to provide us with the transformed model (if the configuration process has been accomplished).

Epsilon proposes that the first element that needs to be added to the context of the module is the .etl file that contains the transformation rules (presented in Section 6.4.1). To do so, the location of this transformation file needs to be provided so that the module can parse (read) the content. The code needed to perform this first step is presented in Fig. 28 below.

```java
"We need to create an EtlModule in which all the parameters of the transformation will be stored."
ETLModule module = new ETLModule();

"First, we need to parse the file that contains the transformation rules and add this to our EtlModule"
try {
    module.parse(new File("ftaGenerator/epsilon/FLA2FTA.etl"));
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

2. Creation of source and target models: The next step is to create the EMFModels (Java class that creates the models which ETL works with) that the transformation uses, which in our case are both the source and target models. In order to create them, some key features, which are explained below, need to be defined:

- **Name**: String parameter which stores the name of the model file. This value must be the same one used in the ETL file that contains the transformation rules.
- **readOnLoad**: Boolean parameter that specifies to the ETL module if the model information has to be read before performing the transformation. This parameter is set to true when the model is the source and false when it is the target.
- **storeOnDisposal**: Boolean parameter that specifies to the ETL module if the model information needs to be updated after performing the transformation. This parameter is set to true when the model is the target one and false when it is the source model.

- **Meta-model**: String parameter that contains the location of the meta-model, on which the created model is going to be based.

- **Model**: String parameter that contains the location of the created/used model file. If this file does not exist, then the module creates it in the selected location if the storeOnDisposal value is true.

Since the configuration process of this parameters is the same for the source and target model, a Java private method called `setEmfModel()` is implemented. This method performs the previously explained configuration process for any given model. Fig. 29 shows how the implementation of the `setEmfModel()` method.

```java
private EmfModel setEmfModel(String name, Boolean readOnLoad, Boolean storeOnDisposal, String metamodel, String model)
{
    EmfModel.emfModel = new EmfModel();
    EmfModel.setName(name);
    EmfModel.setModelFile(model);
    EmfModel.setMetaModelFile(metamodel);
    EmfModel.setReadOnLoad(readOnLoad);
    EmfModel.setstoreOnDisposal(storeOnDisposal);
    EmfModel.setCachingEnabled(true);
    EmfModel.setExpand(true);
    try {
        EmfModel.load();
    } catch (EOFError e) {
        e.printStackTrace()
    }
    return EmfModel;
}
```

**Fig. 29**: EMFModels configuration Java method.

3. Add the configuration parameter to the module and execution: At this point, the transformation rules (which have been parsed in the first step) and the created models need to be included in the ETL module. To do so, these elements are added to the context of the module as Fig. 30 shows. Then, all the required information has been provided, and the module can be executed.

```java
"finally, we add to the EtlModule all the rules and created models so it can be executed."/
module.declaredPre();
module.getTransformationRules();
module.declaredPost();
module.getContext().getModelStateRepository().addModel(flammModel);
module.getContext().getModelStateRepository().addModel(emfaModel);
try {
    result = module.execute();
} catch (EtlRuntimeError e) {
    e.printStackTrace()
}
```

**Fig. 30**: Module configuration and execution.

4. Clean the module: Finally, this step is done for the sake of future transformations. The ETL module and the models are cleaned, so no information is stored for future transformation processes. This is implemented by using the `dispose()` command, which removes all the information contained in the created EMFModels and is shown in Fig. 31 below.

```java
"To finish, we clean both the models and the EtlModule for future transformations."/
flammModel.dispose();
emfaModel.dispose();
module.getContext().getModelStateRepository().dispose();
return pathReturn;
```

**Fig. 31**: Clear of module and models.
Once all the previous steps are implemented, the plugin generates a model based on EMFTA meta-model which contains all the FTs derived from the source model. All the previous code has been implemented in the proposed plugin as a separate Java class called Transformation, which is invoked once the user has selected the source model file. This class returns an IPath object, which contains the location of the generated EMFTA model file.

**Fault tree visualisation and editor**

After generating the target model, in which the FTAModels are created, we integrate the results with a visualizer that transforms the given model to FTA representation format. To do so, and as explained in Section 6.2.2, the selected EMFTA tool includes both the target meta-model and an open source tool that allows the user to visualise the FTAModels. This tool is also implemented as an Eclipse plugin and can be used as a standalone application. The source code needed to implement it can be downloaded from their Github repository [42]. This code includes several plugins related to the EMFTA tool, but the one of significant interest for our solution is called eu.fbk.standardtools.faultTreeViewer. In this plugin, several commands allow the user to create an FT from a file. However, since our approach does not require all of them, we are focusing on the FaultTreeViewerUtil class, that creates and opens the FT representation for a given .emfta file. Our solution is creating a FTAModel for each failure in every output port of the composite component. Therefore, this class creates a separate FT representation for each derived FTAModel, allowing the user to study and focus on individual failures in isolation.

