# Safety Analysis of SysML Models in the Context of Model-Driven Engineering

by

**Bashar Al shboul**

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

## Abstract

Safety Critical Systems (SCS) must operate inside of their prescribed specifications, otherwise, they can cause harm to the user(s) and/or the environment. These systems are used in various domains, such as aerospace, automotive, railway and healthcare.

Safety Analysis (SA) is performed on SCS to ensure that they are safe enough to be operational. Many SA techniques already exist and had proven their effectiveness; hence their use is recommended and, in some cases mandated, by industry standards and certification authorities.

This thesis aims to develop a methodology termed Model-Driven Safety Engineering (MDSE) for integrating well-established SA methods with standard and well-known tools and techniques within the Model Driven Engineering (MDE) system development process. The proposed methodology can be applied with minimal learning effort. It brings multiple benefits, such as increasing the safety and confidence level of SCS, reducing the costs in various aspects and enhancing the communication between all stakeholders.

The proposed approach follows the MDE process by modeling the system under development with the System Modeling Language (SysML), an Object Management Group (OMG) standard. The SysML model is extended with safety annotations using another OMG standard, the UML Profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE), and its dependability extension, the Dependability Analysis and Modeling (DAM) profile. We propose a multi-step automatic model transformation, where a SysML model annotated with safety information is transformed into models for Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA). To guide the synthesis of fault tree models, we propose a set of failure mapping

patterns, which show how model elements representing failure in the source model are mapped to the target model. The first step transforms the annotated SysML model into Component-based Fault Trees (CFT) models (one CFT per component). A second transformation composes CFTs, producing System-level Fault Trees (SFT). A third transformation feeds back the quantitative results obtained by solving the CFTs and SFTs models to the SysML model. A final transformation synthesizes an FMEA model from the system SysML model and the generated FTs, to keep the FT and FMEA models synchronized.

## Acknowledgements

All praises are due to God (ALLAH), the most merciful and most compassionate. I am so grateful to ALLAH for his uncountable bounties on me, including the ability to perform and write this thesis.

First of all, my deep and sincere appreciation goes to my thesis supervisor, Professor Dorina C. Petriu, for continued support, kind cooperation, and many long hours of discussions. This thesis would not have been possible without her guidance and assistance. Thank you for everything you provided me. You will be my model to follow in my academic life.

I am infinitely grateful to my parents and my lovely family. My father is the source of inspiration, and his unlimited support and advice helped me face any challenge. My mother is the source of love, and her prayers are lightening my path to achieve my goals. My wife and my kids always supported and encouraged me to accomplish this achievement and other goals. My siblings specifically Mahmoud. My inlaws for their kind support. I am thankful to all of them, and GOD bless them.

I dedicate this work to my parents, for all the sacrifices they made to raise me and provide me with the means to be a successful person.

I also dedicate this work to my beautiful wife Amani for her love, support, and patience throughout this journey.

Finally, I dedicate this work to my kids Anmar and Gardenia.

# Table of Contents

## List of Tables

## List of Figures

## List of Code Fragments

## List of Acronyms

| | |
|---|---|
| AADL | Architecture Analysis & Design Language |
| ATL | ATLAS Transformation Language |
| BDD | Block Definition Diagram |
| CAFTA | Computer Aided Fault Tree Analysis |
| CBM | Component-Based Modeling |
| CFM | Common Failure Modes |
| CFT | Component-level Fault Trees |
| CSD | Composite Structure Diagrams |
| DAM | Dependability Analysis and Modeling |
| DSL | Domain Specific Language |
| EGL | Epsilon Generation Language |
| EMFTA | EMF-based Fault Tree Analysis |
| EOL | Epsilon Object Language |
| Epsilon | Extensible Platform of Integrated Languages for mOdel maNagement |
| ETL | Epsilon Transformation Language |
| FAA | Federal Aviation Administration |
| FFA | Functional Failure Analysis |
| FMEA | Failure Mode and Effects Analysis |
| FMECA | Failure Mode Effect and Criticality Analysis |
| FTA | Fault Tree Analysis |
| GSPN | Generalized Stochastic Petri Nets |
| IBD | Internal Block Diagram |
| IDM | Intermediate Dependability Model |
| MARTE | Modeling and Analysis of Real-Time Embedded Systems |
| MAS | Mission Avionics System |
| MCS | Minimal Cut Set |
| MDA | Model Driven Architecture |
| MDE | Model Driven Engineering |
| MDSE | Model-Driven Safety Engineering |

| | |
|---|---|
| MMB | Mission Management Bus |
| MOF | Meta-Object Facility |
| NFP | Non-Function Properties |
| OCL | Object Constraint Language |
| OMG | Object Management Group |
| OSATE | Open Source AADL Tool Environment |
| PLPD | Partial Loss of Position Data |
| QN | Queueing Network |
| RBD | Reliability Block Diagram |
| SA | Safety Analysis |
| SCS | Safety Critical Systems |
| SEFT | State/Event Fault Trees |
| SFT | System-level Fault Trees |
| SMD | State Machine Diagram |
| SOAM | Safety Oriented Architecture Modeling |
| SPL | Software Product Lines |
| UML | Unified Modeling Language |
| XSLT | Extensible Stylesheet Language Transformations |

# 1 Chapter: Introduction

Computer systems are widely used today in a multitude of domains. Individuals, communities, governments, industry, and organizations rely on computer technology to produce or innovate many aspects in a variety of areas, such as communication, education, healthcare, transportation, food, services, entertainment. The increase in the utilization of computer systems has significantly raised their complexity levels. Furthermore, there is an increase in the demand to build systems that satisfy various Non-Functional Properties (NFP), such as performance, dependability, security, safety. Regulatory authorities and concerned bodies have put regulations and standards in different domains to control the development of such systems (SAE-ARP4761, 1996). This has triggered many research efforts in the academia and industry targeting the analysis of NFPs.

Dependability, the general NFP of interest in this thesis, defined in (Avižienis et al., 2004) as the ability to deliver services that can justifiably be trusted, is comprised of five main attributes: Availability, Reliability, Maintainability, Integrity, and Safety. The last one, safety, is the specific NFP of interest in this thesis.

Safety is defined as the absence of catastrophic consequences suffered by the user(s) and/or the environment (Avižienis et al., 2004). Safety-Critical Systems (SCS) are the type of systems that can cause harm to the user(s) and/or the environment if they are operating outside the prescribed specifications. These systems are found in various domains, such as aerospace, automotive, railway, and healthcare. Safety Analysis (SA) is performed on SCS to ensure they are safe enough to be operational.

As discussed in (Ericson, 2011), the diminishing of system safety may be a reason for accidents and mishaps that could cause human life losses and high cost damages, and therefore should be avoided. Moreover, it is important to assure that every system is safe, especially the safety critical systems, as safety cannot be achieved by chance. As an example of standards making reference to safety analysis techniques, the document (MIL-STD-1316E, 1998) describes the use of Fault Tree Analysis (FTA) to obtain numerical values for the probability of events and the use of Failure Mode and Effects Analysis (FMEA) to document safety analysis.

The survey (Nair et al., 2014) performs a systematic literature review on safety-critical systems SCS (covering the period from 1990 to 2012), which concentrates on evidence artifacts required for safety certification. Certifying the safety of an SCS is being able to assure that the system operates safely in the presence of known hazards. Furthermore, evidence is defined as: “The available body of facts or information indicating whether a belief or proposition is true or valid”. It shows that the number of publications targeting SCS and the safety evidence is increasing in all domains, based on their respective standards, such as (IEC 61508, 2005) for programmable electronic systems, (RTCA, 2011) in the aviation domain, (CENELEC - EN 50129, 2018) in the railway domain and (ISO 26262, 2018) in the automotive domain. An interesting conclusion is that very few works are using model-based methodologies to obtain evidence of compliance with safety standards.

Model-driven methodologies are gaining momentum within various industries, where abstract models are part of the development methodology. Model-Driven Engineering (MDE) considers models rather than code as the key artifacts of the system engineering

development process. Some of the benefits of using models are: 1) Models are more abstract than code, so they are easier to comprehend by different stakeholders; 2) Models expressed in standard languages can simplify and unify the communication between all interested parties; 3) Model can be used for automatic code generation for software systems; 4) Models enable the verification and the validation of various NFPs, including safety.

According to the survey (Russo and Scippacercola, 2016), MDE can assist in generating safety evidence, because it models the requirements, providing traces from requirements to all kind of system artifacts throughout the lifecycle. Also, MDE can generate reports and artifacts via automatic model transformations, which can contribute to the evidence. Moreover, models can show that potential risks are considered and appropriately dealt with. The survey identifies a number of open issues with regard to the application of MDE and their industrial integration. The first issue is the amount of effort needed to adopt an approach for practical use within industrial processes. The second issue is the integration within existing industrial processes rather than replacing the processes entirely. The third issue regards the use of modeling languages, such as UML, where different approaches introduce new UML profiles to express different practical needs. The diversity of the profiles affects the semantics of the language along with the tool support, making it complicated for the users. The last issue is the availability and maturity of MDE tools. The idea is that the use of open source tools and openly standardized languages is preferable, specifically for long-term projects, rather than using paid close source tools and proprietary languages. The use of SysML was recommended as a more viable option for modeling

safety-critical systems, as it is a standardized language targeting modeling at system level, compared to UML that is a software modeling language.

SysML was chosen as the modeling language in this thesis considering the following characteristics described in (Friedenthal et al., 2014). First, it supports the specification, design, analysis, and verification of systems as a general-purpose modeling language. Second, it provides graphical models representing requirements, structure, behaviour, and properties of systems and their components. Third, it is a standardized and robust language. Finally, it is an extension of a subset of the standard UML language, which adds the benefit of extending SysML with the standardized profiles defined for extending UML, such as MARTE (OMG, 2011).

Many safety analysis techniques have been around for a considerable time and have proven their effectiveness in performing SA; hence they are recommended and, in some cases, mandated by industry standards and certification authorities (e.g., SAE-ARP4761, 1996). Two well established SA techniques are *Fault Tree Analysis* (FTA) and *Fault Model and Effect Analysis* (FMEA). However, applying these techniques raises various challenges. Traditionally, both techniques are performed manually during the preliminary stages of the development cycle, but their application is error-prone, time-consuming and costly, especially for systems with high complexity. These challenges prevent the continuous application of these techniques to keep the models synchronized when design changes occur, or in later phases to validate the safety requirements.

This thesis contributes to an important aspect of safety analysis that was identified in literature surveys as not being enough supported by tools: the automatic derivation of safety analysis models (namely FTA and FMEA) by model transformation from SysML design

models of the system under construction annotated with relevant safety information. The goal is to avoid error-prone manual work, helping to keep the design and analysis model synchronized throughout the development process, and to maintain the cross-model traceability between the software models used for development and the analysis models used to verify their NFPs.

## 1.1 Motivation and objectives

Some systems have severe failures or reaction to failures, such as causing physical harm to people or causing substantial environmental pollution. In (Knight, 2012), it is stated that Safety NFP is about handling severe results of failures, and a Safety-Critical System (SCS) is defined as a system having severe failure consequences. SCS needs a thorough, comprehensive and systematic evaluation of their safety requirements. Many interested parties participated in creating standards and regulations that such systems must meet before they are fully operational.

Safety, as one of the many attributes of dependability, is to be considered within the development of the system throughout its lifecycle. Some works discuss the need for integrating dependability NFPs within the development life cycle. For example, (Kaâniche et al., 2002) addresses this problem and proposes a framework for such integration.

Safety-critical systems are growing and expanding rapidly, raising their safety evaluation complexity when integrated within the development life cycle. MDE has shown its practicality in tackling this complexity, by employing models which are more abstract than code and separating the business/functional logic from the underlying technologies, allowing both to evolve separately.

The survey (Bernardi et al., 2012) reviews the work in modeling and analyzing software systems dependability within the context of MDE using UML. The surveyed works target one or more of the dependability attributes. One of the conclusions is that the surveyed works provide support mainly in the early phases of the software life cycle (i.e., from requirements to design), while there is a lack of support for later phases.

The paper (Zeller et al., 2016) discusses the current practices in the safety-critical systems industry, analyzing the challenges faced and the benefits of MDE to tackle such challenges at Siemens. The challenges discussed are: building and maintaining SA throughout the life cycle, accommodating for system changes while maintaining traceability with the SA artifacts, SA artifacts reuse, and the automation of SA. The use of MDE helps the automation of the SA process, as it has the ability to represent the system at various abstraction levels, based on the development phase and system modularity, allowing for the reuse of SA artifacts and the ability to provide traceability between system model elements and SA artifacts. Moreover, it shows an industrial need to obtain and maintain the SA results from techniques such as FTA and FMEA for certification purposes.

Another recent survey (Berres and Schumann, 2016) analyzes the status of the automatic application of FTA for system safety verification. An interesting point is that most current works provide their own defined model rather than using standard modeling languages such as AADL and SysML. Also, the representation of the failure behaviour of the system is performed using different non-standardized approaches. Furthermore, the algorithms used for the generation of the FTs out of the proposed system models are also diverse, due to the diversity of system models proposed.

The goal of this thesis is to develop an MDE based methodology with safety concerns in mind, utilizing standard and well-known techniques and open source tools that would require minimal effort to perform safety analysis during the development of SCSs, by allowing to maintain and reuse safety artifacts throughout the development life cycle of the system.

The proposed approach follows the MDE process by modeling the system under development with the System Modeling Language (SysML), a standard of the Object Management Group (OMG). The SysML model is extended with safety annotations using another standard from OMG, the UML Profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) (OMG, 2011), along with an extension profile focused on dependability, called Dependability Analysis and Modeling (DAM) profile (Bernardi et al., 2013). We propose a multi-step automatic model transformation, by which a SysML model annotated with safety information is the starting point for synthesizing models for two safety analysis methods, Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA).

Following MDE techniques, the system models are transformed through a multi-step transformation into FTA and FMEA safety analysis models, which are recommended by various safety standards for different domains, such as (IEC 61508, 2005), (SAE-ARP4754, 1996, p. 4754) and (SAE-ARP4761, 1996).

To abstract and guide the FT model generation, we propose a set of failure mapping patterns, which show how model elements representing a failure in the source and target models are mapped to each other. The mapping patterns are divided into two categories: behavioural failure-mapping patterns (where state-machine behaviour is transformed into

fault tree elements) and structural failure-mapping patterns (where structural model elements are transformed into fault tree elements).

The first transformation step realizes the behavioural failure-mapping patterns within the component-based modeling approach by synthesizing component-based fault trees (CFT) models which enable the reuse of CFT. In the following step, a System-level fault tree (SFT) safety model is obtained by realizing the structural failure-mapping patterns, composing the CFTs based on the components' interconnections. The generated SFT models are used for qualitative and quantitative system safety verification. A visualization of the generated FTs is also provided. Subsequently, a feedback step is performed from the SFTs and CFTs, by feeding back related analysis results into the SysML model. In a successive step, an FMEA model is synthesized from the SysML system model and the analysis models, CFTs and SFTs. This is similar to (Knight, 2012), where FTA and FMEA are complementary techniques, where the use of FTA precedes FMEA, allowing to use FTA results as input to FMEA.

## 1.2 Thesis contributions

The objective of this thesis is to integrate safety analysis based on Fault Trees (FT) and Failure Mode and Effects Analysis (FMEA) into model-driven engineering of safety-critical systems. More specifically, a multi-step automatic model transformation is designed and developed, by which a SysML model annotated with safety information is the starting point for synthesizing FT and FMEA models. The thesis contributions are as follows:

- Propose a set of *failure mapping patterns* that show how model elements representing failure in the source and target model are mapped to each other. A mapping pattern is a reusable solution to a mapping problem between the source and the target model of a transformation, and it has two parts: a) a group of inter-related source model elements (i.e., source pattern); and b) a group of inter-related target model elements (i.e., target pattern). A mapping pattern is composed of a source pattern mapped to a target pattern (Section 4.1).

- Design two transformation algorithms for deriving component fault trees (CFT) and system fault trees (SFT) models from a SysML system model annotated with safety information. The two algorithms are based on the failure mapping patterns defined above and are independent of the language used to implement the transformation (Section 4.2 and 4.3). The two algorithms are used to develop two transformations for deriving CFT and SFT in the Epsilon Transformation Language (ETL) specialized for model transformations (Chapter 5).

- Design an in-place transformation to feed back the FT analysis results obtained by solving the generated FT model into the original SysML system model, based on cross-model traces obtained at the time of deriving the CFT and SFT models. The transformation is developed using Epsilon Object Language (EOL) (Chapter 6).

- Propose a transformation mapping and design an algorithm to generate an FMEA model from the SysML system source model and FT models, relying on the cross-model traces between the SysML and FT models. Develop an automated model-to-text transformation realizing the algorithm in Epsilon Generation Language (EGL) (Chapter 7).

- Evaluate and verify the proposed safety analysis approach on a number of industry inspired case studies from literature to make sure that our method covers necessary modeling features and to avoid author bias (Chapter 8).

An objective of the research was to use standard modeling languages and well known open-source tools, which do not require a steep learning curve when they are adopted for existing or for new projects. Also, we participated in the open source community, identifying issues with the tools and reporting them.

The following papers are the outcome of the thesis research:

Alshboul, B. and Petriu, D.C. (2018) Automatic Derivation of Fault Tree Models from SysML Models for Safety Analysis. *Journal of Software Engineering and Applications*, 11, 17. https://doi.org/10.4236/jsea.2018.115013.

Alshboul, B. and Petriu, D.C. (2019), Pattern-based transformation of SysML Models into Fault Tree Models. in preparation, to be submitted to *International Symposium on Software Reliability Engineering (ISSRE)*, 2019.

Alshboul, B. and Petriu, D.C. (2019) Safety Analysis Models generation from SysML Models. in preparation, to be submitted as a book chapter to *Advances in Computers and Software Engineering: Reviews, Vol. 2*, 2019.

## 1.3 Thesis content

This thesis is organized as follows: Chapter 2 presents essential relevant concepts on system modeling and safety analysis and discusses the state of the art in the field. Chapter 3 gives a high-level overview of the proposed safety analysis approach and multi-step

transformation, describing the source and target models involved in the transformation. Chapter 4 introduces a number of failure mapping patterns that show how model elements representing failure in the source and target model are mapped to each other. We identified these mapping patterns in different case studies. Based on the mapping patterns and the properties of the source and target models, two algorithms were developed for the transformation of the system source model for the synthesis of CFT and SFT models. Chapter 5 provides the implementation in the Epsilon Transformation Language (ETL) of the proposed algorithms for the two-step FT transformation to generate CFT and SFT models. Chapter 6 presents the transformation for feeding back the fault tree analysis results and for updating the system source model with these results based on traversing in reverse cross-model tracelinks between SysML and FT models. Chapter 7 gives the details of an FMEA model, then proposes an algorithm to synthesize it. This is completed by providing the transformation details to generate the FMEA model from the updated system model and the generated FT models. Chapter 8 evaluates the proposed approach and the developed transformations by presenting three case studies from the literature to which the proposed approach is applied. Chapter 9 concludes with the achievements and limitations of the work presented in this thesis and discusses directions for future work.