Some changes are implemented in this class to adapt it to the needs of our solution. For this, a new class is created as part of our plugin under the name FaultTreeViewerUtil. In this class, we have implemented the code provided by [42], including some changes for the following methods:

**openFTAViewerFromEmftaFile()**

The original version of this code received a Java Selection made by the user, which included the selected file that needs to be transformed. Since this functionality is provided differently by our solution, we have set that the received parameter is now a Java File which contains the information of the model previously selected by the user. Since the original Selection derived a File by accessing the firstElement contained in this selection, the rest of the method is used similarly.

This method creates the representation from the .emfta file and opens this diagram so the user can visualise it. The changes introduced in this method are shown in Fig. 32. The commented parts of the code (lines 3 and 4) represent the original implementation.

```java
//Changed method
1. public void openFTAViewerFromEmftaFile(IFile file, IProgressMonitor monitor) throws Exception {
2.     //if (selection.getFirstElement() instanceof IFile) {
3.         IFile file = (IFile) selection.getFirstElement();
4.     }
5.     if (file.getFileExtension().compareTo(".emfta") == 0) {
6.         IProject project = file.getProject();
7.         String name = file.getName().replaceAll(".emfta", "");
8.         Session session = createAirdSessionFromCurrentProject2(project, monitor, name);
9.         session.open();
10.         String pathName = file.getAbsolutePath().toOSString();
11.         URI ftaResourceFileURI = URI.createPlatformResourceURI(pathName, true);
12.         FTAModel ftaModel = createFTAModel(ftaResourceFileURI);
13.         Resource emftaResource = openExistingFTAResourceFile(ftaResourceFileURI, ftaModel);
14.         openFTAViewer(emftaResource, session, ftaModel, monitor);
15.     } else {
16.         throw new Exception("No .emfta file selected");
17.     }/
```

**createAirdSessionFromCurrentProject2()**

Even though this method already existed in the original code, we needed to implement some additional functionality in our solution. In order to store the results systematically, a new folder in the user’s workspace is created, to store both the .emfta file and the representations file. To create the representation file (.aird file) in this specific location, the location that this method receives needs to be changed. The original code only created one representation file, and if another one existed in the same location, this was removed and updated. In order to allow the user to derive FTs for different Concerto-FLA result models, each transformed model has its own representation. In this way, the user can perform and store several transformations from several modelled systems.
without losing any information. Fig. 33 shows how this functionality has been implemented in our plugin through a Java private method. This method is invoked in line 9 of Fig. 32, when the representation is created.

```java
// New method
private ISession createAirSessionFromCurrentProject2(IProject project, IProgressMonitor monitor, String name) throws Exception {
    IFile airFile = project.getFile(name + "\representations\" + name + ".aird");
    return getAirSessionFromAirFile(project, airFile, monitor);
}
```

Fig. 33: Introduced changes in visualizer class (2).

Once all the above mentioned changes have been implemented in the visualizer class, the final step is to invoke this new class for the generated .emfta. To do so, a Java IFile object is created by using the IPath the transformation class returns (line 3 in Fig. 34). This IFile represents the created .emfta file and is sent to the openFTAViewerFromFile() method (line 5 in Fig. 34). The implementation of this functionality is shown in Fig. 34 below. With this implementation, our solution is complete and ready to be tested in a real case scenario.

```java
1 // If the file extension is the proper one, we create the representation.
if (target.getFileExtension().equals("emfta")){
  2 IPath path2 = target.getFullPath();
  3 IFile file = (IFile) ResourcesPlugin.getWorkspace().getRoot().getName(path2);
  4 try {
    5        FaultTreeViewerUtil.getInstance().openFTAViewerFromFile(file, new NullProgressMonitor());
    6    pmonitor.done();
    7        progress.close();
    8   } catch (Exception e) {
    9       // TODO Auto-generated catch block
    10      e.printStackTrace();
  11 }
```

Fig. 34: Plugin visualizer implementation.
7. Case Study

This chapter presents the illustration of the proposed solution when it is applied to a real modelled system. As stated in Chapter 3, this thesis is based on the work provided by [46], so the selected case study is focused on the same domain. This chapter is organised as follows: Section 7.1 presents Attitude Controller System (ACS) as the studied system and provides the knowledge needed to model ACS. Section 7.2 introduces ECSS-Q-ST-30C and ECSS-Q-ST-40-12C as the standards that require performing FTA to assess dependability and safety in the aerospace domain. Section 7.3 presents the applicability of our approach in two different scenarios. Finally, Section 7.4 discusses the obtained results, and which are the benefits our approach presents.

7.1. Attitude Control System (ACS)

Most current space missions involve the launch of unmanned satellites, which are used for several purposes such as telecommunications, weather or space exploration. To control the orientation of the satellite ACS is used. In this context, ACS is defined, according to Noteborn et al. [47] as “a suite of components that in close interaction with the rest of the spacecraft, controls the orientation and stability of the spacecraft, and executes the grounds requested velocity changes for adjustment of the otherwise ballistic trajectory”. That is, ACS controls the attitude of the spacecraft, either by performing its own calculations or by applying received orders from the ground station. Since this approach only considers software systems, we have focused on ACS from a software perspective. In this situation, ACS is defined as a software unit that receives data from sensors and calculates the target values to be applied to the satellite actuators, so the desired attitude is maintained. Therefore, this software unit is in turn composed of software functions used to perform specific tasks. According to [47], the main functions present in an ACS are listed below. In addition, Fig. 35 shows how these functions interact with each other.