# 2 Chapter: Background and state of the art

This chapter provides some relevant background concepts and discusses related works. It starts with a brief description of related concepts and terminology, followed by the details of related works within the field of safety analysis. Finally, a summary of the most relevant works and their significant contributions is given.

## 2.1 Concepts

Object Management Group's (OMG) Model Driven Architecture (MDA) (OMG, 2014) is a model-driven development process that targets the separation of business logic from the realizing technology, which tackles the challenge of separating the evolution of the business logic and the technology. This is achieved by modeling the business logic using OMG's standards such as UML/SysML to be independent of the realizing technology.

The MDA Guide (OMG, 2014) is listing several benefits of this development approach based on models, such as the following:

- The use of models has the advantage that it improves and enhances inter-team communication. As stated in multiple studies, e.g., (Zoughbi et al., 2011) the communication between different stakeholders is a source of concern.
- Specific artifacts can be synthesized from models using automated transformations. The transformation can be partially or fully automated. The synthesis of an artifact consists of a transformation process that takes as input a source model parametrized for interpretation and generates a target artifact. This includes the generation of code and analysis models for a given source system.

Other benefits and further details can be found in the MDA Guide (OMG, 2014).

Component-Based Modeling (CBM) is a structural approach to handle the system complexity, where a system is decomposed into interacting components in order to deliver the target system functionality. The decomposition into components is recursive, applied until a sufficient granularity level is reached.

### 2.1.1 System Modeling Language (SysML)

SysML is a general-purpose standard modeling language for system engineering standardized by the Object Management Group (OMG, 2015a). SysML is an extension of a subset of another OMG standard, the Unified Modeling Language v2 (UML) (OMG, 2015b). Figure 2-1 taken from (OMG, 2015a) shows a visualization of the relationship between SysML and UML. Furthermore, SysML introduces new diagrams and modifies existing UML diagrams, as shown in Figure 2-2 taken from (OMG, 2018). It is worth mentioning that OMG's members range from government agencies and industry to academia, representing a variety of stakeholders.

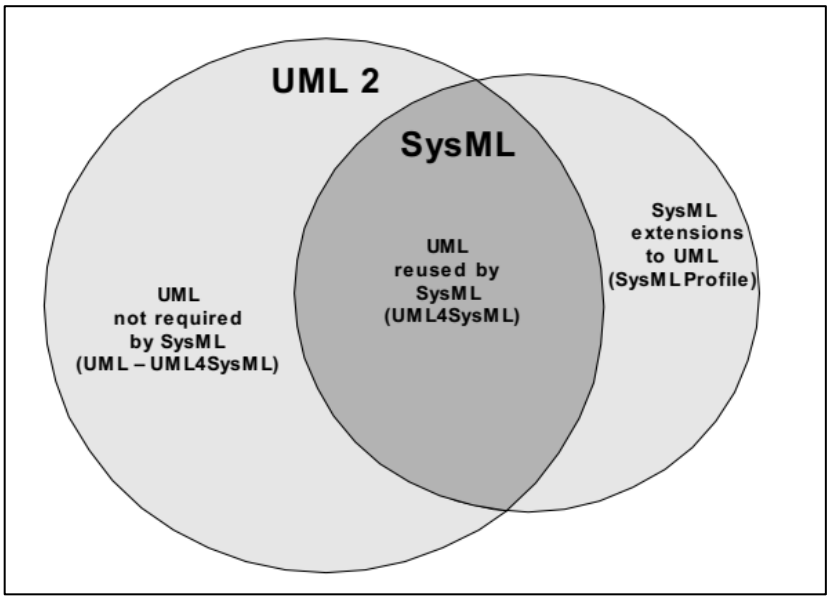



**Figure 2-1: SysML and UML relationship visualization (OMG, 2015a)**

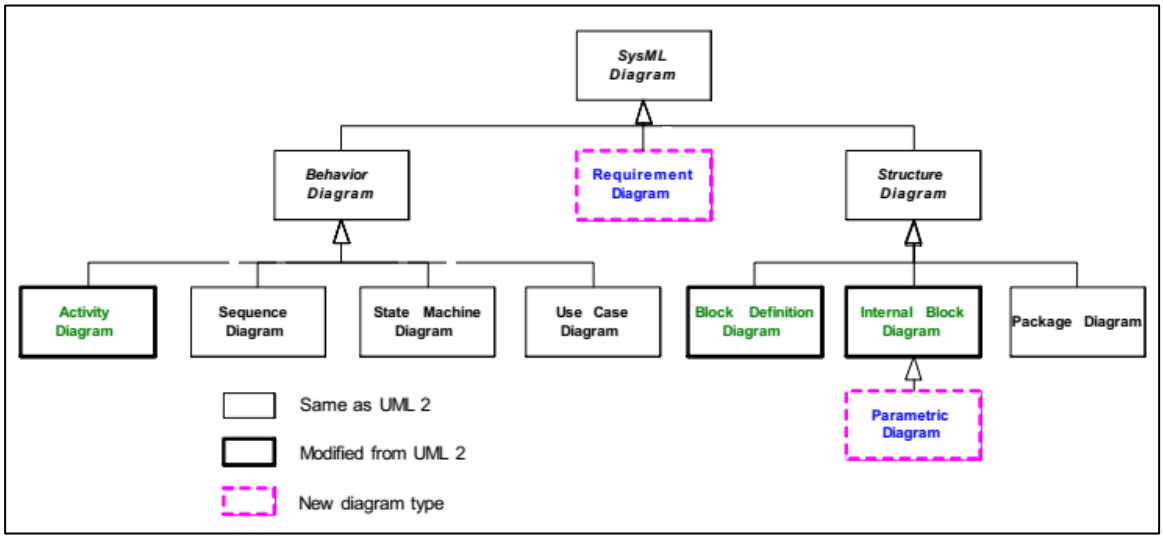



**Figure 2-2: Taxonomy of SysML diagrams taken from (OMG, 2018)**

SysML can be used within MDE methodologies for systems from various domains, including safety-critical systems. Various modeling tools support the OMG standard modeling language UML and SysML. One of these tools is Papyrus, an industrial-grade open source Model-Based Engineering tool (The Eclipse Foundation, 2017a), which is used in this thesis.

SysML is described by a metamodel, which is itself a more abstract model that describes the elements of SysML. In other words, a SysML model conforms to (or is an instance of) the SysML metamodel. In turn, a metamodel is described by a meta-metamodel. The Meta-Object Facility (MOF) is a meta-metamodel that provides the basis for metamodel definition in OMG's family of modeling languages (including the UML and SysML).

A powerful feature of UML/SysML is the profile mechanism, which allows for extending existing model elements for a specific need, such as modeling domain-specific concepts. Profiles can be a way to model Domain Specific Languages (DSLs) based on UML, such as MARTE and DAM profiles.

A UML profile standardized by OMG to add the power to model Real-Time and Embedded Systems is called Modeling and Analysis of Real-Time and Embedded Systems (MARTE) (OMG, 2011). This profile supports the specification, design, and verification/validation of real-time systems. The benefits of MARTE consist not only in facilitating the modeling of real-time systems but also in supporting performance and schedulability analysis. Since SysML is an extension of UML, the MARTE profile can be applied to it as well.

To allow the modeling of specific dependability NFPs and their attributes, another UML profile was defined, called the Dependability Modeling and Analysis (DAM) profile (Bernardi et al., 2013), which in turn extends MARTE. In this work, we apply both MARTE and DAM to SysML system models.

### 2.1.2 Failure Tree Analysis (FTA)

Failure Tree Analysis (FTA) is a deductive top-down failure-based analysis technique (Stamatelatos et al., 2002), in which an "undesired state" of a system (also known as a "system failure condition" or "top event") is analyzed using Boolean logic to combine a series of lower-level events, represented in a tree-like structure called Fault Tree (FT) model. A FT depicts how the set of identified events are combined logically, leading to the specific undesired top event. It contains various symbols for the several types of events and logical relations called gates. The FTA benefits consist mainly of providing critical qualitative assessment, but it can be quantified to provide quantitative results as well. Such information enables FTA to be an instrumental technique that can help system developers to understand how systems can fail, to identify the best ways to reduce the risk or to determine the event rates of a safety accident or a particular system-level (functional) failure.

Here are some FTA basics. FTA is a technique that identifies the set of events that will cause the occurrence of a specific undesired event. FT depicts how the set of identified events are combined logically, leading to the specific undesired top event. FT contains various symbols for the various types of model elements, which are either events or logical relations called gates.

According to (Stamatelatos et al., 2002), there are a number of benefits of using FTA to support decision making throughout a system's life cycle:

- The logical comprehension of the causes for inducing a specific top event and understanding what intermediate events are involved.
- The prioritization of the top event contributors.
- A proactive top event prevention tool, which allows evaluating the upgrades and corrective measures on the top event.
- Evaluation of design alternatives.

A FT can be evaluated both qualitatively and quantitatively. The key qualitative information FT provides are: a) the cut set representing the set of basic events leading to the top event, and b) the Minimal Cut Set (MCS) consisting of the smallest set of basic events whose occurrence will result in the undesired top event. A FT can have multiple cut sets and MCSs. These can be obtained not only for the top undesired event of the tree, but also for any intermediate higher-level event. The cut sets and the MCSs can be ordered for importance, based on the number of events they contain.

The FT quantitative evaluation mainly provides the occurrence probability of the top event and the significance of the basic events toward the top event. The identified events are

assigned a probability of occurrence; once such information is available, this allows the ordering of the events and obtaining the cut sets based on their contribution to the occurrence of the top event. The cut sets with the highest significance in the occurrence of the top event are called “dominant cut sets”.

The FTA technique starts with defining a failure mode for the system as an undesired event. The undesired event is systematically analyzed to deduce the immediate events that are necessary for its occurrence. Then, each identified event is seen as an undesired sub-event and - similarly to the top event - is analyzed to obtains its causing events. This process is recursively performed until a certain level is reached, such as the existence of failure probabilities for the identified events, which will be considered as basic events.

A FT is used to model the safety properties by setting the undesired system failure as a top event for the FT. The events that have been identified contributing to the occurrence of the top event are then combined using Boolean logic, which specifies how they lead to it. This process is performed for each of the events until basic events are reached.

An example FT shown in Figure 2-3 contains a top event named System Failure (SF), which is an undesired event that will lead to causing harm to the operational context of the system. The OR-gate connected to the SF event indicates that the occurrence of any of the connected events will result in the occurrence of the SF event: either the basic event of component X failure, or the intermediate event of the joined failure of component Y and Z, YZ event. The YZ event can be seen as a sub-top event, which is connected to an AND-gate, illustrating that both failure events of component Y and Z have to occur, for the YZ event to occur.

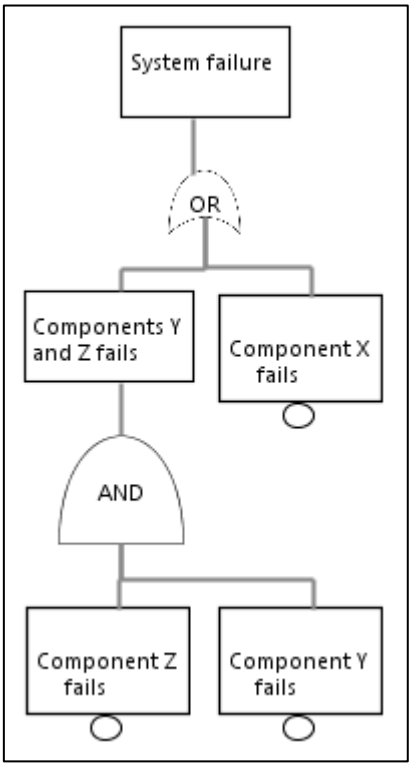



**Figure 2-3: A simple Fault Tree**

### 2.1.3 Failure Mode Effect and Analysis (FMEA)

Another technique used for the safety analysis of SCS is FMEA, an inductive bottom-up analysis technique, working in the opposite direction than deductive techniques such as FTA. It is inductive in the sense that it starts with a list of known specific events and analyzes their more general effects. FMEA considers the failure modes of low-level components individually and analyzes their effects on the system. It is used for performing safety analysis on safety critical systems, as described in (GSFC, 1996). It provides both qualitative and quantitative values of the analyzed system properties.

An extension of FMEA called Failure Mode Effect and Criticality Analysis (FMECA) (Xiao et al., 2011), adds more focus on the criticality of the failure modes along with considering assurances and controls against their occurrence.

Reliability Block Diagram (RBD) (Bernardi et al., 2013) is another inductive technique. The system is decomposed into blocks, which are then connected to represent success pathways. RBD is used to performed in-depth search of possible success paths of the systems' blocks with explicit dependency among the blocks.

Applying deductive and inductive techniques have been proposed and recommended by interested parties in various domains, such as NASA's FTA handbook (Stamatelatos et al., 2002), which can help in obtaining a comprehensive list of failures.

### 2.1.4 Languages for model management

The Extensible Platform of Integrated Languages for mOdel maNagement (Epsilon) (Kolovos et al., 2017) provides a set of languages with a shared expression language. Epsilon empowers model management through performing various model-based tasks implemented as task-specific languages. Epsilon tasks include model transformation, code generation, model comparison, model merging, model refactoring, and model validation.

The shared expression language used in all task-specific languages within Epsilon, called Epsilon Object Language (EOL), is a model-based language combining JavaScript style along with utilizing Object Constraint Language (OCL) strength in object querying.

One of the task-specific languages of Epsilon is Epsilon Transformation Language (ETL). ETL is a task-specific language specialized for model-to-model transformation. ETL accepts multiple source models and can generate multiple target models. It is a rule-based language categorized as a hybrid language, because it supports both declarative and imperative rules. Additional features are also supported, such as rule inheritance. ETL is

used in this thesis work for realizing transformations from SysML to FT and FMEA models.

## 2.2 State of the art

This section provides a literature review of related works in safety modeling and analysis followed by a summary of the current status.

### 2.2.1 Literature review

The works presented here have been discussed with regards to the following characteristics:

- The representation of the system of interest, i.e., the modeling language used.
- Any proposed extension to enhance the representation of the system of interest.
- Modeling the NFP of interest.
- The formal analysis targeted/performed.
- The execution of the proposed work concerning the level of automation.

#### 2.2.1.1 UML/SysML system model: custom profiles for safety information

In (Hassan et al., 2005) three well known SA techniques are integrated: Functional Failure Analysis (FFA), FMEA and FTA on UML based system models, generating Cost of Failure graphs (Sherer, 1988), which enable the calculation of the cost of scenarios executions failure for participating components and connectors. FFA is a manual process that identifies the system-level hazards out of the system scenario diagrams. The process follows by performing FMEA on sequence diagrams obtaining components and connectors

failures. The process wraps up by combining the results of the techniques mentioned above using FTA.

In (Hause and Thom, 2007) it is discussed how SysML/UML models can be used to improve team collaboration and communications by integrating various concerns within these models by utilizing a UML profile proposed by the authors. An example of how a safety case can be obtained from SysML models annotated with a proposed profile is illustrated in the paper.

In (Douglass, 2009) a Safety Analysis profile is introduced as a UML extension to model FT utilizing the IBM Rational Rhapsody tool. The work shows how to model SA results manually as a FT using the profile. The FT model is explored, and a tabular representation of the list of modeled faults and their attributes is generated. Then, the elements of the FT model are related to the requirements, using information from the profile. This assists in communication between different stakeholders.

In (Zoughbi et al., 2011) it is proposed a UML profile called SafeUML whose synthesis is based on DO-178B (RTCA, 1992) in the aerospace domain. This work shows that the lack of communication between involved stakeholders, such as software engineers, safety engineers, and certification authorities, is a source of many safety-critical challenges. The profile adds safety information to class diagrams.

In (Biggs et al., 2014) it is proposed a profile for SysML called SafeML to tackle the challenges of the communication between safety and system engineers along with the documentation of the safety analysis results. The system is modeled using SysML, then safety analysis is performed using a traditional safety analysis technique. Once the SysML model is built, the SafeML profile is used to include safety attributes in Block Definition

Diagram (BDDs) as part of the system model, depicting the safety concerns and their mitigation techniques.

In (Gómez-Martínez et al., 2014) an approach to model safety requirements on safety critical systems as OCL constraints is proposed, representing safety contracts on a component-based model. The system is modeled using UML: SD and CD are annotated using MARTE with OCL constraints representing the safety requirements in terms of system components. The system model is transformed into a formal model of Generalized Stochastic Petri Nets (GSPN). The ArgoSPE tool is used to obtain the GSPN model, which is then manually modified using the GreatSPN tool to represent the OCL constraints.

In (Yakymets et al., 2015) a methodology is proposed for performing MBSA of models created with Papyrus in SysML, utilizing the Sophia framework to perform FTA. The system is modeled using BDDs and (Internal Block Diagram) IBDs, then a safety engineer analyzes the system and provides safety objectives. A proposed profile is used to annotate the system blocks with failure modes. The failure modes of a component are modeled using Boolean expressions of the internal and input failures combinations. The framework translates the system model into AltaRica model using the ARC engine to generate FTs, which are then analyzed using the proprietary XFTA engine. Also, it includes a discussion about the possibility of using FMEA with FTA to better analyze system faults.

In (Wu et al., 2015) a methodology for the avionics domain based on the DO-178B and DO-178C (RTCA, 2011, 1992) is proposed. The methodology, called Safety Oriented Architecture Modeling (SOAM), introduces a UML profile and a process for its application. The safety requirements in DO-178B and DO-178C are the domain concepts represented in the profile, which also subsets the MARTE profile. The profile stereotypes

and Data types application is constrained using OCL constraints. The system is modeled as a UML Component diagram. The methodology is performed at the architecture design level, providing a process for applying the profile by relying on OCL constraints to validate the conformance of the elements with the mentioned standards. IBM Rational Software Architect (IBM, 2017) is the UML tool used with this methodology.

In (Müller et al., 2016) an approach to be used in the early stages of the development life cycle is proposed, along with a SysML profile to add safety-related information to the model elements. The procedure starts in the Requirements Elicitation phase, where safety requirements are identified. Manual analysis is performed on the functional architecture proposing safety functions, then the component architecture is analyzed manually adding the information from each analysis to the system model by stereotyping elements in (ADs) and BDDs using the proposed profile.

#### 2.2.1.2 Automated generation of safety analysis models

Various approaches target the automation of the generation of SA models. They vary in the use of modeling language, the representation of the safety information, synthesizing, analysis and feedback of the analysis model and tools used.