- **Signal Conditioner**: This software function takes the value measured by the sensors and transforms it into satellite reference frame values. Thereby, the satellite can interpret and use these values since the position of the sensors is always the same from its own reference frame. Then, these values are used by the signal conditioner to calculate the satellite orientation from the Sun.

- **State Estimator**: This software function receives the conditioned values from the Signal Conditioner and estimates the current location of the satellite, as well as the angular velocity. To do so, it compares the data of previous iterations with the information sent by the Signal Conditioner, and if there are any discrepancies between them, this function computes the estimated values. If there are no records of previous iterations, these values are considered as zero. In addition, from the conditioned data received from the gyroscope, the State Estimator calculates the disturbance torque generated from cross-coupling, and sends this value to the Control, so it can derive a torque to compensate this.

- **Control**: This software function takes the state of the satellite in the form of the values derived by the State Estimator. Then, it calculates the difference between the current state and the target state (the final orientation of the satellite needed for the mission or other operations). Based on that difference it calculates the torque which shall be applied on the satellite using thruster actuator. According to [46], the Control function of an ACS is composed of the following four software components:
  - **PD Controller**: this component receives the Sun direction and angular velocity values estimated by the State Estimator. Then, it computes the pointing error detected from these values and calculates the proportional torque to be applied.
  - **Steer Controller**: this component calculates the proportional torque to be applied. However, the torque applied by this one is greater to get a faster convergence of the target location.
  - **Feedforward Controller**: this component computes the compensating torque for cross-coupling “to achieve the equilibrium state and stabilize the satellite body” [46].
  - **Torque Selector**: this software component receives all the values calculated by the above-mentioned controllers and based on the current position (sun direction), it decides which value is applied to the actuators. For example, “if the satellite is oriented 180 degrees opposite to the desired pointing direction, a rapid change in attitude is required. Hence, relatively greater torques, which are computed by SteerController, are applied. On the other hand, when the satellite is closer to the desired pointing direction, rather a seamless convergence and stabilization is of interest. Therefore, smaller torques and stabilization torques, computed by PDController and FeedforwController respectively, are applied to satellite” [46].

---

47
ACS can have different operational modes, which are depending upon the reference frame relative to which the orientation of satellite needs to be maintained. Typically, the object of interest to which the satellite should point is motivated via the satellite mission. For example, weather satellite needs to point towards the earth to record the weather data. Additionally, the modes are also designed to fulfil the critical needs of operations. For example, in Sun Acquisition and Survival Mode (SASM), the satellite needs to point towards the Sun to charge its critical components. Therefore, depending on the purpose of the satellite mission and the operational mode used only some physical devices, such as sensors or actuators, are used at the same time. In Table 1 of [47], the operational modes of ACS are presented and described.

To demonstrate the proposed methodology, in this case study, we model ACS in one of the operational modes presented in [47], specifically SASM. This mode is defined in [60] as “ultimate backup mode ensuring spacecraft survival in case of major onboard contingencies”. As previously mentioned, in this operational mode the spacecraft is pointing towards the Sun. According to [47], to compute the attitude in this mode, the ACS uses the values provided by the Sun sensor and the gyroscope to calculate the needed torques, which are later applied to the thrusters (i.e., the actuators).

7.2. ECSS-compliant FT

As stated in Section 5.1, our approach generates FTs, compliant with IEC-61025 standard, from a modelled system. Since the proposed case study domain is the aerospace one, the solution is focused on the standards related to this context, i.e. the ECSS standards series. These standards deal with the aerospace domain, not only in the engineering domain but also in the product assurance and process management. Fig. 36 presents all the documents provided by ECSS according to the domain they apply to.

From the list of standards provided in Fig. 36, ECSS-Q-ST-30C and ECSS-Q-ST-40C are those focused on dependability and safety respectively. While the former lists all the methods needed to develop dependability analysis at all levels of the system, the latter presents the required methods related to safety analysis. Specifically, the guidelines for performing ECSS-compliant FTA are provided in ECSS-Q-ST-40-12C [51]. The procedural guidelines of this standard for deriving FTs reference to two external documents, [19] and [50], which are used to implement our solution. In particular, IEC-61025 [50] is the standard which needs to be followed, specifically Chapter 7, which discusses FT development and evaluation. Additionally, [19] is used as a reference guide since it is not mandatory. In [50], generic guidelines for the development of FTs were presented. Since our solution is compliant with this standard, we list below the guidelines that are implemented in our approach:

- Fault Tree development must always start with the definition of a "top-event, or unfavourable outcome which should be defined unambiguously" [50]. This statement is strictly followed by our approach since the first created event is the top-event. This one is identified through the failure that causes it, the port in which it appears and the owner component of this port.