In (Miguel et al., 2008) and (Briones et al., 2007) a UML profile to annotate UML structural diagrams with safety information is proposed. The profile is based on the concepts of FTA and FMECA. The safety information to be added as annotations are a result of a preliminary hazard analysis. Next, a Java program is used to build FTA and FMECA models and perform formal safety analysis.

In (Kim et al., 2012) FT models are generated from UML models to be used in run-time fault detection. The system is modeled using Component diagram, State Machine Diagram (SMD) and Class diagram. Each component is modeled as a class, with at least one property providing input and one property providing output. In the class diagram, a note is linked to each class out property specifying its faulty output based on its input and the system operating state. A failure event is manually selected and set the top event of the FT, and its related class is used as a starting point for tracing toward basic events based on the annotated output classes' properties. The generated FTs only contain OR gates. An expert is to review and possibly modify the tree as required. At run-time, self-healing can use the generated tree for possible faults.

In (Zhao, 2014) automatic model transformation from systems modeled in UML into FTs is proposed in order to perform dependability analysis. The system is modeled using Use Case Diagrams (UC), Sequence Diagrams (SD), and Composite Structure Diagrams (CSD). These models are then annotated with dependability information using MARTE and DAM (Bernardi et al., 2013). The annotated system models are transformed into an FT model considering one kind of failure modes. The transformation is written in ATLAS Transformation Language (ATL) (Obeo, 2017).

In (Mhenni et al., 2018) a methodology is introduced for integrating SA within the software development process by using Python (Python Software Foundation, 2017), a generic programming language. The SA techniques used are FTA and FMEA, with the system modeled in SysML. The system functional architecture is modeled using a set of ADs representing the functions and their breakdown. A Python program parses the XMI model file generating a Functional FMEA table containing a list of functions associated with a

predefined list of generic failure modes, which must be completed and validated by a safety expert. Functions are assigned to components, generating a logical architecture modeled in BDD. A Python program generates a Component FMEA table similar to the Functional FMEA table by adding a component column that groups functions of the Functional FMEA table. The generated table has to be completed and validated by a safety expert. The FMEA details are integrated into the system model with custom stereotypes. A FT corresponding to a generic top event failure is generated from IBDs based on back-tracking the flow from the output to the inputs. The paper proposes to use the Component FMEA results to build a FT for a specific failure mode without detailing the procedure. Generated FTs are then completed and validated by a safety expert. The result of FTA is either an image or a file based on the Open-PSA, which can be explored by an XFTA engine. The proposed tool is implemented in Python and third-party libraries.

In (Joshi et al., 2007) the generation of FT from Architecture Analysis & Design Language (AADL) models is proposed instead of UML-based models. The system failure behaviour is textually annotated in the AADL models using an Error Annex. The annotated AADL models are transformed into a generic format model of FT, to be exported to a FT analysis tool (the commercial Computer Aided Fault Tree Analysis (CAFTA) tool was selected).

In (Feiler and Delange, 2017) AADL is used to model SCS architecture and specify the failure behaviour textually using Error Model Annex V2 (EMV2). The Open Source AADL Tool Environment (OSATE) is used to build and annotate the AADL models. A FT is generated from the annotated AADL models based on the data flow in the system architecture model. The FTs are visualized and analyzed using an in-house developed open source tool called EMF-based Fault Tree Analysis (EMFTA). EMFTA is integrated into

the OSATE and is provided as a standalone plugin for Eclipse (The Eclipse Foundation, 2017b). EMFTA is also used in this thesis to perform fault tree analysis.

In (Oliveira et al., 2014) safety analysis of Software Product Lines (SPL) is proposed using an MDSE methodology based on single product concepts, combined with SPL concepts. Existing tools are used for modeling and analysis to obtain FTAs and FMEAs. The SPL feature models are created using MATLAB/Simulink, then HAZOP (HaZards & Operability) technique are used with the HiP-HOPS tool to perform hazard analysis of the model. HiP-HOPS is used to annotate the model with failure logical expressions. Once a product is derived, HiP-HOPS can be used to generate FTAs and FMEAs for that product.

In (Wang et al., 2010) FMEA and FTA models are generated from Little-JIL (LASER, 2017) models. Little-JIL is a process definition language modeling a task's process in terms of agent activities and their use of resources. This paper proposes to automatically generate FMEA then FTA based on an Artifact Flow Graph, which shows how an artifact is dependent on another artifact. In an earlier work (Chen et al., 2006) proposed to generate a FT model from Little-JIL models.

In (Papadopoulos et al., 2011) FTA and FMEA are performed using HiP-HOPS from Modelica (Modelica Association, 2017) based models. The system is modeled with SimulationX tool (ESI ITI GmbH, 2017) using CBM. Failure information of each component is manually annotated on each of them. An expression representing the possible failure output of a component is based on the combination of internal failures and input failures. During analysis, the model is traversed from output ports/output failure toward the input ports generating and connecting component-level FTs. An FMEA table is generated from the FT.

In (D'Ambrogio et al., 2002) FT are generated from UML SDs and DDs for reliability analysis. Each sequence diagram models a scenario, which is converted by a "mapper" to a fault tree. The mapper first decomposes the SD into its basic constructs, which can be of the following types: synchronous, asynchronous or local call, branch, loop, object creation/destruction, then it converts them to the corresponding FT constructs. Scenario FTs are combined into a system-level FT. Failure information, such as failure rate, is manually added to the global FT. A tool prototype was developed to implement the proposed method. The tool utilizes the XMI format, and it produces FT tree in XML files to be feed to an existing analysis tool.

In (Pai and Dugan, 2002) the generation of an extended type of fault trees called Dynamic FT (Dugan et al., 1992) from UML models is proposed. UML structure diagrams are used to model the system, specifically class, object, and deployment diagram. Dynamic FT is an extension of the FT to allow for the modeling of sequence and functional dependencies and spares. A proposed UML profile is used to annotate the source models with required reliability information. The system is modeled using the commercial tool Rational Rose. A custom script is used to generate the Dynamic FT, which is then fed to the Galileo tool (Dugan et al., 1999) for analysis.

In (Towhidnejad et al., 2003) FT generation guidelines are proposed from UML sequence diagrams for the validation of OO software design. It also investigates the applicability of UML SMD and AD as source models, as well.

(Xiang et al., 2011) proposes the generation of custom reliability models and FTs from SysML models. The system is modeled using SysML's IBDs and SDs stereotyped with additional reliability information. A custom tool implemented by the authors is used to

obtain reliability models specified in Maude (Clavel et al., 2017), which is a textual executable algebraic formal specification language. The reliability models are translated into Maude-based FT models.

In (Zornoza Moreno, 2018) an approach within the CHESS platform for systems modeled using CHESS ML is proposed. The model is examined using another tool called Concerto-FLA, which injects failures at inputs and calculates the system failures at the outputs. The results of this tool are used to obtain FTA models. This work presents the latest relevant standards for different domains that require and guide the use of FTA. The FTA analysis tool used in this work is the same tool we are using, EMFTA.

#### 2.2.1.2.1 Component Fault Tree (CFT) and extensions

An extension of FT called Component Fault Tree (CFT) was proposed in (Kaiser et al., 2003). CFT extends the FT model by adding the concepts of components, input ports, and output ports. This approach highly couples the concept of the component from CBM with the fault tree model. This extension has been initially proposed by the working group ESSaRel (Embedded Systems Safety and Reliability Analyser) in (Kaiser et al., 2003), followed by various extensions and enhancements presented in (Adler et al., 2010; Domis and Trapp, 2008; Grunske and Kaiser, 2005; Höfig et al., 2015; Höfig and Zeller, 2016; Kaiser et al., 2007; Mohrle et al., 2015). This approach uses XMI files for exploring the models using generic modeling languages.

In (Kaiser et al., 2003) the FT, as a concept and a model, is extended by adding concepts from CBM, mainly, components, their interfaces and interface connections. So, the fault trees are modeled per component and can contain subcomponents. Each component has

interfaces of possible failure modes going out of the component or coming into the component. These interfaces are called input ports and output ports. The CFT is transformed in Binary Decision Diagram for analysis. The modeling and analysis of CFT are only supported by an in-house developed proprietary tool called UWG3.

In (Grunske and Kaiser, 2005) is presented an application of the CFT proposed in (Kaiser et al., 2003) to COTS components for CBM. It starts with the use of the IF-FMEA technique to build a CFT for each component that is not composed of other components. Then, an algorithm is provided to build CFT for composite components, which results in a CFT at the system level, a regular fault tree. The modeling and analysis of the CFT use an in-house developed proprietary tool called UWG3.

In (Kaiser et al., 2007) an extension to the CFT introduced initially in (Kaiser et al., 2003) is proposed by adopting concepts from Statecharts (Harel, 1987) and Petri Nets (PN) within the CFT. The generated model is called State/Event Fault Trees (SEFT). SEFT is generated at the component level using the ESSaRel tool, which visualizes the model and performs some analysis on it. For the system level, the SEFT are translated into Deterministic and Stochastic Petri Nets (DSPNs) (Ciardo and Lindemann, 1993) for formal analysis.

In (Domis and Trapp, 2008) functional system models using MATLAB/Simulink are used to derive failure behaviour models. Once the functional models are generated, the IF-FMEA is applied on the input and outputs using guide words to search for failure modes, causes and effects. These guide words are based on the respective application domain. A CFT is built using the results of the IF-FMEA. This work utilizes the work of (Kaiser et al., 2003) in terms of CFT. A proprietary tool called ComposeR is used to derive the system FTs by composing the CFTs. The FT analysis is performed either using ESSaRel or Fault

Tree+. A sample walkthrough was provided in the paper for the generation of the IF-FMEA and the CFT without presenting the underlying algorithms.

In (Adler et al., 2010) another extension of the work on CFT introduced in (Kaiser et al., 2003) is presented. The paper proposes to model the CFT using UML, by defining a metamodel comprising CFT concepts as a Domain Specific Language (DSL). Then the argument of using more than a single DSL to model various aspects of the system disappears, as both the software architecture and the fault trees are represented in UML. An integrating approach based on refactoring and harmonization to integrate the two DSL, CFT, and ArchDSL, is also shown.

In (Höfig et al., 2015) an extension of the CFT from (Kaiser et al., 2003) is proposed by modeling the system in two layers, one software and the other hardware, while the original CFT considers a single layer. In this work, a new dependency relation is proposed to propagate the failure in between these layers' components, which then it can be captured in CFTs.

In (Mohrle et al., 2015) an extension to the CFT from (Kaiser et al., 2003) is proposed, which is focused on connecting the failure modes represented by failure ports in a Component Based Models (CBM). Each of these ports split into either an output failure port representing a top event failure for a component or an input representing failure propagation from another component. They propose to connect the outputs of components to the inputs of propagated-to components based on their names. For each pair of connected ports, an output port and an opposite input port, a scoring function measures how similar are the events names. This is supported as a plugin for MagicDraw UML (No Magic, Inc., 2017).

In (Höfig and Zeller, 2016) a State Aware Fault Trees Analysis (SpARTA) is proposed, which is extending the Component Fault Tree (CFT) introduced in (Kaiser et al., 2003). The challenge tackled in this paper is that a system can behave differently depending on the state in which it is currently operating. Hence, a component fault might be caused by different events based on the operational state of the component.

#### 2.2.1.3 FMEA analysis

In (Bowles and Wan, 2001) a method for applying FMEA to the software of small embedded systems is proposed. The work defines four types of FMEA based on functionality: 1) system-level functionality FMEA, 2) system functions FMEA, 3) software-software and software-hardware interfaces FMEA and 4) software variables details FMEA.

In (David et al., 2009) and (David et al., 2010) the use of a database to maintain a list of component types along with their dysfunctional behaviour model is proposed. The system is modeled using SysML (more specifically, the BDD, IBD, SD, and Requirements diagrams). The system XML file is analyzed using data translation techniques to explore the source model and create a preliminary FMEA table with generic failure modes, to be reviewed and completed by a human expert. A database of components' dysfunctional behaviours is maintained and checked while creating the preliminary FMEA to speed its generation utilizing previous experiences. The final FMEA report and the source model are used to manually create a formal model based on the AltaRica data flow language. The formal model is then analyzed, and the results compared to the non-functional requirements.

In (Flores and Malin, 2013) a tool to automate the systematic generation of FMEA is introduced. This paper mentions the mandate of using FMEA for most of NASA's products. This work was driven by the problem of inconsistent failure modes and their reuse along with the utilization of the Common Failure Modes (CFM). The proposed tool uses a standard CFM list providing a subset of such modes based on the system attributes selected by safety and/or system engineers, who consider only relevant failure modes. It also provides a list of common causes and effects for the failure modes. The engineer builds the system using the Master Equipment List (MEL) by selecting the desired components, which then can be modeled using a Reliability Block Diagram (RBD) which show their connections.

In (Höfig et al., 2014) the reuse of FMEA results for reusable parts when designing a system is discussed from an industrial point of view. The authors propose a metamodel for maintaining and reusing these results as an answer to issues that can arise due to inconsistencies in failure modes and effects. A tool is implemented based on the metamodel, associating failure modes with parts and assemblies, allowing for their management and reuse.

In (Deji, 2016) the generation of FMEA tables from UML models annotated with a UML profile is proposed. ADs and Composite Structure Diagrams (CSD) were annotated with failure mode information. Papyrus (The Eclipse Foundation, 2017a) is used to model the system in UML and annotate it. Emfatic (The Eclipse Foundation, 2017c) is used to create the FMEA metamodel. The Epsilon Transformation Language (ETL) (Kolovos et al., 2017) is used to perform the transformation from the annotated UML models to an FMEA

model. Extensible Stylesheet Language Transformations (XSLT) generates a table view of the FMEA model.

#### 2.2.1.4 Integration of NFP analysis in the development process

(Berardinelli et al., 2009) focuses on the integration of multiple NFP analyses (such as availability, reliability, and performance) in the software development process. The source models are expressed in UML and are annotated with information concerning the NFP of interest using MARTE and DAM. The paper discusses existing procedures for generating various analysis models from the annotated models. FT models are generated from annotated UCDs and SDs; Generalized Stochastic Petri Nets (GSPN) from annotated SMDs and DDs; and Queueing Network (QN) from CoD and SD diagrams.

(Mustafiz, 2010) proposes a methodology targeting the Requirements Elicitation phase for Reliability and Safety concerns. An extension of the use case diagram along with a textual use case model are proposed to add exceptions that correspond to the non-functional properties of interest, Safety and Reliability. Another extension is proposed for UML Activity Diagrams and Statecharts to model the exceptional use cases. Rules in plain English are used to manually transform the textual use-case specifications and use case diagram into activity diagrams and state charts. The proposed methodology allows developers to consider exceptional scenarios in the Requirements Elicitation phase.

(Montecchi et al., 2011) aims to provide a complete solution for modeling and verification of extra-functional properties of CBMs as part of an European project CHESS (ARTEMIS-JU-100022, 2017) based on OMG's MDA (OMG, 2014). One of the main contributions is the introduction of CHESS ML (CHESS Modeling Language) as an extension of UML,

SysML and MARTE to model dependability information within the source model, which is then transformed into an Intermediate Dependability Model (IDM) (Montecchi et al., 2011). The next step is to transform the IDM into a formal analysis model, Petri nets, which is analyzed with an existing analysis tool.

(Helle, 2012) proposes integrating safety analysis within a MDSE process where the system is modeled using SysML. Activity diagrams are used to model the system functions, where Actions represent functions. The Actions are allocated to blocks stereotyped with required failure information. Safety requirements are represented as failure use cases and are associated with Actions. An IBD is used to model components interconnections. A Java program called "SafetyAnalyzer" is implemented to analyze the model and to generate a minimal cut set for each safety use cases with representation in RBD and the evaluation of the model against the specified safety requirements.

(Ziani et al., 2012) proposes a Dependability Modeling Framework to facilitate modeling redundancy. The framework consists of a metamodel for modeling dependable system redundancy and fault tolerance mechanisms. The metamodel is realized with a UML profile that makes use of MARTE datatypes. The system is modeled using component diagrams and component sequence diagrams, which are annotated with the proposed profile.

(Grant and Datta, 2015) proposes the use of formal specification techniques to validate and verify the software models against the DO-178C (RTCA, 2011), the software considerations in airborne systems and equipment certification required by regulatory agencies such as the Federal Aviation Administration (FAA) in the USA. The proposed approach models the DO-178C in a visual manner, using UML models to improve comprehension. It is also proposed to transform the UML artifacts in each modeling phase

into Z-notation models. UML Class diagrams where specifically targeted to be transformed into Z-notation models, which are analyzed using the Z/EVES tool for verification and validation against the DO-178C.

(Malhotra and Trivedi, 1995) proposes the generation of Generalized Stochastic Petri Net (GSPN) and Stochastic Reward Net (SRN) from an FTA model. With the use of two case studies, they compare the model complexity of the generated GSPN and SRN models, showing that the complexity of the SRN model is lower. However, the models' solution complexity is the same.

(Gherbi and Khendek, 2006) proposes an MDA-compliant approach for performing schedulability analysis on UML models. To perform schedulability analysis, they propose a metamodel containing the required information using a specialized tool. The UML model is annotated with relevant schedulability information using the UML Profile for Schedulability, Performance, and Time Specification (SPT profile). The annotated model is then transformed into an instance model of the proposed schedulability metamodel. The transformation is realized using the ATL transformation language program.

### 2.2.2 Summary of the state of the art

This section summarizes the works in literature that are most relevant to this thesis, based on a set of comparison criteria presented in Table 2-1. Table 2-2 contains a comparison between the selected approaches and the approach proposed in this thesis.