- “Fault trees may be drawn either vertically or horizontally. If the vertical arrangement is used, the top event should be at the top of the page and the basic events at the bottom.” [50]. As the examples provided in Section 6.2.2 show, the visualizer used presents the trees in a vertical way, with top-event on the top and basic events on the bottom.
According to [50], “if an event represents a repeated event or common cause event, it is shown in the fault tree repeatedly”. This is implemented in our solution since events are created independently of each other.

“Strict adherence to the concept of immediate cause is necessary to ensure that failure modes are not omitted” [50]. Since the results provided by Concerto-FLA take into account all the ports of all the components in the system, no failures are omitted, so the immediate cause is fulfilled.

“The construction process down the tree, transferring attention from mechanism to mode, a continually approaching a lower level” [50]. As explained in the algorithm design section, this is accomplished by analysing the ports from the output ones to the input ones.

By fulfilling all the previous guidelines, we can conclude that the proposed approach is ECSS-compliant, which is one of the primary goals of this thesis.

### 7.3. Failure logic analysis of ACS and fault tree generation

In order to perform this case study, the Concerto-FLA methodology presented in Section 2.5.3 is expanded to include our solution. This methodology, which also integrates the three-step process proposed in Section 6.3, is represented in Fig. 37.

This methodology proposes an iterative process in which, once the FTs are generated, the results need to be analysed to determine whether they meet with the system requirements. If the results do not satisfy the requirements, a re-design of the system is necessary, and the process is repeated. Based on this loop, we formulate two scenarios to apply the above-mentioned methodology. In Scenario 1, we present the preliminary design of the system under study (i.e., ACS in SASM), which refers to the design having no dependability means. Based on the generated results, we propose dependability means to improve the design, and that is presented in Scenario 2, referred to as Improved design.

The results obtained from the application of the three-steps methodology presented in Section 6.3 for both scenarios are presented below.
Scenario 1: Preliminary Design

Step 1: Modelling the system

The first step is to model the ACS in CHESS ML. To do so, the definition of the composite component and its simple components along with the involved ports are specified.

According to Section 7.1, ACS is composed of three main software functions: Signal Conditioner, State Estimator and Control. In [46] the last function has been modelled and analysed. However, as previously stated, the span of this case study also includes two other software functions which process the sensors data and estimate the state of the system. Therefore, the ACS composite component is composed of the six components mentioned in Section 7.1.

Regarding the ports, we need to identify which are the input and output ports of the composite component, and from these, follow the data flow inside ACS to determine the ports for each simple component. As stated in Section 7.1, the ACS operating in SASM receives two different values from two different sensors, i.e., sun direction and gyroscopic data. Hence, the composite component needs two input ports. The values provided in these ports are sent to the Signal Conditioner, which adapts them to the reference frame of the satellite. Therefore, this component needs two input ports and two output ports. Then, each conditioned value is transmitted to the State Estimator, which estimates the current position of the satellite and computes the estimated values of the angular velocity and the existing cross coupling disturbances. To perform these tasks, the State Estimator needs two input ports and three output ports. Next, this component sends these values to the three controllers inside the Control unit and to Torque Selector component. However, since each controller computes different torques, they do not receive the same information. In the case of the PD Controller, we previously mentioned that it calculated the proportional torque using both the estimated position of the satellite and the angular velocity. Then, this component needs two input ports and one output port. Both the Steer Controller and Feedforward Controller compute the proportional torque from a single value, which is the estimated position of the spacecraft and cross-coupling disturbances respectively. Consequently, these two controllers need one input port and one output port. Next, the three previous controllers send the computed values to the Torque Selector, which decides which is applied depending on the estimated position. To do so, this component needs four input ports and one output port. Finally, the selected value is sent to the thrusters, but the torques computed by the Feedforward and PD Controller are also stored for future iterations. This means that the ACS composite component needs three output ports. These interactions and dataflows are illustrated through a block diagram in Fig. 38 for the sake of understanding.

After deriving the information related to the components and the ports, the ACS is modelled using CHESS ML, resulting in the model presented in Fig. 39.
Step 2: Defining the failure behaviour for the components and failure logic analysis

As previously mentioned, this first scenario represents the preliminary design of the system. At these stages, the implementation of components does not have any fault tolerance means. Therefore, the behaviour of the components, from the four possible behaviours defined in Section 2.5.3, is propagator.

In Concerto-FLA three failure modes (presented in Section 2.5.1) can be specified for the components to define their failure behaviour. In this case study, we only focus on the failure mode of type value to demonstrate and evaluate the proposed methodology. Usage of all the failure modes in this scope is neither needed nor desirable. To specify the value failure mode, both of its specialisations (i.e., value subtle and value coarse) are used for the sake of completeness. The former represents the value failure which is not detectable (even when a failure detection mechanism is applied), while the latter represents a value failure that is detectable. However, the focus in this work is not on detection and rather on analysis.