**Table 2-1: Comparison criteria**

| Criteria | Description |
|---|---|
| System model language | The language used to model the system to be analyzed (Optionally, the diagram types are indicated). |
| Model extension | Use their own extension or a standard one to add safety information to the system model.<br><br>Notation: Std: Standard, Cust: Custom. |
| System modeling tool | Tool availability for building the system model.<br><br>Notation: OSS: Open Source Software, PS: Proprietary, NA: Not available. |
| Analysis model | The model used to perform NFP analysis of the system. |
| Analysis Tool | Availability of an analysis tool (i.e., solver for the analysis model)<br><br>Notation: OSS: Open Source Software, PS: Proprietary Software, NA: Not available |
| Failure behaviour | Modeling failure behaviour of the system consisting of multiple failures. |
| Automation | Specifies whether the transformation approach from the system model to analysis model is automated.<br><br>Notation: Y: Yes, N: No, P: Partial |
| Feedback | Whether the approach feeds back the results obtained from the analysis model to the system model. |
| Ease of adoption | It is easy to adopt the approach or considerable effort is required. |
| Model oriented transformation | The use of model-oriented language for the transformation from the system model to the analysis model as opposed to general purpose language. |
| Safety NFP | Whether the NFP analyzed by approach are related to Safety. |
| Completeness of the generated analysis model | Is the generated analysis model complete or manual intervention is required. |

**Table 2-2: Methods comparisons**

| Method ref | System model language | Model extension | System Tool | Analysis model | Analysis Tool | Failure behavior | Automation | Feedback | Ease of adoption | Model oriented transformation | Safety NFP | Completeness of the generated analysis model |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **[Mhenni et al. 2018]** | SysML: AD, BDD, IBD | Cust | NA | FMEA, FT | PS | N | P | Y | N | N | Y | N |
| **[Feiler and Delange 2017]** | AADL | Cust | OSS | FT | OSS | Y | Y | N | N | N | Y | Y |
| **[Deji 2016]** | UML: AD, CSD | Cust | OSS | FMEA | -- | N | Y | N | N | Y | Y | Y |
| **[Mohrle et al. 2015]** | UML | Cust | PS | FT | PS | N | P | -- | N | N | Y | N |
| **[Wu et al. 2015]** | UML: CoD | Cust | PS | -- | -- | N | P | Y | N | -- | Y | N |
| **[Yakymets et al. 2015]** | SysML: BDD, IBD | Cust | OOS | AltaRica, FT | PS | N | Y | N | N | N | Y | Y |
| **[Höfig et al. 2015]** | SysML | Cust | NA | FT | -- | N | P | N | N | N | Y | N |
| **[Grant and Datta 2015]** | UML: ClD | OSS | NA | Z-model | OSS | N | Y | N | N | Y | Y | Y |
| **[Gómez-Martínez et al. 2014]** | UML: SD, CoD | Cust | NA | GSPN | PS | N | P | N | N | N | Y | N |
| **[Zhao 2014]** | UML: UC, SD, CSD | Std | OOS | FT | OSS | N | Y | N | Y | Y | N | N |
| **[Kim et al. 2012]** | UML | Cust | NS | FT | N | N | P | N | N | N | N | N |
| **[Helle 2012]** | SysML: AD, IBD | Cust | PS | RBD | -- | N | Y | N | N | N | Y | Y |
| **[Xiang et al. 2011]** | SysML: AD, IBD | Cust | PS | FT | -- | N | Y | N | N | N | N | Y |
| **[Montecchi et al. 2011b]** | CHESS ML | -- | PS | PN | PS | N | Y | N | N | N | N | Y |
| **[Papadopoulos et al. 2011]** | Modelica | -- | PS | FT, FMEA | PS | N | Y | Y | N | N | Y | Y |
| **[David et al. 2010]** | SysML: BDD, IBD, SD, Req | Cust | PS | FMEA, AltaRica Data Flow | -- | N | N | N | N | N | N | N |
| **[D'Ambrogio et al. 2002]** | UML: SD, DD | -- | NA | FT | NA | N | Y | N | N | N | N | N |
| **Proposed approach** | SysML: BDD, IBD, SM | Std | OOS | FT, FMEA | OSS | Y | Y | Y | Y | Y | Y | Y |

From the comparison Table 2-2, it can be seen that the approach proposed in this thesis handles areas that were not covered by other works, providing a more comprehensive approach for safety analysis based on more than one technique, applied to a source model expressed in standard language/extensions and built with open source tools.

# 3 Chapter: High-Level Overview

This chapter gives a high-level overview of the proposed approach for performing an automatic model transformation from SysML to Fault Tree models and their integration in the safety analysis of the system under development.

## 3.1 Proposed approach

This section presents the activities of the proposed approach for integrating fault-tree analysis in the development process of safety-critical systems. It is assumed that the elicitation and analysis of the requirements are performed paying close attention to identifying and handling the component and system failures. One such approach is described in (Mustafiz, 2010) that directs toward obtaining the system failure information at the requirements phase.

In the design phase, a system architect synthesizes a system's structural and behavioural characteristics as a system model. In this research, we selected SysML to build the system model, for the reasons discussed in section 2.1.1. The proposed approach is then applied to the SysML model to perform safety analysis and to check whether the system satisfies its safety requirements. The approach activities are as follows (see Figure 3-1):

- *Build / Annotate source model:* Building a system model based on the requirements elicited in previous phases, where the system failure information is combined within the system model. We use the Modeling and Analysis of Real-Time Embedded Systems (MARTE) and Dependability Analysis and Modeling (DAM) profiles to add failure information to safety-relevant model elements. In this work, the open source Papyrus

tool was used for building a SysML system model annotated with relevant safety information, but other UML tools conforming to the OMG UML standard may also be used.

- *Transformation to CFT*: using model transformation techniques, the SysML source model is transformed into Component Fault Tree models (CFTs) at the component level. This activity produces a set of CFT analysis models (a fault tree corresponding to each component) and a Trace model of the transformation (containing tracelinks between the mapped source and target model elements). We used the Epsilon Transformation Language (ETL) for implementing this transformation step.
- *Transformation to SFT*: this transformation step takes as input the following models: a) the set of CFT models produced in the previous step, b) the SysML source model, and c) the CFT generated trace model. It transforms the inputs into a System-level Fault Tree (SFT) model representing the system-level failure, by composing the CFTs according to the interconnection of the component instances in the system. We used the Epsilon ETL language for developing this transformation step.
- *Fault Tree Analysis (with EMFTA)*: perform analysis of the generated SFT analysis model, obtaining results that describe the safety properties of the system under consideration. We use the existing open source EMFTA tool to perform the fault tree analysis.
- *Feedback of results:* Update the SysML source model with the FTA analysis results obtained by following the tracelinks between the source and target elements in reverse. This activity considers the previously generated artifacts. We use Epsilon EOL language to perform this task.

- *Generate FMEA model:* This is a model-to-text transformation that generates an FMEA model as a table, taking as input the annotated SysML source model and the synthesized FTA models. We use the Epsilon EGL language to implement this transformation.

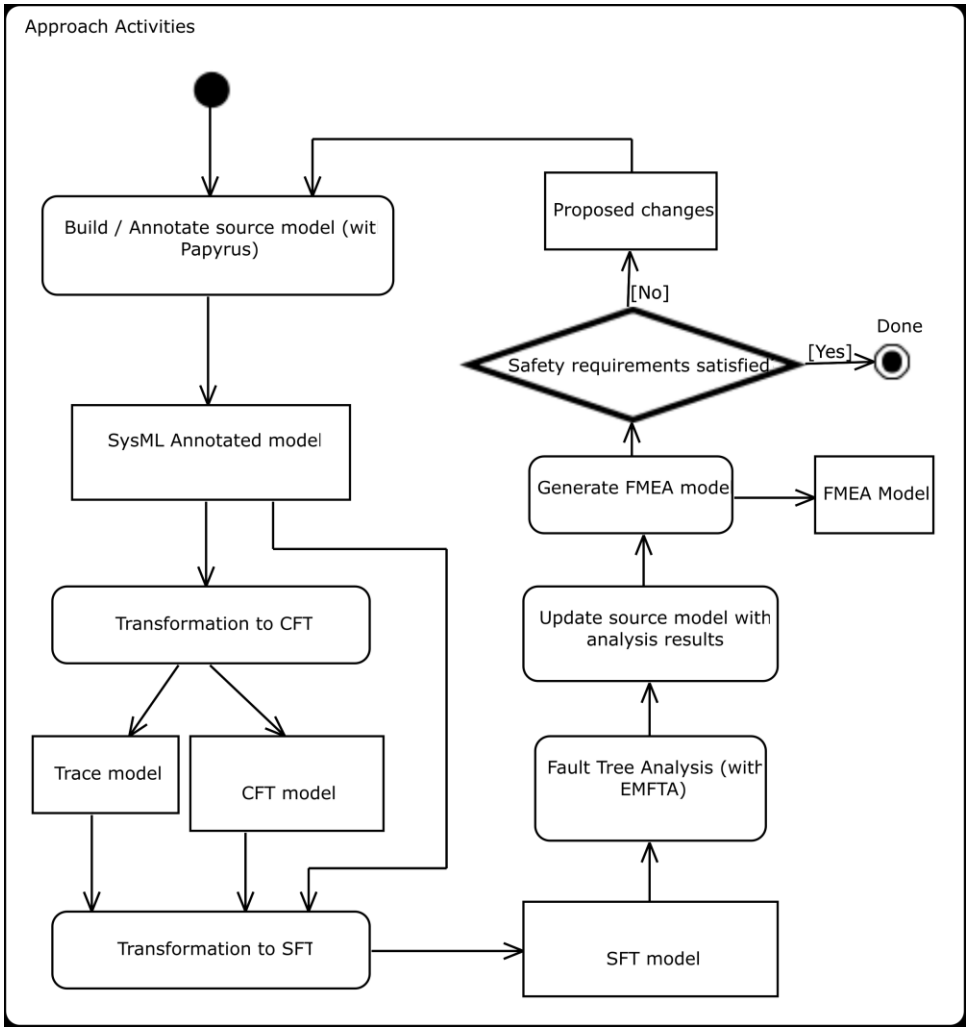



**Figure 3-1: High-level overview of integrating safety analysis in MDE**

- *Safety requirements satisfied?* The results of the system FTA are then compared with the safety requirements:
    - *If yes,* the designed system satisfies the requirement, and SA is completed.

- *If no,* changes for improvement are found and applied to the SysML system model, then the proposed process is repeated on the modified system model.

## 3.2 Source model

This subsection discusses the system source model in the proposed approach. The SysML relevant model elements will be briefly introduced.

After research and analysis of existing modeling languages, SysML (OMG, 2015a) was selected as the language for modeling the system under consideration. A specific set of SysML elements that were found sufficient for modeling the required information permitting the safety evaluation of a system are discussed here. A comprehensive SysML description can be found in the OMG SysML specification (OMG, 2015a).

### 3.2.1 Structure View

Figure 3-2 depicts the SysML elements used to model a system structure for safety evaluation with the proposed approach.

Block is the basic structural construct in SysML, holding a set of features describing an element of interest. Features are both structural and behavioural. In the context of component-based modeling, blocks can represent components. The name of "block" stems from the need to model logical and physical decomposition and to specify software, hardware, and human elements. A block's properties may represent values, parts, and references to other parts.

SysML Block properties are classified as follows:

- *Port*, a property specifying the possible connections among blocks.

- *Part*, describing the composite aggregation of a block typed property.
- *Reference*, a block typed property that does not have composite aggregation.
- *Value*, a property typed by a Value Type.

As part of SysML's objective to have modular and reusable blocks, it was intuitive to support the specification of how the blocks connect and interact within a desired context. Blocks' connection is achieved, among others, through *Ports*, *Flow properties*, and *Item flows*.

*Port* in SysML has been classified into *Proxy port* (that relays the features of its owning block), and *Full port* (that has its features).

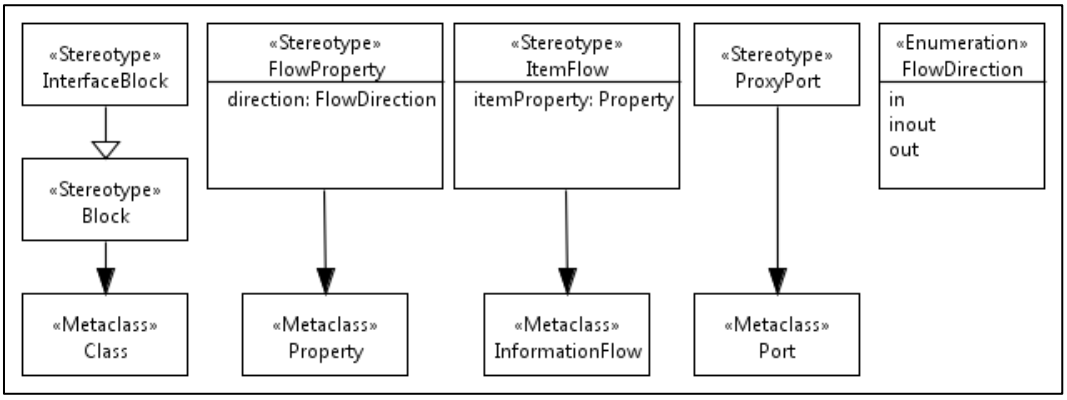



**Figure 3-2: An excerpt of OMG's SysML metamodel showing the elements used in the transformation**

SysML *Interface Block* is a specialization of *Block*, that is used to type SysML ports. Interface Blocks do not own behaviours including classifier behaviour, internal parts or methods.

SysML *Flow property* is used to specify what a *Block* can have flowing into it and out of it.

SysML *Item Flow* is used to specify what flows through a connector of blocks/parts in a specific context.

SysML *connectors* are of two kinds: *delegation* and *assembly*. Delegation connectors are used to pass the items received at a boundary port to the internal parts of its block, and the rest of the connectors are assembly kind.

*Package diagram* provides an organization of the model elements and also defines a namespace for the packageable elements.

*Block Definition Diagram* (BDD), is used to define system blocks, their features, and relations among each other including, associations, generalizations, and dependencies, such as modeling a system hierarchy or a classification tree.

*Internal Block Diagram* (IBD) is used to specify the internal structure of a block regarding its properties and their connectors.

Here are some guidelines for building/annotating the structure view of SysML models that will be transformed to fault trees:

BDD is used to model the system components as blocks.

- Part *association* models the composition relations between block instances (i.e., parts).
- The *DaComponent* stereotype from the DAM profile should be used to add the failure information of a specific block at this level. The *DaComponent* can annotate a block with information such as the *resource multiplicity*, the minimum number of resources required for the system to operate safely and the *occurrence probability* of the component's failure. If the component represents a spare, the *DaSpare* stereotype is applied instead, as it extends the *DaComponent*.

- A *software component* can have an *allocation dependency* relation with a *hardware component*, which is modeled using the *allocate* relation.
- A component can replace another component when it fails. The *substitute relation* can be used to model this relation.

*IBD* is used to model the internal structure of a *block* containing *parts*.

- The *internal parts* are instances of other blocks where they accept input from ports and produce output to ports. The use of *ports* allows for the design of modular and reusable blocks.
- All ports are of type *proxy port* and typed using *interface blocks*. A clearly defined way of connecting blocks and interact with them makes for a clear design. Item flows are used to connect these ports and represent what is exchanged over the connections.

### 3.2.2 Behaviour View

Figure 3-3 depicts the metamodel elements found sufficient for modeling system behaviour as needed to perform safety evaluation with the proposed approach.

As already mentioned, SysML is partly an extension of UML; considering that these elements have proven their usefulness in UML, they were inherited directly from UML into SysML. Behaviour modeling represents how structural modeled elements can change.

State Machine is used for modeling behaviour as a finite state transition system. States can be a simple state or composite state containing nested states. For the activity of modeling the behaviour, we propose the following guidelines:

- At the specified fine-grained level of component decomposition, each component has a SM as the block's classifier behaviour. The SM will represent normal and faulty behaviour.

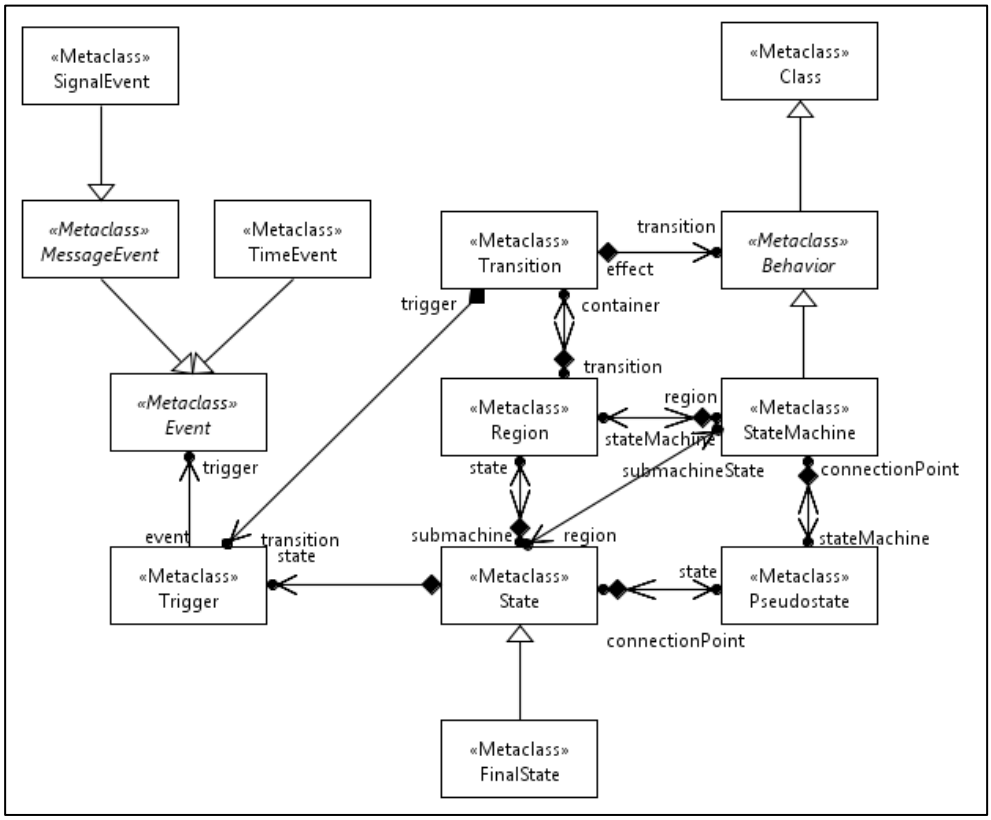



**Figure 3-3 Excerpt of UML/SysML metamodel related to state machines**

- A SM transition shall be triggered using either a signal event or time event. A signal event models the arrival of expected component input. Furthermore, the signal event is extended to model possible error transitions leading to failed states.
- A time event is used to signify the waiting for the arrival of specific component input.
- Each transition triggered by an error signal event or a timeout event is stereotyped using the DAM profile's `DaStep` stereotype and represents an error transition.

- An error transition shall have an effect, which can be sending a failure signal to its output port.
- Each state reached via a transition stereotyped with `DaStep` has to be also stereotyped with `DaStep` as a failure state.
- In the case where a component is expecting more than one input, this is to be modeled as a composite state, with concurrent regions. Each region should receive the expected input at each port.

## 3.3 System-Level FT target model

The target analysis model for performing safety analysis is the Fault Tree model (FT). For the thesis work, we selected the EMFTA tool (CMU-SEI, 2017), introduced in (Feiler and Delange, 2017) for performing visualization and analysis of FTs. The selection of EMFTA is based on the following rationale. It is an open-source software maintained and supported by SEI at CMU, running on top of the Eclipse platform (similar to the other tools Papyrus and Epsilon used in this thesis), and developed based on industry standards like (SAE-ARP4761, 1996). Other existing FTA tools are ill-suited because they are only commercially available or are not being maintained and supported. Being developed as an Eclipse plugin, EMFTA uses the Eclipse Modeling Framework, which provides the foundation for the other metamodels used in this work: the source SysML metamodel, the target FT metamodel and the transformation Epsilon metamodel. We can import all these metamodels directly, without modifications in order to generate and analyze fault trees. The FT metamodel used in the EMFTA tool is shown in Figure 3-4. It consists of the following elements with an example model instance visualized and presented in Figure 3-5:

- *FTAModel* is the top element in the model and is the container for events and gates that together form the fault trees. An FTAModel can contain multiple fault trees with distinct top/root events. The visualized fault trees in Figure 3-5 are modeled in an FTAModel element.