In order to model the failure behaviour of the components, FPTC rules are defined following the syntax presented in Section 2.5.3. Since the behaviour of all the components is propagator, these rules have the same failure value in both the input expression and the output expression. Fig. 40 shows an example of the implemented rules for the Feedforward Controller in CHESS FLA editor, where both the value subtle and value coarse failure in the input port are transmitted to its output port.

Fig. 38: ACS data-flow.

Fig. 39: ACS system modeled in CHESS ML.
Next, the input ports of the composite component are injected with the proposed failure modes, i.e., value subtle and value coarse. Then, Concerto-FLA is executed, generating in the output ports of the composite component FPTC specifications, which provide the computed failures for such ports. As expected, since all the components behaviour is set to propagator, the three generated FPTC specifications shows both the value subtle and value coarse failures (which were injected). Fig. 41 shows how these FPTC specifications are presented in the modelled system once the analysis is performed. Additionally, Fig. 42 presents the files generated by Concerto-FLA, containing the failure propagation paths. From these two files, the .flamm is used to automatically generate FTs in the next step.
**Step 3: Automatic FT generation**

Once the system and the failure behaviour are modelled, and Concerto-FLA is executed through Steps 1 and 2, the next step is to generate automatically the FTs. To do so, the .flamm file obtained in the previous step needs to be imported into the CHESS toolset workspace. Then, the plugin is invoked, and the process described in Section 6.4.2 starts. By selecting the .flamm as the source model, the plugin performs the transformation process and generates the results into a new folder. Inside this folder, two files are created: the result of the transformation (model) and the representations folder, which contains all the created FTs representations. For the sake of understanding, a user manual is derived in Appendix C, in which all the steps that the user needs to follow, from the selection of the source file until the results visualisation, are explained in detail.

As stated in the first step, the modelled system for Scenario 1 contains three output ports, and all components are modelled to propagate every received failure. Since the injected failures are two (value subtle and value coarse), our algorithm generates six different FTAModels, as well as six FTs representations. These figures are compliant with the guidelines proposed in Section 6.4. Even though all the generated FTAModels for the Scenario 1 are included into the Appendix A, only one of them is analysed and presented in this section, which is titled as “value coarse of ctrlTorque in ACSComposite”. We have decided to focus the analysis of the results in this port since it is responsible for sending the control command to the actuators.

Due to space reasons, the FT representation obtained for the selected FTAModel is divided into two figures (Fig. 43 and Fig. 44), which represent the top section and the bottom section of the representation respectively. The former presents the derived top-event for this FT, which corresponds to the value coarse failure in the composite component output port (i.e., ctrlTorque). This top-event is created by a previous value coarse failure in the ctrlTorque port of the Torque Selector component. This failure appears when at least one of the input ports of the component presents the value coarse failure. Therefore, the event related to the failure of the output port in Torque Selector is connected to the four input ports events through an OR gate, as Fig. 43 shows.

On the other hand, the bottom section of this FTAModel presents the four failure paths (branches) for each of the Torque Selector inputs. Each path is derived following the connections modelled for each port and finishes in a basic event. Five basic events have been derived by the transformation, representing the value coarse failures that were injected into the input ports of the composite component. Although there are only two input ports in the composite component, the port estSunVec has three different connections in the modelled ACS. Therefore, the root cause of this failure (sunVec failure) appears three times in the generated FTAModel (red circles in Fig. 44). This proves that the fourth clause provided in Section 7.2 (common cause failure) is implemented in our solution. Fig. 44 presents the results for the bottom section of this FT, including the information previously explained. Both Fig. 43 and 44 follow the system data-flow, presented in Fig. 38, from the system’s outputs to its inputs.

---

**Fig. 43: Scenario 1 result top section of FTAModel.**
Fig. 44: Scenario 1 result bottom section of FTAModel
Scenario 2: Improved Design

From the results obtained in Scenario 1, the dependability of the system can be improved by deriving some countermeasures. To do so, we have assumed that, after generating the results of the Scenario 1, the designer of the system has decided to implement failure detection mechanisms in some of the software components. As stated previously, these mechanisms can be used to remove the value coarse failures from these components, but value subtle failures will persist since cannot be detected.

Two of the simple components (i.e., PD Controller and Feedforward Controller) are adapted to detect and prevent such failures. Both include a low pass filter that detects value coarse failures on the angVelocity port of the PD Controller and value coarse failure in the gyroscope torque port of Feedforward Controller. Therefore, these two components sink these failures, which should be removed from the resulting FTs, as well as all the dependent failures to these. In order to perform the analysis, the same steps used in the Scenario 1 are followed.

**Step 1: Modelling the system**

As stated at the beginning of this section, this case study is focused on addressing the system architecture in early design phase. Therefore, the proposed changes, related to dependability means, will be applied to the system in the following phases, such as implementation. Consequently, the system is not modified at this stage and Fig. 39 still represents the ACS system model used in the Scenario 2.

**Step 2: Defining the failure behaviour for the components and failure logic analysis**

As previously stated, the changes introduced do not affect to the modelled system. Nevertheless, these new functionalities modify the failure behaviour of the updated components. Therefore, the rules defined for PD Controller and Feedforward Controller components need to be modified according to the proposed changes.

In the case of the PD Controller, the value coarse failure in the estAngVelocity port is now sunk by the component since fault tolerance means were introduced. Therefore, the output port of this component, i.e., pdTorque, does not generate this value coarse failure. This behaviour is also implemented in the Feedforward Controller since the same means were introduced. The feedforwardTorque port does not generate anymore the value coarse failure. To implement both updated failure behaviour in CHESS, the FPTC rules defined for Scenario 1 are modified in the editor for each component. Fig. 45 and 46 show the updated FPTC rules for the PD Controller and the Feedforward Controller respectively.

After updating the failure behaviour for the proposed components, Concerto-FLA is executed, and the model is annotated with the generated FPTC specifications. We can observe that the specification for feedforwardTorque port (red circle in Fig. 47) has removed the value coarse failure, due to the introduced dependability means for the Feedforward Controller. In addition, a new .flamm file (as in Fig. 42) is generated with the updated results.

---

**Fig. 45: Rule for PD Controller.**

**Fig. 46: Rule for FeedforwardController.**

**Fig. 47: Scenario 2 modelled system annotated with Concerto-FLA results.**
**Step 3: Automatic FT generation**

As it was done for Scenario 1, the plugin is invoked, and the .flamm file (generated in step 2) is selected as the source file. Then, the steps proposed in Appendix C are followed, and the FTs related to the improved design are generated. In this case, the number of generated FTs is reduced to five, which is due to the implementation of the new rule for the Feedforward Controller. As Fig. 47 shows, the value coarse failure of the feedforwardTorque port of the ACS is removed, so the related FTAModel, and thus its FT representation, is no longer generated by the solution. Therefore, we can state the implementation of the dependability means reduces the number of generated FTs and consequently increases the dependability of the system.

To compare the generated results with the representations obtained from Scenario 1, the same FTAModel (value coarse of ctrlTorque in ACSComposite) is analysed in this section, while the other four FT representations are included in the Appendix B. By looking at the results of the top section, presented in Fig. 48, we find a substantial difference compared to representations obtained from the previous scenario. Now, the events causing the value coarse failure of the ctrlTorque are three instead of the four derived from Scenario 1 (Fig. 43). The updated FPTC rule for Feedforward Controller removes the occurrence of the value coarse failure in the feedforwardTorque port of the Torque Selector. Thereby, all its related failures are removed from the system and the possibility of the top-event occurrence, i.e., the system failure, is reduced.

Finally, the results for the bottom section of this FTAModel, presented in Fig. 49, show that not only the branch related to the Feedforward Controller is removed, but also all the previous events to the value coarse failure of the angVelocity port of the PD Controller. These events are removed from the representation due to the updated rule for PD Controller, which sinks the value coarse failure in the estAngVelocity port. Therefore, according to Section 2.6.3, the dependability of the improved design is increased, since the number of basic events in the derived FT is reduced to three. This generates a minimal cut set smaller than the one presented in Scenario 1.

Fig. 48: Scenario 2 results top section FTAModel.
Fig. 49: Scenario 2 results bottom section FTAModel.
7.4. Discussions

In this chapter, the proposed solution has been applied in a real case in the context of the aerospace domain. This case study covers two scenarios to illustrate the applicability of the proposed solution. To design them, the software development process for safety-critical systems has been followed, using Concerto-FLA extended methodology proposed in Fig. 37.

Scenario 1 presents a preliminary design of the system, resulting from the early stages of software design. After modelling the system and applying the proposed solution, the obtained results showed that the system was not safe enough to operate in its intended environment, and thus it needed to be re-designed. This statement is derived from the fact that the occurrence of each event leads to the top-event of the FT (i.e., system failure). To improve the dependability of the modelled system, some dependability means are proposed to reduce the number of single points of failures.

Scenario 2 presents an improved design of Scenario 1, which implements the proposed means to remove some of the existing failures. To do so, the failure behaviour of the affected components is updated through new FPTC rules. The generated results show that both the size and number of the FTs is reduced. This implies that, although the improved design is not safe enough (all events are single points of failure), the related dependability has been increased compared to the preliminary design proposed in Scenario 1.

Through this application, it is shown that the fault trees generated with our solution adhere to the guidelines of IEC-61025 standard for conducting fault tree analysis. Moreover, the tool support automating our solution and proposed in this thesis may enable the derivation of FTs with ease. This ease could facilitate the integration of our approach into the current industry since complex solutions that require a significant amount of knowledge does not usually succeed.

Additionally, from the applicability of the approach, we can state that our solution presents a high opportunity for safety engineers in four different aspects:

- the generated fault trees are ECSS-compliant by incorporating all the procedural guidelines proposed in [50], which are explained in Section 7.2. This assures that the generated evidence will not be rejected due to their content or the used syntax.