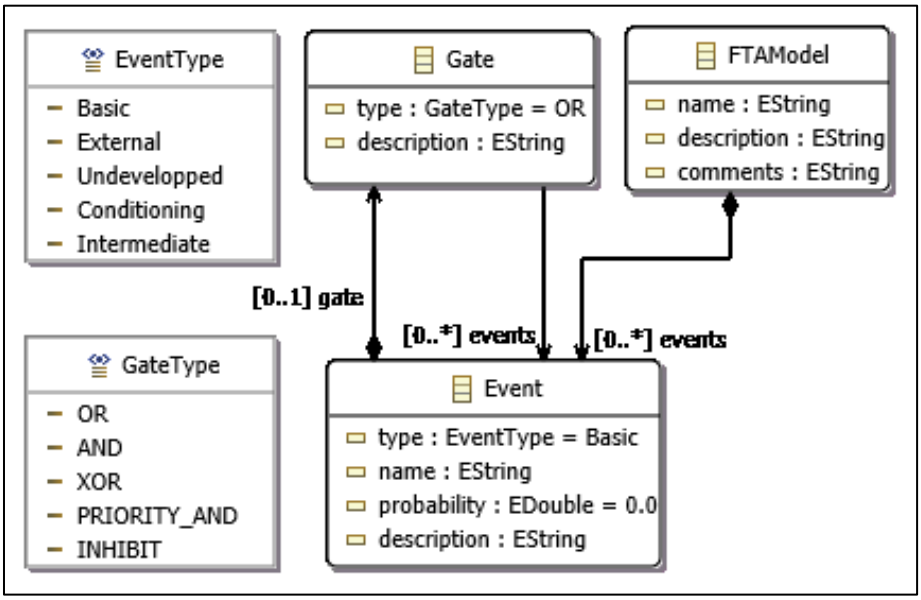



**Figure 3-4: SFT target metamodel**

- *Event*: one or more events are contained in an FTAModel instance. Each event may contain zero or one gate, which represents the logical gate producing the event. The properties of an event are *type* (enumerated type *EventType*); *name* (a String); *probability* of occurrence (a double); and *description* (a string). The values of the enumerated *EventType* are:

    1. *Basic*: the lowest level of identified events leading to the top event that does not require further analysis.

    2. *External*: regular events that occur during regular behaviour.

3. *Undeveloped*: events that have not been analyzed either due to the unavailability of information or because do not have any effects.

4. *Conditioning*: used to specify conditions or restrictions affecting the logical gates.

5. *Intermediate*: events that have been further analyzed and their immediate causing events have been identified.

The first four event types are sometimes referred to as primary events.

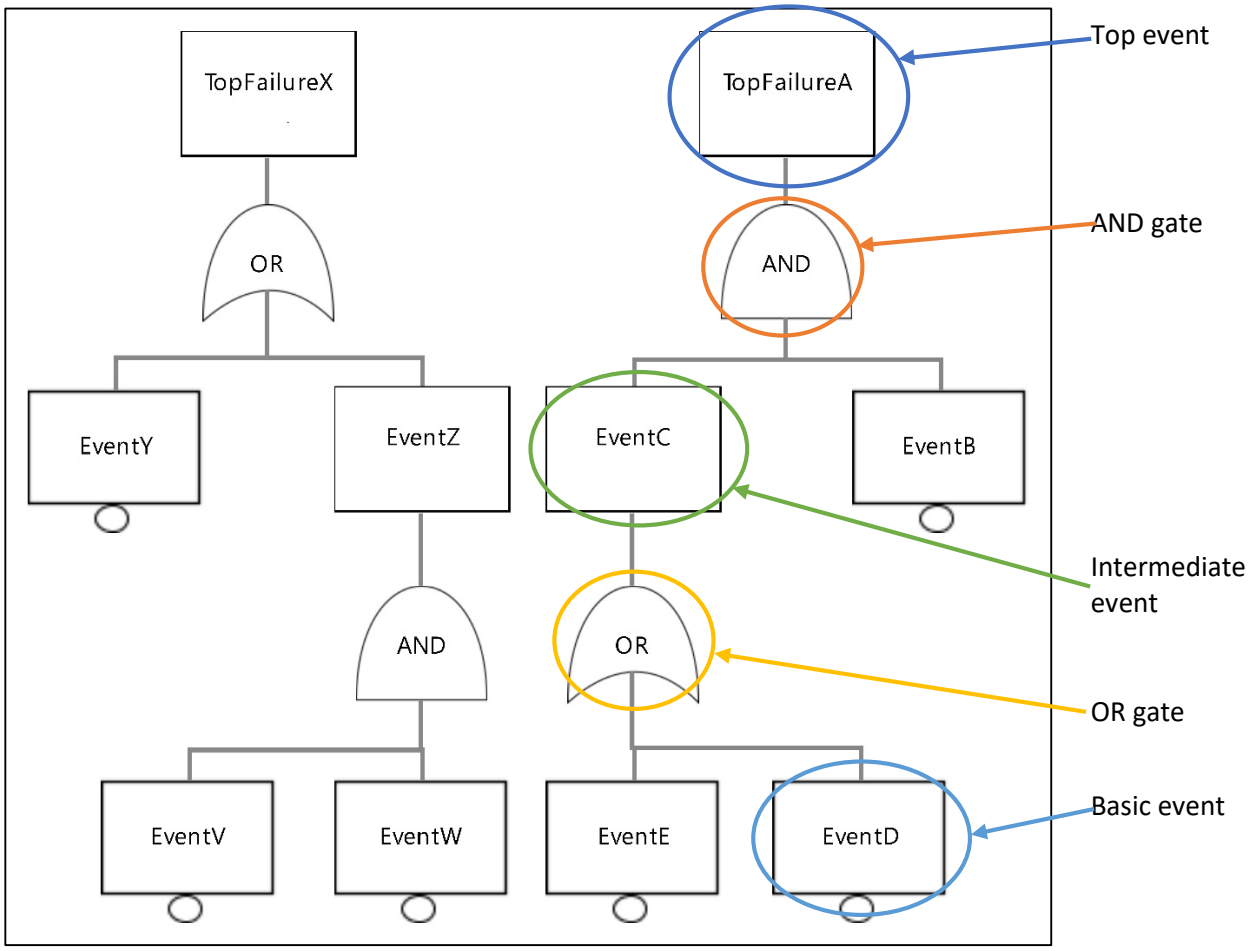



**Figure 3-5: An FTAModel with multiple trees**

- *Gate*: is used to link the causing events as input lower level events to the resulting event as an upper-level output event using Boolean logic. The property *GateType* has the following enumerated type values.
    - *AND*: all the input events must occur for the output event to occur.
    - *OR*: an occurrence of any input event can cause the output event to occur.
    - *XOR*: Exclusive Or, exactly one input has to occur for the output to occur.
    - *PRIORITY_AND*: The input events have to occur in a specific order for the output to occur. The order is specified as a conditional event.
    - *INHIBIT*: The input must satisfy a condition for the output event to occur. The condition is specified as a conditional event.

## 3.4 Component-Level FT target model

This section introduces a fault tree model at the block/component level as the first transformation step toward generating the system fault tree model. Such an intermediate model enables reuse and simplification of the generated analysis models at the block/component level. The generated CFT models can be maintained per block and only be generated when the block behavior is modified; this allows for minimizing the execution time when handling systems with a considerable number of blocks.

The development of the intermediate model was adapted from the CBM. It is applied in modeling the system of interest (the source model). Then, its applicability was considered for the analysis model targeted (the target model), which resulted in a Component Fault Tree (CFT) model. Figure 3-6 depicts the metamodel for CFT models.

This metamodel reuses the definition of the target analysis model of the approach, a FT, which is modeled using the EMFTA tool and explained in the following section. This model is provided as a visual representation of a single block type of failure.

The System is the primary container for all components and their Component-based FT. Each block/component defined in the system is added as a component which might contain other components and ports along with its FT failure behaviour. A port is contained within a component, and it has a direction attribute.

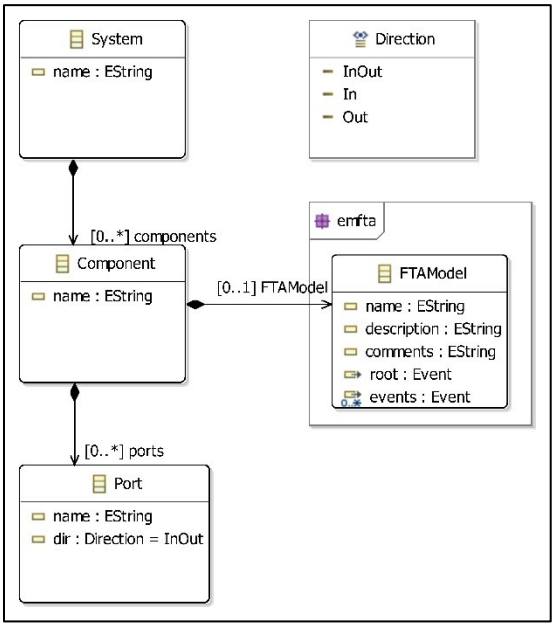



**Figure 3-6: CFT metamodel**

# 4 Chapter: Pattern-based Transformation Design

The adoption of SysML proved its benefits for modeling systems from different application domains. However, if we want to analyze a given non-functional property of a SysML model (such as performance, schedulability, safety, reliability, availability) we need to transform the SysML model into a formal analysis model (such as queueing networks, Petri Nets, fault trees). This chapter will discuss the transformation from SysML to fault trees at an abstract level, based on failure-mapping patterns and independent of the implementation language.

The proposed transformation from SysML to fault trees has been divided into two consecutive steps. The division into two transformations allows us to generate and save FT models for each component separately, producing modular, reusable analysis models, that can be later composed into a system-level FT model. If changes are necessary only in some components, then there is no need to rebuild everything from scratch, only those that have been changed. The transformation flow is depicted in Figure 4-1, which provides an overview of the proposed transformation steps. These steps are part of the overall approach presented in Figure 3-1.

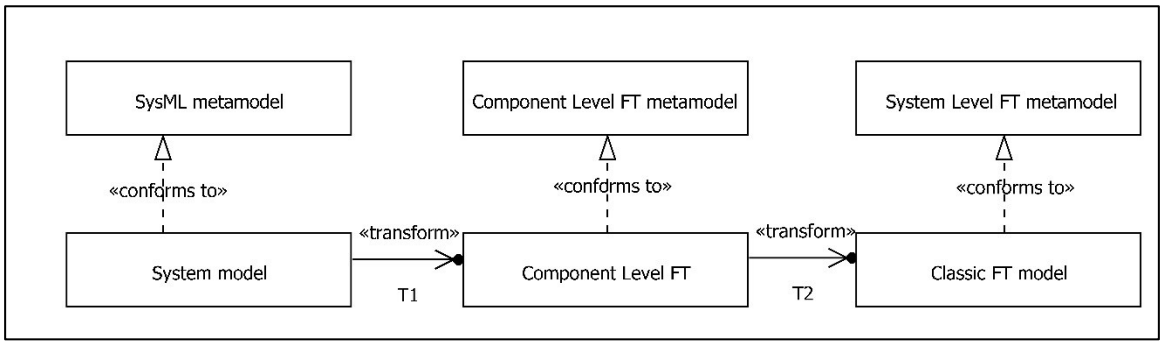



**Figure 4-1: Two model transformation steps**

## 4.1 Failure-Mapping Patterns

Mapping patterns play an important role in model transformations similar to the role of design patterns in software design. A mapping pattern represents a reusable solution to a mapping problem between a source and a target model. It has two parts: a) a group of inter-related source model elements (i.e., a source pattern); and b) a group of inter-related target model elements (i.e., a target pattern). A mapping pattern is composed of a source pattern mapped to a target pattern. When the source pattern occurs in the source model, it triggers the generation of the target pattern in the target model.

### 4.1.1 Behavioural Failure-Mapping Patterns

In this section, we present a set of mapping patterns that map the occurrence of a failure in the context of a SysML state machine to the occurrence of a failure event in the generated fault tree.

SMs provided as source patterns in the following subsections present only a fragment of a block's state machine, which is related to the failure-related behaviour. This is due to the fact that we are interested in describing the mapping of failure-related elements from the SysML source model to elements of the fault-tree target model. A complete state machine of a SysML block would contain more elements than presented in these figures, representing states and transitions for the entire behaviour of the block. Furthermore, in the following patterns, SMs will be presented without the normal initial states or other states and transitions leading to the presented SM extract. In some cases, only the failure-related transitions of a choice node are shown, even though a choice node may have other transitions related to the normal behaviour.

#### 4.1.1.1 Basic Pattern (1)

The pattern in Figure 4-2 models a transition triggered by an error signal event which changes the state of the component from a normal state to a failure state. The transition, the trigger and the failure state are stereotyped with *DaStep* signifying that they are a part of the component's failure behaviour augmented with failure information. We will refer to such a transition as an "error transition" when describing this and other failure-based mapping patterns.

In the corresponding target model, the following elements are created: a) an event modeling the error signal triggering the transition; b) another event that represents the entering of the failure state; and c) an OR-gate linking the two events. The OR gate is necessary because the syntax of the fault tree does not allow linking directly two events together. This target pattern is interpreted as follows: the occurrence of the error signal triggering the transition will lead to the undesired top event.

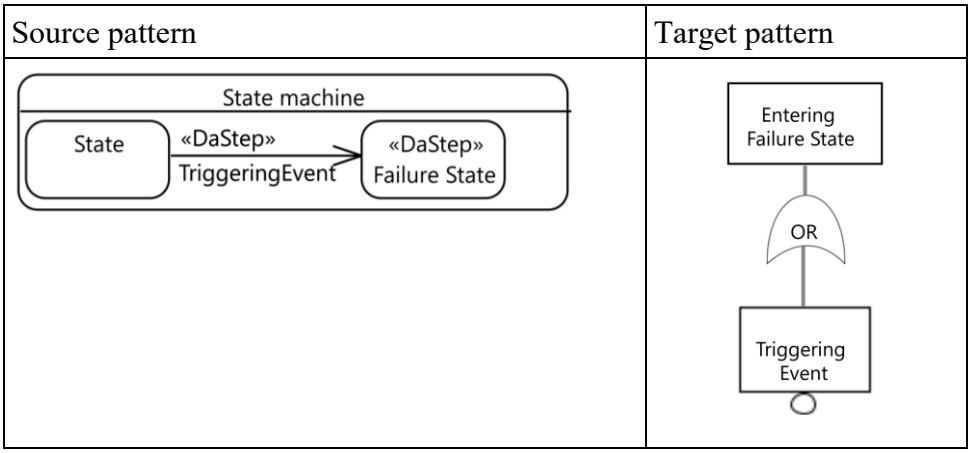



**Figure 4-2: Basic Pattern 1 mapping**

#### 4.1.1.2 Double Trigger Pattern (2A and 2B)

This pattern defines two variants of similar source model behaviours to be mapped to a target model Pattern with an OR gate. The first pattern models the behaviour of an error transition having two triggers that move the SM to a failure state. The second pattern models two error transitions each with a single trigger targeting the same failure state.

The target model comprises the following elements: a) two basic events, one for each error signal triggering a transition, b) an event for entering the failure state; and c) an OR-gate linking the two basic events to the top event. The target pattern is interpreted as follows: the occurrence of any of the basic events will lead to the occurrence of the undesired top event entering the failure state.

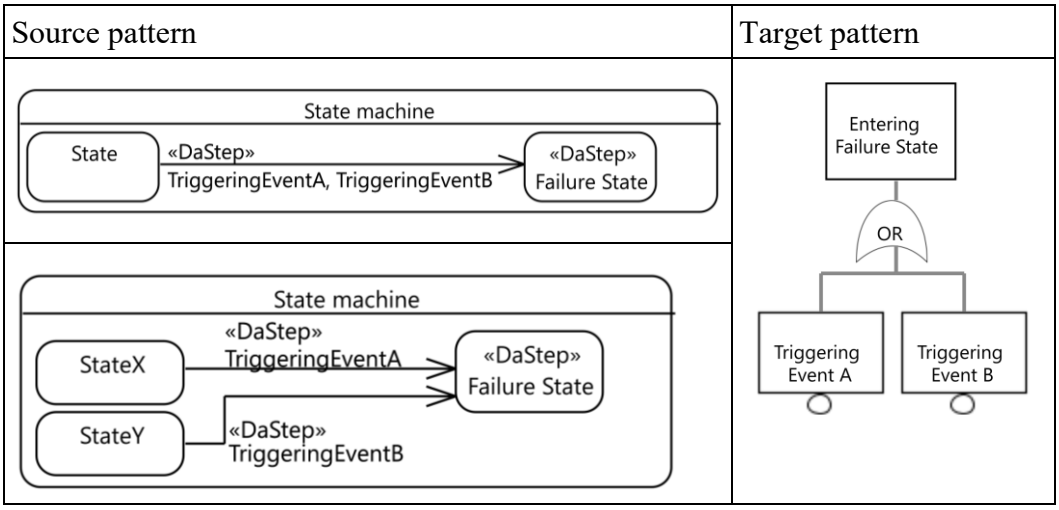



**Figure 4-3: Double Trigger Pattern (2A and 2B) mapping**

#### 4.1.1.3 Concurrent Execution Pattern (3A and 3B)

This pattern covers two similar pattern variants involving concurrent error events that are transformed into a target pattern containing an AND gate. In the first variant, the behaviour is split into two concurrent executions; each of them could fail. Both need to fail, for the two concurrent paths to join, and the SM to enter a failure state. Otherwise, the failure state

is not reached. The second pattern variant represents the same behaviour as the first using a different SM notation. If each concurrent state inside the composite state receives its corresponding triggering event, it ends by reaching its final state. If both concurrent states are completed by reaching their final state, the group transition out of the composite state is triggered, moving the SM to the failure state. Otherwise, the failure state is not reached. The target model comprises the following elements: a) two basic events, one for each error signal; b) an event for entering the failure state; and c) an AND-gate linking the two basic events to the top event.