- the amount of time spent on generating standard-compliant FTs is significantly reduced, compared to the manual method. Besides, the modelling of the system and the failure behaviour in CHESS toolset presents the results in a standardised solution, understandable by all stakeholders involved in the safety assessment process. Additionally, the use of CHESS toolset as a unified solution allows the user to store safety and design data in the same model, which was presented in Section 1.1 as a critical issue not solved by similar solutions.

- the safety assessment process can benefit from the use of our solution. The generation of FTs is a tedious and time-consuming task which requires previous knowledge from the analyser. Therefore, these documents cannot be used as feedback for deriving safety requirements. The evaluation of our approach shows that FTs can be used to derive such requirements in the early stages of the system design.

- the quality of the generated FTs is improved. As stated in Section 5.1, the increasing complexity of the systems has made this task error-prone when it is performed manually. The presented solution can manage any system and thus derive the FTs without committing such errors since the process is automatically developed.

The automatic generation of FTs from modelled systems has proven to be a suitable solution, not only for generating required evidence by the standards but also for deriving safety requirements. Following the methodology presented in Fig. 37 at early stages of the design, failures in the system can be detected before they are implemented, and fault tolerance mechanisms can be derived. Therefore, the robustness of the designed system is improved, and both the time needed to detect these failures and the costs related to re-design could be reduced.

Finally, the applicability of the case study shows that our approach still relies on expert-knowledge, since the solution is based on the modelling of both the system and the failure behaviour. These two aspects must be developed adequately since the generated FTs are as complete as the provided model.
8. Conclusions and future work

In this chapter, conclusions are drawn from the solution and its evaluation. Section 8.1 presents a summary of the whole work performed in this thesis. Lastly, Section 8.2 presents the limitations found during the development of the proposed solution, as well as some ideas for future improvement of our work.

8.1. Summary

This thesis has proposed a novel solution for the automatic generation of ECSS-compliant FTs from component-based systems using model-driven technologies under the scope of CHESS platform. In the early stages of design, the automatic generation of such documents, which enables fast feedback, could contribute to increase the robustness of the system and thus improve the resulting design. In order to develop our solution, an M2M transformation algorithm has been designed and implemented using ETL as the transformation language. This transformation algorithm uses as the source of information, the results generated by another failure analysis already implemented in CHESS toolset; Concerto-FLA. Understanding how this analysis works, as well as the involved meta-model, has been a critical step to design and derive our algorithm. Besides, M2M transformation principles state that the generated model must be compliant with a target meta-model. Therefore, to generate FT-based models we found and used a target meta-model, namely EMFTA, which implements all the requirements needed to fit under CHESS platform. To allow the automatic use of our solution, an Eclipse plugin that integrates the proposed algorithm has been developed as part of the CHESS toolset. The use of this platform as a unified solution which integrates all the tools needed to generate the FTs allows storing in the same model both design and safety information of the system. This presents a major leap compared to similar solutions since, according to [62], they fail to address this issue.

A case study in the aerospace domain has been conducted to evaluate the benefits presented by our approach. More specifically, an ACS has been studied and modelled, as well as the failure behaviour of its components. Even though the complexity of the system under study is low, the proposed approach contributes to reducing both the effort and time needed to derive FTs. Besides, the quality of the generated documents and the compliance with ECSS standards is implemented by following the guidelines proposed in IEC-61025 [50] (which is referred from ECSS-Q-ST-40-12C [51] for FT development and evaluation). With our solution, most human errors can be removed from the FTs, since all the elements and ports of the system, as well as its failure behaviour, are automatically analysed, no matter how many components the system has.

Besides all the practical work conducted in this thesis, we have also conducted a related studies research. This research has been used for understanding the state-of-the-art in which our solution is developed and to learn from similar solutions. Understanding the difficulties other researchers experienced, as well as identifying their strengths and limitations has helped us to develop the solution.

8.2. Limitations and future work

Besides all the advantages provided by our approach, some limitations were found during the development of this thesis:

- No systematic approach has been followed to find related studies, due to this there might be an omission of relevant literature. Even though this was not the primary goal of the thesis, making a more profound analysis could have resulted in gaining more knowledge and thus, make the process of deriving and implementing the solution easier.

- The used meta-model for the target model (FTs) was an existing one. Instead of selecting this, a novel meta-model could have been derived, which would have resulted in a more straightforward implementation of the algorithm. Nevertheless, since the used plugin for the visualisation of the FT is based on the existing meta-model, the development of a new one would not have allowed the integration of our solution with the existing visualisation tool. Therefore, it would have been necessary to develop a new visualisation tool, which would not have been possible due to the time limitation of this thesis.

- To evaluate the proposed approach, a space domain case study, more specifically the design of an ACS in the context of ECSS standards, was performed. Therefore, we can state that the drawn conclusions are only applicable to systems with similar complexity. To generalise the results to different complex software-intensive systems, it would be required to perform case studies from other domains and complexities along with their applicable standards.