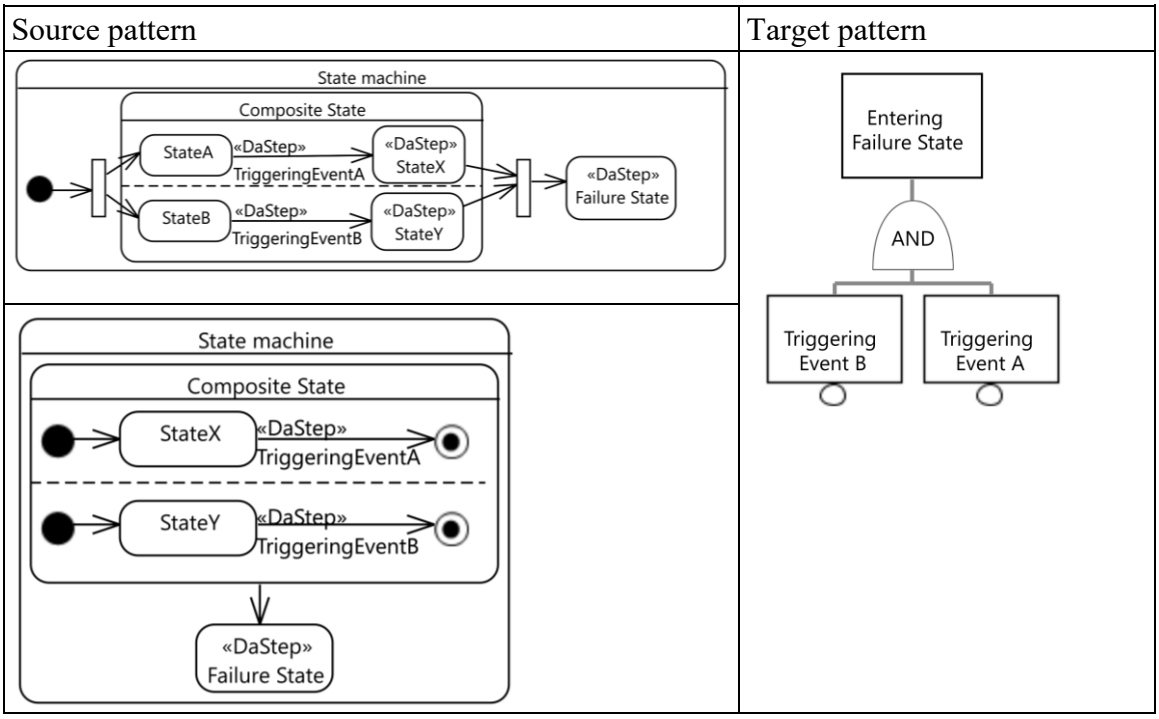



**Figure 4-4: Concurrent Execution Pattern (3A and 3B) mapping**

#### 4.1.1.4 Intermediate Failure Pattern (4)

This pattern models the occurrence of an error signal moving the SM to an intermediate failure state, which can receive two signals leading to total or partial failure states.

The target pattern contains two top events, which represent reaching the total and partial failure states, with a logical AND-gate for each. The initial triggering event is considered as a cause for both top events, while the following partial/total failure event is a cause for one related top event only.

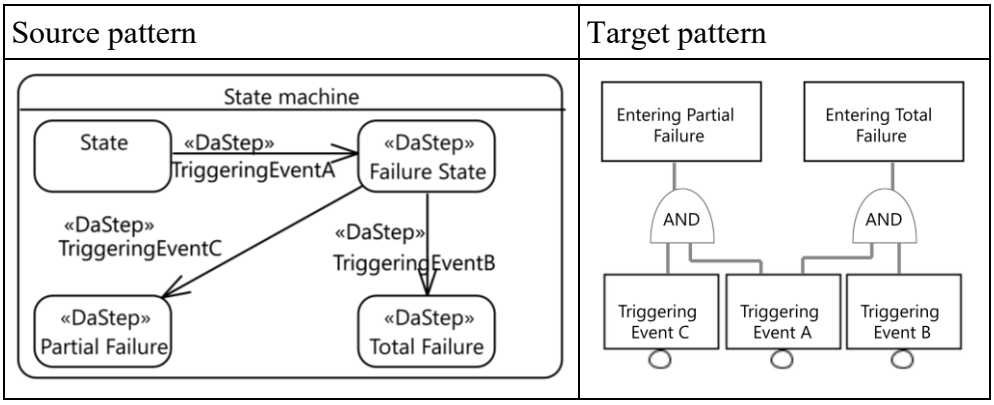



**Figure 4-5: Intermediate Failure Pattern (4) mapping**

#### 4.1.1.5 Retry Failure Pattern (5)

This pattern models the behaviour of retrying after a failure several times (twice in this case) before declaring the component failed. The target mapping of this pattern results in a top event reaching the failure state with a logical AND-gate joining a number of consecutive instances of the basic error event TriggeringEventA. The first instance is triggeringEventA[1], followed by two event instances triggeringEventA[2] and triggeringEventA[3] as allowed by the retrial threshold of 2. Other possible behaviours of the SM (e.g., when State receives a signal OK) do not lead to failure, therefore do not have any counterpart events created in the fault tree. Note that the transitions outgoing from the FailureRetryState are triggered by timeout events “at t” (where t is the time parameter).

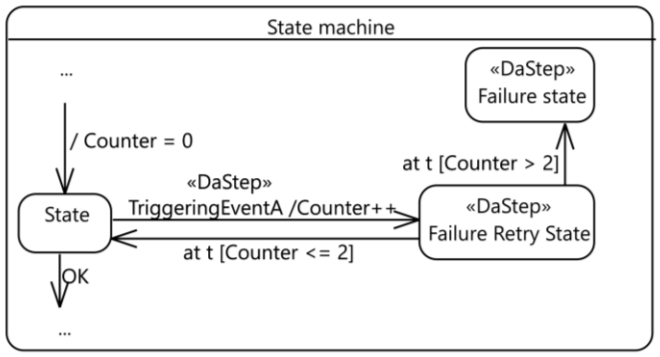
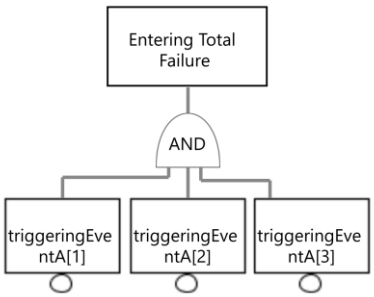
| Source pattern | Target pattern |
| --- | --- |
|  |  |



**Figure 4-6: Retry Failure Pattern (5) mapping**

#### 4.1.1.6 Single Failure Concurrent Execution Pattern (6)

This pattern covers a behaviour that splits into two concurrent executions contained inside a composite state. Each concurrent execution will complete by reaching its final state in two possible ways: either normally (triggered by a normal event) or failing (triggered by a failure event). When both concurrent executions reached their final state, a group transition out of the composite state (which has a choice node) is triggered. The choice node checks the guard condition [AFailed OR BFailed], and if true, a failure state is entered. Otherwise, the failure state is not reached.

The generated target model is comprised of the following elements: a) two basic events, one for each error signal; b) an event for entering the failure state; and c) an OR-gate linking the two basic events to the top event.

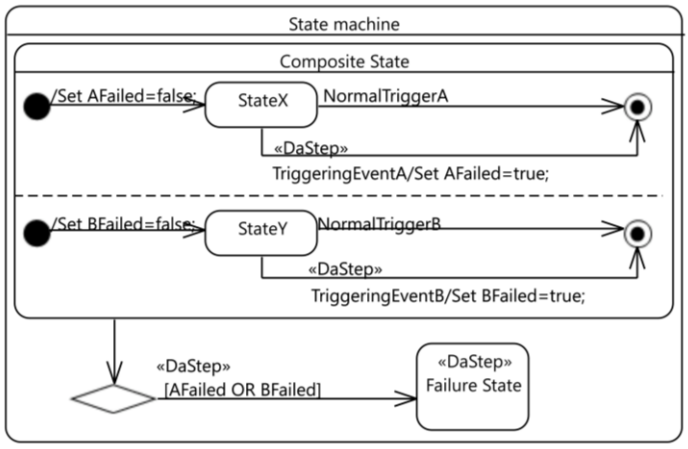
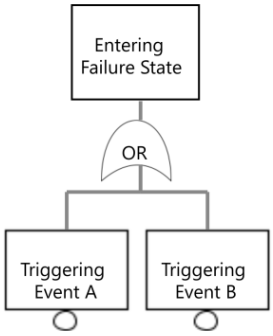
| Source pattern | Target pattern |
|---|---|
|  |  |



**Figure 4-7: Single Failure Concurrent Execution Pattern (6) mapping**

### 4.1.1.7 Multi Failure Concurrent Execution Pattern (7)

This pattern models a behaviour similar to Single Failure Concurrent Execution Pattern (6, with the difference that the guard condition checked by the choice node is [AFailed AND BFailed]. Both concurrent executions must fail for the component to fail. The generated target model contains: a) two basic events, one for each error signal; b) an event for entering the failure state; and c) an AND-gate linking the two basic events to the top event.

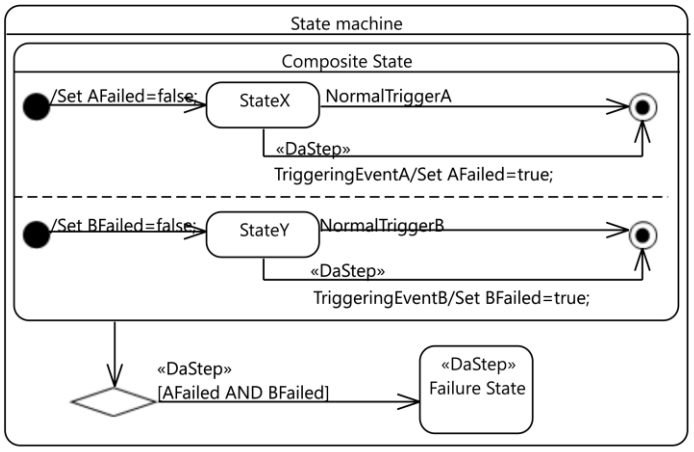
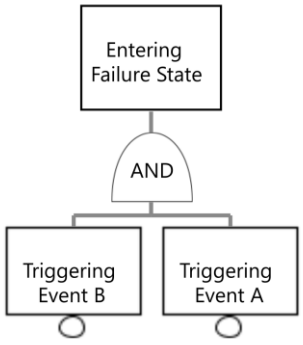
| Source pattern | Target pattern |
|---|---|
|  |  |



**Figure 4-8: Multi Failure Concurrent Execution Pattern (7) mapping**

#### 4.1.1.8 Exclusive Failure Concurrent Execution Pattern (8)

This pattern models similar behaviour to Single Failure Concurrent Execution Pattern (6, with the difference that only one execution has to fail for the component to fail. The guard condition checked by the choice node is [AFailed XOR BFailed]. The generated target model consists of a) two basic events, one for each error signal; b) an event for entering the failure state; and c) an XOR-gate linking the two basic events to the top event.

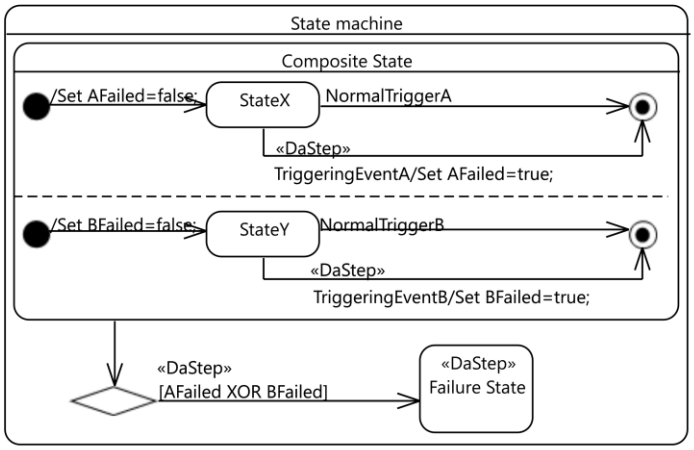
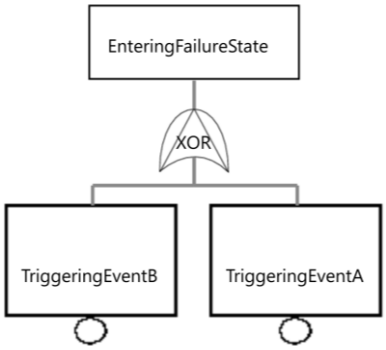
| Source pattern | Target pattern |
|---|---|
|  |  |



**Figure 4-9: Exclusive Failure Concurrent Execution Pattern (8) mapping**

### 4.1.2 Structural Failure-Mapping Patterns

A set of structural patterns for failure-mapping have been identified in different system models with SysML used in literature for safety analysis.

#### 4.1.2.1 Simple Dependency Component Failure Pattern (9A and 9B)

In this pattern, two source model representations are mapped to a similar target model. The first pattern shows that Component1 has an instance of Component2, which has failure information annotated with *DaComponent*. The second pattern shows an allocation relation of Component1 to Component2, which has its failure information identified and annotated.

In both patterns, the failure of Component2 should lead to the failure of Component1. The target pattern shows that the failure of Component2 causes the failure of Component1.

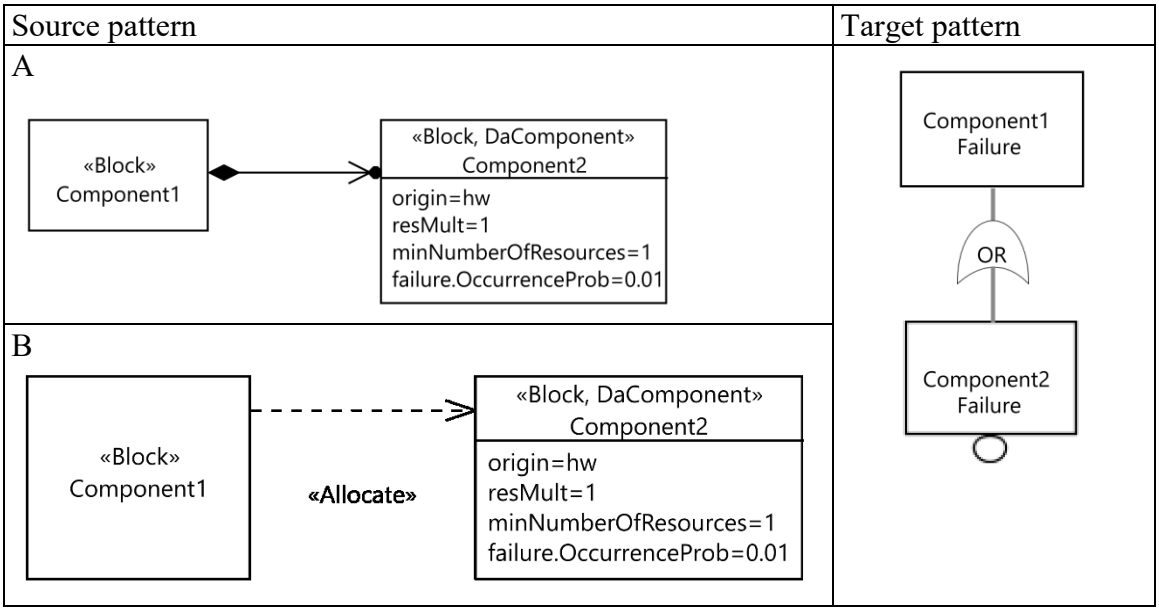



**Figure 4-10: Simple Dependency Component Failure Pattern (9A and 9B) mapping**

### 4.1.2.2 Single Instance Component Failure Pattern (10)

This pattern shows that ComponentA has an instance of ComponentB, which is annotated with failure information having resource multiplicity of two, and the minimum number of required instances is also two. This signifies that if any instance B fails, the dependent ComponentA fails, as well.

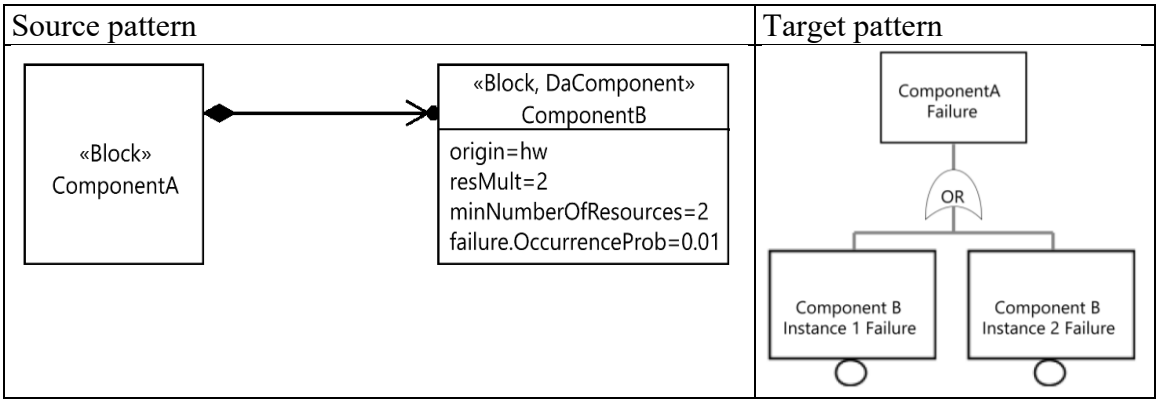



**Figure 4-11: Single Instance Component Failure Pattern (10) mapping**

The generated target pattern consists of: a) two basic events, one for each instance failure of ComponentB; b) an event for entering the failure state of ComponentA; and c) an OR-gate linking the two basic events to the top event.

#### 4.1.2.3 Multi Instance Component Failure Pattern (11)

This pattern represents that ComponentA has an instance of ComponentB with its failure information, which has two resource instances and at least one is required to function.

The generated target pattern consists of: a) two basic events, one for each instance failure of ComponentB; b) an event for entering the failure state of ComponentA; and c) an AND-gate linking the two basic events to the top event (i.e., both have to fail for the failure event of component A to occur).

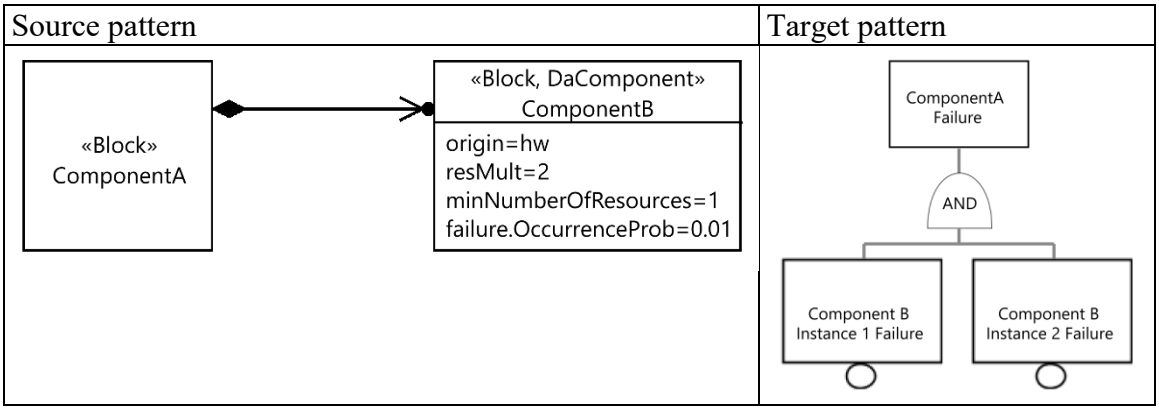



**Figure 4-12: Multi Instance Component Failure Pattern (11) mapping**

#### 4.1.2.4 Substitution Component Failure Pattern (12)

In this pattern, the *spare* and *substitute* relations are presented. ComponentA has an instance of ComponentB, whose failure information is annotated using *DaComponent*, and an instance of ComponentC, which has its failure information annotated using *DaSpare*.

Both B and C have one instance that is required for the system to function. The relation *substitute* indicates that componentC (which is declared as spare) can replace componentB.

The generated target model consists of an event for the failure of component B and another for the failure of component C; both are joined under an AND-gate meaning that both componentB and its substitute componentC have to fail for the event of the failure of componentA to occur.

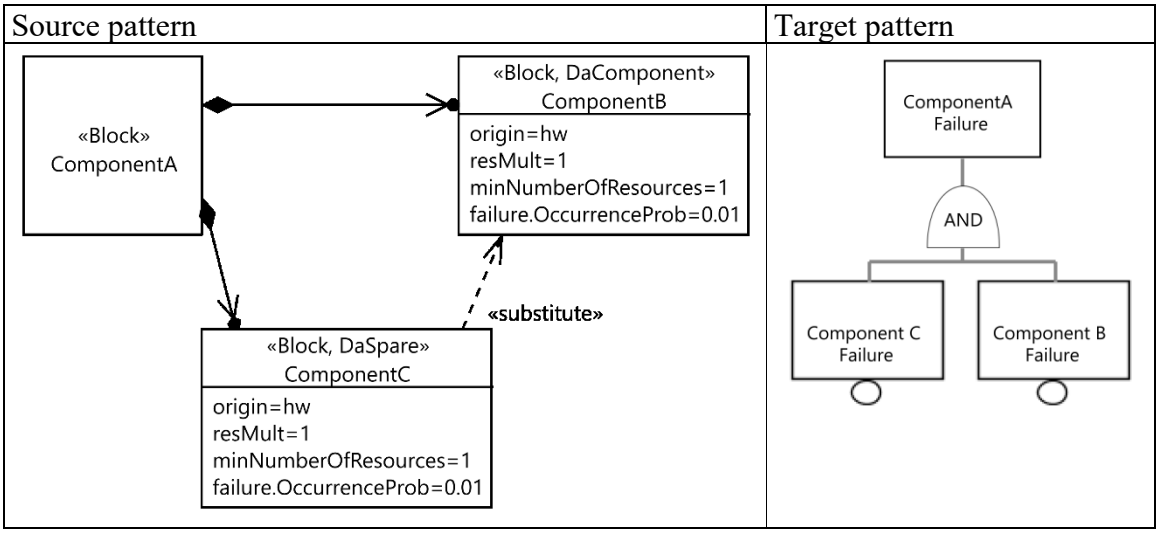



**Figure 4-13: Substitution Component Failure Pattern (12) mapping**

#### 4.1.2.5 Inter Component Failure Pattern (13)

The System block has Component1 and Component2. Both are connected using an item flow with the failure signal flowing from PortB of Component1 to PortA of Component2. The state machine of Component1 shows that upon receiving the Triggering Event, which causes the component to enter the Failure state, a Failure signal is sent to PortB. Furthermore, in Component 2 there is a transition triggered by the arrival of a Failure signal that moves the SM to the Failure state.

The generated target model has an event for the occurrence of the Triggering event in Component1 that causes the event representing Component2 failure.

This pattern is used to compose the component fault tree models generated in the first transformation step into a system fault tree model in the second step.

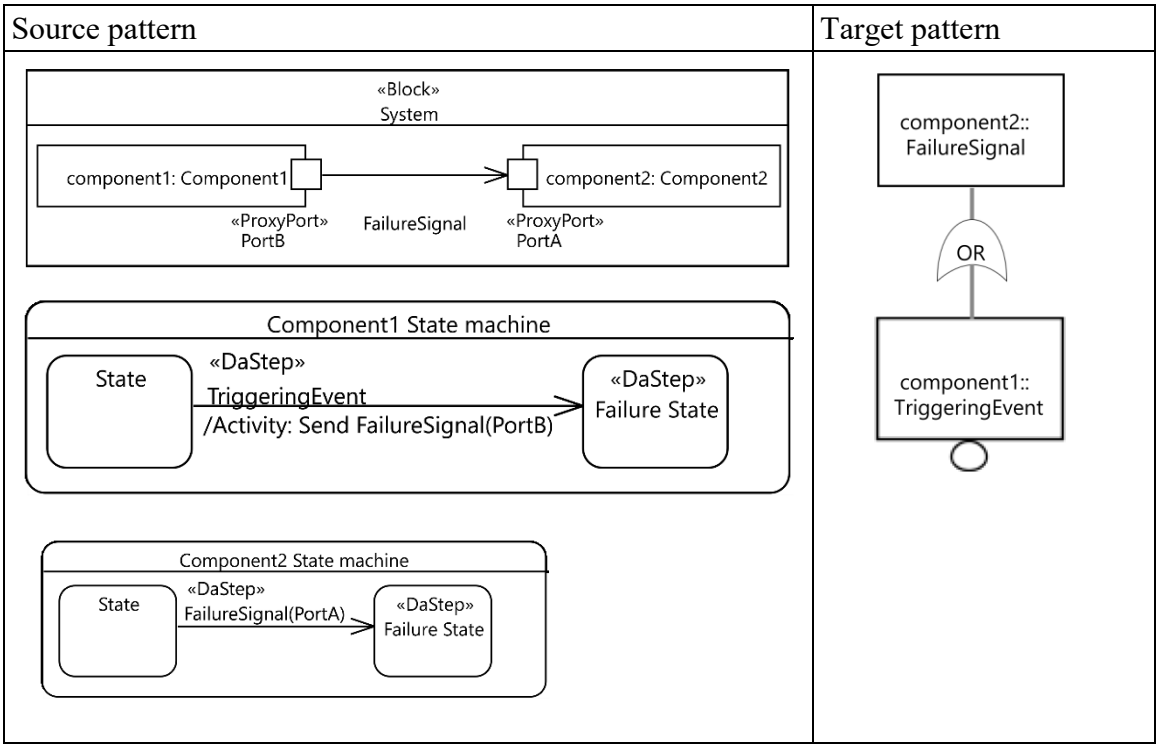



**Figure 4-14: Inter Component Failure Pattern (13) mapping**

## 4.2 Transformation Algorithm from annotated SysML to CFT models

In this transformation step, the failure behaviour at the component level will be transformed into a CFT model. A component can fail in different ways, which are modeled as multiple failure states in its SM. A CFT model may have more than one undesired top failure events (corresponding to entering the failure states), but they might share intermediate and basic events.

The transformation starts with realizing each component block defined in the source model into a component in the intermediate model. A block is examined to check for other elements that are to be transformed as well, such as its ports and behavioural model.

The port transformation considers the port type. The source model ports are represented by proxy port typed with Block Interface elements, containing Flow Properties with a direction attribute. So, the port type is used to identify its direction.

As discussed previously, a block behavioural model is specified using SMs, which should be developed with reference to the guidelines provided in the previous chapter. Based on these guidelines, a set of mapping/transformation rules have been identified. A block SM is transformed into an FT model of the block's component followed by the examination of the SM elements to synthesize the complete CFT.

In this subsection we introduce an algorithm developed to perform the transformation from annotated SysML to CFT. The presentation of the algorithm is independent of the transformation language used later for implementation. The transformation takes as input the system source model and produces a set containing the generated CFT models and a Trace model linking the source elements with the target elements they caused to be generated. The algorithm realizes the mapping of failure-mapping patterns introduced in section 4.1.1.

The algorithm starts by collecting all blocks that have failure behaviour modeled along with their normal behaviour in each block's state machine. Looping through all the selected blocks, an FTAModel is generated per block as a container for all synthesized events from its behaviour model.

Next, we obtain all failure states stereotyped with *DaStep* within the SM of a block. For each failure state, an FT event is generated and added to the container element with a default OR-gate. This is presented in Basic Pattern (1) in Figure 4-2, where entering the state called *Failure State* is transformed into the event *Entering Failure State*. It also applies to Double Trigger Pattern (2) in Figure 4-3 as the default gate for an event is an OR-gate. Additionally, if a failure state has a subsequent failure state (e.g., partial or total) with the transition triggered by an error signal, then, the event of entering the first state and the transition are set as an input to the gate of the event of entering the subsequent state, which is an AND-gate. This is presented in Intermediate Failure Pattern (4) Figure 4-5.

Next, the algorithm examines the transitions annotated with failure conditions. If a transition destination is a Join pseudo-state, which in turn has a transition going into a failure state, then an intermediate event is created followed by an AND-gate, whose output is the event indicating that the failure state has been reached. As shown in Pattern 3A in Figure 4-4, a Join node is reached from the concurrent executions of the composite state. These transitions are originating from failure states, that were reached by transitions triggered by error signals. These identified triggers are set as the inputs of the AND-gate, which are handled later in the algorithm as a general case of transitions triggered by error signals going into a failure state.

For a transition whose destination is a final state within a composite state, the composite state is examined if it has an outgoing completion transition targeting a failure state. If so, an intermediate event is created with an AND-gate, whose output is the event indicating that the failure state has been reached. Similar to the Join pseudo-state case, the inputs are added later in the algorithm. This handle Concurrent Execution Pattern (3B) in Figure 4-4.

If a transition going into a failure state has the effect of incrementing a failure counter and the target failure state has a subsequent failure state, then the event entering the subsequent failure state is obtained as a top event resulting from this transition trigger. The transition leading into the subsequent failure state has a counter guard that is evaluated to obtain the counter value. This is presented in Retry Failure Pattern (5) Figure 4-6.

If a transition targets a final state contained in a composite state, which in turn has a group transition targeting a choice pseudo-state, then its choice function is evaluated, and a target event is created based on the constraint. The inputs of the gate are added by the generic handling of the triggers of error transitions described below. This handles Figure 4-7, Figure 4-8, and Figure 4-9 in Single Failure Concurrent Execution Pattern (6), Multi Failure Concurrent Execution Pattern (7), and Exclusive Failure Concurrent Execution Pattern (8), respectively.

If a transition goes into a failure state, then the event of reaching the failure state is transformed into a top event in the target model. The triggers of the transition generate source events that are inputs for the gate of the corresponding top event. This is the general case of transitioning into a failure state.

Now, for each trigger of the transitions just described, such as in Basic Pattern (1), where the trigger of the *TriggeringEvent* transitions the SM into the *Failure State*, an event is created and is added as an input to the gate generated per specific case. For the transition described in Retry Failure Pattern (5), which has a failure counter guard, the generated events match the number of occurrences of the triggering event. This applies to all the triggers of the described patterns.

**Algorithm 4-1: Step 1: Synthesizing CFT model from a System behaviour model**

```
1: Input: System Source Model: System
2: Output: A set of Fault Tree models of the system behavior model at component level:
CFTModel, A trace mapping of the source model elements to their output model elements: Trace
3: blocks = getAllBlocksHavingStatemachinesWithStatesAndTransaitionsStereotyped(DaStep)
4: Initialize SetOfCFTModels(CFTModels)
5: Initialize Trace = map(sourceElements, CFTEvents)
6: for each block in blocks do
7:     stateMachine = block.behaviorModel
8:     cftModel = createFTAModel(block.name)
9:     SetOfCFTModels.add(cftModel)
10:    failureStates = stateMachine.getAllStatesStereotypedAndNotRetryFailure(DaStep)
11:    for each state in failureStates do                                  ▷ Pattern: 1A
12:        event = createFTEvent(state.name,  OR )
13:        if state.isIntermediateFailure() then                           ▷ Pattern: 4
14:            subsequentState = getSubsequentFailureState(state)
15:            parentEvent = Trace.get(subsequentState)
16:            parentEvent.gate =  AND
17:            parentEvent.gate.add(event)
18:        end if
19:        cftModel.events.add(event)
20:        Trace.put(state, event)
21:    end for
22:    transitions = stateMachine.getAllTransitionsStereotyped(DaStep)
23:    for each transition in transitions do
24:        target = transition.target
25:        Initialize parent
26:        if target.isJoinNode() then                                     ▷ Pattern: 3A
27:            parent = getIntermediateEvent(target,  AND )
28:        else if target.isFinalNode() and target.isContainedInCompositeState() and
target.containerState.outgoingTransition.target.isState() then          ▷ Pattern: 3B
29:            state = target.containerState
30:            parent = getIntermediateEvent(state,  AND )
31:            transition = state.outgoingTransition
32:            event = Trace.get(transition.target)
33:            event.gate.add(parent)
34:        else if transition.hasAnEffectOfACounterIncrement() and
target.hasASubsequentCompleteFailureState() and target.hasStereotype(DaStep) then ▷ Pattern: 5
35:            parent = Trace.get(getSubsequentCompleteFailureState(target))
36:            replicateCount = transition.constraintCounter
37:        else if target.isFinalNode() and target.isContainedInCompositeState() and
target.containerState.outgoingTransition.target.isDecisionNode() then   ▷ Pattern: 6, 7, 8
38:            outgoingTransition=target.containerState.outgoingTransition.target.outgoingTransition
39:            parentType = parseConstraint(outgoingTransition.constraint)
40:            decisionNode = target.containerState.outgoingTransition.target
41:            parent = getIntermediateEvent(decisionNode, parentType)
42:            event = Trace.get(decisionNode.outgoingTransition.target)
43:            event.gate.add(parent)
44:        else if target.isState() and target.hasStereotype(DaStep) then  ▷ Pattern: 1B, 2A, 2B, 4
45:            parent = Trace.get(state)
46:        end if
```

**Algorithm 4-2: Step 1: Synthesizing CFT model from a System behaviour model (cont.)**

```
47:          for each trigger in transition.triggers do                    ▷ Pattern: 1B
48:              eventsSet
49:              event = createFTEvent(trigger.event.name)
50:              parent.gate.add(event)
51:              eventsSet.add(event)
52:              for i = 0; i < replicateCount - 1; i + + do                ▷ Pattern: 5
53:                  event = createFTEvent(trigger.event.name + [i])
54:                  parent.gate.add(event)
55:                  eventsSet.add(event)
56:              end for
57:              Trace.put(trigger, eventsSet)
58:          end for
59:      end for
60: end for
```

Following is a companion to Algorithm 4-1 that provides some helper functions relevant to the algorithm details. The first function checks if an event has been created in the Trace map, and if not, creates a new one with the specified gate type and adds it to the Trace. The second function creates an event with the specified name and type. The last function checks a constraint to identify the logical operator used in the constraint.

**Algorithm 4-3: Helper functions for synthesizing CFT model from a System behaviour model**

```
1:  function getIntermediateEvent (Trace, state, gateType)
2:      event = Trace.get(state)
3:      if isUndefined(event) then
4:          event = CREATEFTEVENT(gateType, gateType)
5:          Trace.put(state, event)
6:      end if
7:      return event
8:  end function
9:  function createFTEvent(name, gateType)
10:     event = createFTEvent(name)
11:     gate = createFTGate(gateType)
12:     event.gate = gate
13:     return event
14: end function
15: <constraint> ← <variable> <logical> <variable>
16: <logical> ←  and | or | xor
17: function parseConstraint(constraint)
18:     if constraint.contains(xor) then
19:         return xor
20:     else if constraint.contains(and) then
21:         return and
22:     else if constraint.contains(or) then
23:         return or
24:     end if
25: end function
```

## 4.3 Transformation Algorithm from CFT to SFT

In this step of the transformation, a System-level FT (SFT) is synthesized representing the failure behaviour of the system of interest. The transformation examines the structural interconnections and dependencies between the analyzed components in the previous section to identify how their failure affects the system as a whole. In other words, the CFTs generated in the previous step will be composed based on the interconnection of their respective instances represented in the system structure by blocks.

Here, we present an algorithm for the synthesis of an SFT model developed to realize the failure patterns introduced in subsection 4.1.2. This algorithm considers the system's structure view rather than behaviour view. It accepts the system's source model, the artifacts obtained from the previous behavioural view transformation step the CFT model and the Trace model that maps the elements of the CFT model with their source elements in the system's source model.

The algorithm starts with the initialization of the SFT model as a container for generated FT events. The algorithm composes the previously generated CFT models based on Inter Component Failure Pattern (13) that is presented in Figure 4-14. It loops over the events of the CFT models, and from the first cross-model trace it gets its SysML source elements. A source element is used to obtain the number of instances of the block containing the element and to check whether the triggering event came at a port from outside the block or it was an internal event. A new SFT event (SFTEvent) is generated for each CFT event of each SysML block instance. If it was an external event, considering the port (e.g. PortA) it was received at, the algorithm will find an error transition that has the effect of sending a signal to a port (e.g. PortB) connected to the port of interest (e.g. PortA). Entering this state

was mapped to an event in its containing block's CFT model (in the previous algorithm), we retrieve this using the cross-model trace. If the retrieved event has a gate, then the gate input events are transformed into SFT events recursively and added as input events to gate of the new SFT event (SFTEvent). In addition to the figure associated with Inter Component Failure Pattern (13), this composition can be seen visually in the case study presented in Chapter 8 in Figure 8-11.

Next the algorithm considers allocations of components with failure behaviour information known that is Simple Dependency Component Failure Pattern (9) presented in Figure 4-10. For each allocation that the client has failure information, we get the SFT model events of its CFT model top events and add the SFT event of the supplier failure as a causing event.

Here, the algorithm will consider blocks without State Machines to describe their behavioural failure, but with allocation-related failure information identified and annotated using *DaComponent* stereotype. We loop through all of these blocks and extract their instances. Iterating over the instances, we create a failure event for the owner of the instance if it does not exist with an OR-gate. If the block resource multiplicity is one and does not have substitutions or allocations, then a new event is created and added as the cause for the owner's failure event. This corresponds to Simple Dependency Component Failure Pattern (9) presented in Figure 4-10. Now, if the block has resource multiplicity, substitutions or allocations greater than one, an intermediate event is created and added as the cause to the owner failure event. If the block minimum number of required resource attribute is greater than one, we set the intermediate event's gate to an OR-gate, corresponding to Single Instance Component Failure Pattern (10) in Figure 4-11, otherwise, set it to an AND-gate that is Multi Instance Component Failure Pattern (11) in

Figure 4-12. A number of events are created matching the resource multiplicity, the substitutions and the allocations that are Substitution Component Failure Pattern (12) presented in Figure 4-13, which are added as causing events for the intermediate event.