These limitations can be considered as opportunities to improve our solution, so the following future lines of work have been extracted:
• As Section 7.4 discussed, a new methodology to develop safety requirements at the software level by the generated FTs analysis could be derived from our approach. The results provided could be used to derive such requirements, and companies could benefit from this methodology to facilitate and improve the process of deriving safety requirements. This work should include a comparative study between currently used methodologies and the newly proposed.

• As presented in Section 2.6, FTs are used to derive FTA, which is used to evaluate the dependability of a system. To generate such analysis, an interpretation of the results provided by our solution shall be performed. This analytical part could be added to the current approach, by automatically deriving the cut-sets for the generated FTs and informing the user which are the minimal cut-sets through a complete report. Thereby, qualitative FTA would be performed, and the dependability of the system would be evaluated in a standardised manner.

• Finally, in order to provide empirical results on how our solution improves the manual development of such documents, a set of case studies, for establishing both the time and cost benefits that our approach presents, should be conducted. Additionally, a survey on the users of our solution could be conducted to determine current limitations or improvements needed for the proposed solution, in order to be suitable for industry processes.
References


Enrique Zornoza Moreno  

Model-based approach for automatic generation of IEC-61025 standard compliant fault trees

[51] European Cooperation for Space Standardization ECSS-Q-ST-40-12C Space product assurance- FTA.
[58] European Standards “Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems” EN 50128:2011.
Appendix A: Generated FTs for Scenario 1

Fig A. 1: FT value subtle ctrlTorque.

Fig A. 2: FT value subtle of pdTorque.
Fig A. 2: value coarse of feedforwardController.

Fig A. 2: FT value coarse of pdTorque.

Fig A. 4: FT value subtle of feedforwardController.
Appendix B: Generated FTs for Scenario 2

Fig B. 1: FT value subtle of ctrlTorque.

Fig B. 2: FT value coarse of pdTorque.
Fig B. 3: FT value subtle of pdTorque.

Fig B. 4: FT value subtle of feedforwardController.
Appendix C: User manual

This appendix presents the steps that the user needs to follow to use the solution proposed in this thesis. We assume that the user has already modelled both the system and the failure behaviour of the components and that Concerto-FLA has been executed.

1. Eclipse does not allow to create or import files into the workspace unless they are contained in a project. Due to this, the user needs to create an empty project into the current workspace. If the user already has one project into the workspace, this step can be skipped. Fig. C1 and C2 show how to create an empty project into Eclipse.

![Fig. C1: Creation of Eclipse project (1).](image1)

![Fig. C2: Creation of Eclipse project (2).](image2)
2. Once this project is created, the results obtained from the execution of Concerto-FLA (.flamm file) need to be imported or copied into this project. Then, we click on the button titled “Dependability Analysis” from the Eclipse top menu section. This opens a sub-menu section which only contains the option “Generate Fault Trees” which needs to be clicked to start the transformation process. Fig. C 3 shows the previously explained buttons.

![Fig. C 3: Activate the fault tree generation process.](image)

3. Once the process starts, a dialogue like the one displayed in Fig. C 4 appears. Then, the user has to select the .flamm (imported in step 2) that needs to transform. Once it is selected, the “Open” button is clicked, and then the results are generated in a new folder under the same name of the selected .flamm file. Inside this, two files are created, i.e., the .emfta, which is derived by transformation, and the .aird, which contains the representations for the FTs. If the user selects a .flamm file which has been already used (and the results has not been deleted from the workspace), the plugin will not perform the transformation, and a warning message will appear, informing the user of such situation. Fig. C.5 presents how the previous folder is created in case a file called FirstExample.flamm is selected.

![Fig. C 4: Dialog for .flamm file selection.](image)

![Fig. C 5: Generated results in the workspace.](image)
4. When the results are generated, an editor containing the first FT created is opened automatically. To access all the FTs, the representations file needs to be opened. Then, inside this, the folders named “Representations per category” and “EMFTA” are accessed. Inside this last one, there are three directories, i.e., FTA Real Tree, FTA Tree and FTA Table View. Inside the FTA Tree one, all the representations for each generated FT are stored. By clicking on each one, they are opened on the EMFTA editor. Fig. C 6 shows the contents of such folder for the FirstExample.flamm case.

![Fig. C 6: Results layout.](image)

In the previous figure, three main elements can be distinguished. The project explorer, inside the blue box, stores all the representations for the fault trees by following the previously explained structure. The green box, on the top right section, is the editor which presents the diagram for the selected FT. By clicking on each FT contained in the FTA Tree folder, a new representation is opened in this editor. Lastly, the red box on the bottom left section presents an outline of the FT presented in the editor.

5. Finally, the user can export these representations as image files. To do so, once the selected FT representation is opened in the editor, the user needs to right-click anywhere inside it. Then, the option “Export diagram as image” is selected as Fig. C 7 shows. Once the user specifies the target location to store the file, the image is exported.
Fig. C 7: Export diagram functionality.