**Algorithm 4-4: Step 2: Synthesizing SFT model from a System structure model**

```
1:  Input: System Source Model: System, Set of CFTModels from Step 1: SetOfCFTModels, Mapping  trace from
step 1: Trace
2:  Output: A Fault Tree model of the system failure: SFTModel
3:  Initialize SFTModel
4:  Initialize Trace2 = map(sourceElement, sftEvents)
5:  for each cftModel in SetOfCFTModels do                                  ▷ Pattern: 12
6:      for each event in cftModel.events do
7:          GENERATESFTEVENT(cftEvent)
8:      end for
9:  end for

10:  function GENERATESFTEVENT(cftEvent)
11:      result = Trace2.get(cftEvent)
12:      if isDefined(result) then
13:          return result
14:      end if
15:      sourceElement = Trace.getSourceElement(cftEvent)
16:      if sourceElement.hasEffect = true and sourceElement.effect.sendOnPort and
sourceElement.effect.port.connector.type.assembly = true then
17:          skip
18:      end if
19:      instances = sourceElement.getContainingBlock().instances
20:      eventsSet
21:      for each instance in instances do
22:          sftEvent = createFTEvent(cftEvent.name + instance.name)
23:          eventsSet.add(sftEvent)
24:          if sourceElement.isTrigger() and sourceElement.isOnPort() then
25:              triggerPort = sourceElement.getPort()
26:              sourcePorts = System.getSourcePortsConnectedToDestinationPort(triggerPort)
27:              for each sourcePort in sourcePorts do
28:                  stateMachine = sourcePort.containingBlock.stateMachine
29:                  transitions = stateMachine.getTransitionsWithEffectsOfSendSignalToPort(sourcePort)
30:                  for each trigger in transitions.triggers do
31:                      sourceCftEvent = Trace.getCFTElement(trigger)
32:                      if isDefined(sourceCftEvent.gate) then
33:                          sftEvent.gate = createFTGate(sourceCftEvent.gate.type)
34:                          for each event in sourceCftEvent.gate.events do
35:                              child = GENERATESFTEVENT(event)
36:                              sftEvent.gate.events.addAll(child)
37:                          end for
38:                      end if
39:                  end for
40:              end for
41:          end if
42:      end for
43:      SFTModel.events.addAll(eventsSet)
44:      Trace2.put(cftEvent, eventsSet)
45:      return eventsSet
46:  end function
```

**Algorithm 4-5: Step 2: Synthesizing SFT model from a System structure model (cont.)**

```
47: allocations = System.getAllocations()
48: for each allocation in allocations do                    ▷ Pattern: 8B
49:     client = allocation.client
50:     cftModel = SetOfCFTModels.get(client)
51:     if cftModel.isDefined then
52:         event = createFTEvent(allocation.supplier)
53:         Trace2.add(allocation.supplier, event);
54:         topEvents = getCFTModelTopEvents(cftModel);
55:         for each topEvent in topEvents do
56:             sftEvent = Trace2.get(topEvent)
57:             sftEvent.gate.events.add(event)
58:             SFTModel.events.add(event)
59:         end for
60:     end if
61: end for
```

**Algorithm 4-6: Step 2: Synthesizing SFT model from a System structure model (cont.)**

```
62: blocks = System.getAllBlocksHavingFailureInfoWithoutStateMachine()
63: for each block in blocks do
64:     instances = block.instances;
65:     for each instance in instances do
66:         owner = Trace2.get(instance.owner)
67:         if isUndefined(owner) then
68:             owner = CREATEFTEVENT(instance.owner.name, 'OR')
69:             Trace2.put(instance.owner)
70:         end if
71:         currentParent = owner
72:         Initialize eventsSet
73:         substitutes = instance.getSubstitutionClients()
74:         allocationSuppliers = instance.getAllocationSuppliers()
75:         if block.resourceMultiplicity > 1 or substitutes > 1 or allocationSuppliers > 1 then
76:             intermediate = createFTEvent(instance.name)
77:             currentParent.gate.events.add(intermediate)
78:             eventsSet.add(intermediate)
79:             if part.minNumberOfResources > 1 then                    ▷ Pattern: 9
80:                 currentParent.gate = createFTGate('OR')
81:             else                                                     ▷ Pattern: 10
82:                 currentParent.gate = createFTGate('AND')
83:             end if
84:             for i = 0; i < instance.resourceMultiplicity; i + + do
85:                 event = createFTEvent(instance.name + i)
86:                 intermediate.gate.events.add(event)
87:                 eventsSet.add(event)
88:             end for
89:             for each substitute in substitutes do
90:                 for i = 0; i < substitute.resourceMultiplicity; i + + do   ▷ Pattern: 11
91:                     event = createFTEvent(substitute.name+i)
92:                     intermediate.gate.events.add(event)
93:                     eventsSet.add(event)
94:                 end for
95:             end for
96:             for each supplier in allocationSuppliers do
97:                 for i = 0; i < supplier.resourceMultiplicity; i + + do  ▷ Pattern: 11
98:                     event = createFTEvent(supplier.name+i)
99:                     intermediate.gate.events.add(event)
100:                     eventsSet.add(event)
101:                 end for
102:             end for
103:         else                                                         ▷ Pattern: 8A
104:             event = createFTEvent(instance.name)
105:             currentParent.gate.events.add(event)
106:             eventsSet.add(event)
107:         end if
108:         SFTModel.events.addAll(eventsSet)
109:         Trace2.add(instance, eventsSet)
110:     end for
111: end for
```

# 5 Chapter: Transformation Implementation

In this chapter, we show the implementation of the proposed approach of synthesizing Component-level Fault Tree (CFT) and System-level Fault Tree (SFT) analysis models out of SysML system model using the Epsilon Transformation Language (ETL), which is part of the Epsilon family (Kolovos et al., 2008). ETL is a hybrid transformation language with declarative characteristics through the rules and imperative features inherited from the Epsilon Object Language (EOL) (Kolovos et al., 2006). An ETL transformation is contained in an ETL module, which can contain a number of *pre* and *post* blocks that execute before and after a transformation. Also, an ETL module contains a number of transformation *rules* and *operations*. An ETL transformation may handle one or more models as input and may generate one or more models as output.

An ETL rule declares that a given element source type is transformed into one or more target type elements (1-to-N mapping), with the ability to force a constraint on the source element to which the rule is applied. While an ETL transformation rule can consider only a single source element, it can generate multiple target elements. We define two ETL transformations to implement each of the algorithms defined in Sections 4.2 and 4.3 for the generation of CFT and SFT models. The algorithms were designed based on the mapping patterns defined in Section 4.1. As previously mentioned, a pattern is mapping a group of inter-related source model elements with a group of inter-related target model elements (M-to-N mapping), so more than one rule may be necessary to implement a mapping pattern. Utilizing ETL built-in capabilities and operations permitted to have a complete implementation of the mapping patterns using ETL's single source element rules.

The declarative nature of ETL posed a challenge in implementing the CFT and SFT imperative algorithms defined in Chapter 4. The approach followed in implementing the algorithms is based on two concepts. First, due to its imperative nature, a main outer loop is used to iterate over the elements of the source model. Second, the use of nested selection statements allows for generating matching target element selectively. The first concept - the outer loop - is performed by the ETL engine that iterates over the elements that match the type of given source element. The outer loop contains inner loops that are dedicated to different transformation rules. The second concept is realized through the use of ETL guards, that are specified per rule and can be a simple expression or a block of statements with a Boolean return value. Once the ETL engine obtains the elements matching the type of the source element, it also has to satisfy the defined guard.

In the following, we define the rules making reference to the algorithms introduced in the previous chapter, and the complete implementation can be found at (Al shboul, 2019).

## 5.1 Implementation of the transformation from SysML to CFT

This subsection presents the implementation of the first transformation of the SysML model into a CFT model. We list all the rules of the transformation.

### 5.1.1 Rules

This subsection provides the rules that compose the first ETL transformation, that is synthesizing SysML model behavioural view to the CFT model.

#### 5.1.1.1 Rule model2cfta

The Code Fragment 5-1 shows the rule that transforms a SysML Model element, which is the root of a SysML model into a CFT System element, that is the container for all CFT model elements. It assigns the Model element name attribute to the System element name attribute. This is not listed in the algorithm as it is a trivial root to root element transformation.

```
rule model2cfta
  transform sysml : SysML!Model
  to sys : CFTA!System  {
    sys.name = sysml.name;
}
```

**Code Fragment 5-1: Rule model2cfta**

#### 5.1.1.2 Rule block2component

In the Algorithm 4-1 line 3, obtaining the list of blocks that have failure behaviour is implemented by specifying the source type parameter as *SysML!Block* and setting the guard to check whether the block has a failure behaviour. This rule takes as input a Block that either has ports or has failure information and transforms it into a CFT Component element. It executes the transformation of the state machine with failure information into an FTAModel element and sets in the component (see line 7 and 8 in Algorithm 4-1). Then if it has a port, it is also transformed into an equivalent port of the CFT. Lastly, it obtains the containing Model element that has a System element mapped to it, which contains the component. This method allows modularization at the component level.

```
rule block2component
  transform blk : SysML!Block
  to cmp : CFTA!Component {
  guard: blk.hasPorts() or blk.hasFailureBehaviour()
  var base = blk.base_class;
  cmp.name = base.name;
  if(base.classifierBehavior.isDefined()){
    cmp.FTAModel = base.classifierBehavior.equivalent();
  }
  if ( blk.hasPorts() ) {
    for(part in blk.getPorts()){
      cmp.ports.add(part.equivalent());
    }
  }
  var modelElement = getModelElement(base);
  modelElement.equivalent().components.add(cmp);
}
```

**Code Fragment 5-2: Rule block2component**

#### 5.1.1.3 Rule statemachine2fta

This rule transforms a SysML state machine with failure information into an FTA model element. The FTA model acts as the container for all events and could represent more than one FT (see Algorithm 4-1, lines 7 and 8).

```
rule statemachine2fta
  transform sm : SysML!StateMachine
  to ft : CFTA!FTAModel {
  guard: sm.hasFailureBehaviour()
  ft.name = sm.name;
}
```

**Code Fragment 5-3: Rule statemachine2fta**

#### 5.1.1.4 Rule enterningfailurestate2event

This rule corresponds to the lines 10-21 in Algorithm 4-1. It applies to a *SysML!State* that is stereotyped as a failure state using the *DaStep* stereotype (not within a composite state and without intermediate failure for a retry behaviour). The rule generates an FT event representing that the failure state has been reached. Its containing state machine is retrieved to obtain the corresponding FTAModel that will contain the transformed event. It is named

based on its containing state machine. By default, it will have an OR-gate as being the general case. If the event probability is set in the source model, it will be passed into the target model. On condition that the state has a subsequent failure state, then the event reaching the subsequent failure state is generated and set as the top event for the previous event.

```
rule enteringfailurestate2event transform state : SysML!State to event :
CFTA!Event {
  guard { return state.hasStereotype("DaStep")
    and not state.isInternalIntermediateErrorJoin()
    and not state.isRetryState();
  }
  var ft = state.containingStateMachine().equivalent();
  event.name = nameElementWithParent(state.name, ft.name);
  event.gate= new CFTA!Gate;
  event.gate.type = CFTA!GateType#OR;
  event.probability = state.getPropFromDaStepAsDouble();
  if (state.isIntermediateFailure()) {
    var outgoings = state.getOutgoings().select(transition
      | transition.getTarget().isTypeOf(SysML!State)
      and transition.getTarget().hasStereotype("DaStep"));
    for (finalState in outgoings.target) {
      var parnetEvent = finalState.equivalent();
      if (parnetEvent.gate.isUndefined()) {
        parnetEvent.gate = new CFTA!Gate;
        parnetEvent.gate.type = CFTA!GateType#AND;
      }
      parnetEvent.gate.events.add(event);
    }
  }
  ft.events.add(event);
}
```

**Code Fragment 5-4: Rule enteringfailurestate2event**

#### 5.1.1.5 Rule transitionlogic2interevent

According to Algorithm 4-1 lines 37-43, this rule was implemented for the source type transition, which represents the outgoing transition of a choice pseudo-state with constraints. This rule applies to a *DaStep* stereotyped guarded transition oriented from a *Choice* pseudo-state to a failure state. This represents behavioural branching based on a certain constraint. Such a constraint can be of two types: a) a simple true/false constraint,

b) logic expression with more than one operand. The former type generates a single event without a causing event, while the latter adds a gate with a type matching the expression logical operator. If the source element has a defined probability, it is passed to the target element. This event is set as a causing event for the targeted state failure event and is also added to the corresponding FTA model.

```
rule transitionlogic2interevent
  transform transition :SysML!Transition
  to event :CFTA!Event {
  guard {
    return transition.hasStereotype("DaStep")
    and transition.trigger.isEmpty()
    and transition.getGuard().isDefined()
    and transition.source.isTypeOf(SysML!Pseudostate)
    and transition.source.getKind() = SysML!PseudostateKind#choice
    and transition.getTarget().isTypeOf(SysML!State)
    and transition.getTarget().hasStereotype("DaStep");
  }
  var ft = transition.containingStateMachine().equivalent();
  var topEvent = transition.getTarget().equivalent();
  if ( isSimpleGuard (transition.getGuard())) {
    event.name = nameElementWithParent(transition.getGuard()
      .getSpecification().getBodies().first(), ft.name);
  } else {
    event.gate= new CFTA!Gate;
    handleConstraint(event.gate, transition.getGuard());
    event.name = nameElementWithParent(event.gate.type.name, ft.name);
  }
  event.probability = transition.getPropFromDaStepAsDouble();
  topEvent.gate.events.add(event);
  ft.events.add(event);
}
```

**Code Fragment 5-5: Rule transitionlogic2interevent**

#### 5.1.1.6 Rule regiontrigger2basicevent

The lines 28-33 from Algorithm 4-1 were implemented using this rule. It applies to triggers initiating *DaStep* stereotyped transitions targeting a final state within a composite state. The behaviour when multiple executions are occurring, and the failure leads to the termination of the execution into a final state. In this type of state composition for failure

behaviour, a completion transition should exist targeting a choice node with an outgoing transition into a failure state. This outgoing transition from the choice node is transformed by the 5.1.1.5 rule into an intermediate event that is caused by the event being created. The event's name is assigned depending on whether its source event is a timeout or received on a port.

```
rule regiontrigger2basicevent
  transform trigger : SysML!Trigger
  to event : CFTA!Event {
  guard {
    var transition = trigger.eContainer();
    return transition.hasStereotype("DaStep")
      and transition.getTarget().isTypeOf(SysML!FinalState)
      and transition.source.getContainer().getState().isDefined()
      and transition.source.getContainer().getState().isComposite();
  }
  var transition = trigger.eContainer();
  var ft = transition.containingStateMachine().equivalent();
  var triggerEvent = trigger.getEvent();
  if (triggerEvent.isTypeOf(SysML!TimeEvent)) {
    event.name = nameElementWithParent(triggerEvent.name, ft.name+"::Timeout");
  } else {
    event.name = nameElementWithParent(triggerEvent.name, ft.name);
  }
  if ( not trigger.getPorts().isEmpty()) {
    event.name += " on " + trigger.getPorts().first().name;
  }
  event.probability = trigger.getPropFromDaStepAsDouble();
  var outgoingTransitions = transition.source.getContainer().getState()
    .getOutgoings().select(x |
      x.trigger.isEmpty() and not x.getGuard().isDefined());
  var decision = outgoingTransitions.first().target;
  for (outrns in decision.getOutgoings().select(x|x.hasStereotype("DaStep"))) {
    var topEvent = outrns.equivalent();
    topEvent.gate.events.add(event);
  }
  ft.events.add(event);
}
```

**Code Fragment 5-6: Rule regiontrigger2basicevent**

#### 5.1.1.7 Rule join2and

The case in lines 26 and 27 in Algorithm 4-1 was implemented by this rule. It applies to Join pseudo-state that has a *DaStep* stereotyped transition into a failure state. It is translated into an intermediate event with an AND-gate representing the synchronization represented by Join. It is added as a causing event of the event translated from reaching the failure state.

```
rule join2and
  transform join : SysML!Pseudostate
  to andtrgt : CFTA!Event {
  guard {
    return join.getKind() = SysML!PseudostateKind#join
    and join.getOutgoings().size() == 1
    and join.getOutgoings().first().hasStereotype("DaStep")
    and join.getOutgoings().first().getTarget().isTypeOf(SysML!State)
    and join.getOutgoings().first().getTarget().hasStereotype("DaStep");
  }
  var ft = join.containingStateMachine().equivalent();
  var topEvent = join.getOutgoings().first().getTarget().equivalent();
  andtrgt.name = nameElementWithParent("AND", ft.name);
  andtrgt.gate= new CFTA!Gate;
  andtrgt.gate.type = CFTA!GateType#AND;
  andtrgt.probability
    = join.getOutgoings().first().getPropFromDaStepAsDouble();
  topEvent.gate.events.add(andtrgt);
  ft.events.add(andtrgt);
}
```

**Code Fragment 5-7: Rule join2and**

#### 5.1.1.8 Rule triggercomposit2basicevent

This rule handles the special case of the triggers of the transitions targeting a join node presented in Algorithm 4-1 in lines 26 and 27. We needed to split this case into two rules due to complexity, namely this one and the one above. This rule applies to SysML Triggers that initiate transitions stereotyped with *DaStep* from within a composite state into a state that has a completion transition going into a Join node. This is the case of multiple executions that all have to fail for the containing component to fail. This case is mapped as

a basic event that causes an intermediate event transformed by another rule from the Join node. The naming of the generated event considers if the source event is a timeout event or received on a port.

```
rule triggercomposit2basicevent
  transform trigger : SysML!Trigger
  to event : CFTA!Event {
  guard {
    var transition = trigger.eContainer();
    return transition.hasStereotype("DaStep")
      and transition.getTarget().isInternalIntermediateErrorJoin()
      and transition.getSource().getContainer().getState().isDefined()
      and transition.getSource().getContainer().getState().isComposite();
  }
  var transition = trigger.eContainer();
  var triggerEvent = trigger.getEvent();
  var ft = transition.containingStateMachine().equivalent();
  var outgoingFailureTransitions = transition.getTarget().getOutgoings();
  var topEvent = outgoingFailureTransitions.first().getTarget().equivalent();
  if (triggerEvent.isTypeOf(SysML!TimeEvent)) {
    event.name =nameElementWithParent(triggerEvent.name, ft.name+"::Timeout");
  } else {
    event.name =nameElementWithParent(triggerEvent.name, ft.name);
  }
  if (not trigger.getPorts().isEmpty()) {
    event.name += " on " + trigger.getPorts().first().name;
  }
  event.probability = trigger.getPropFromDaStepAsDouble();
  topEvent.gate.events.add(event);
  ft.events.add(event);
}
```

**Code Fragment 5-8: Rule triggercomposit2basicevent**

#### 5.1.1.9 Rule errorTrans2event

This rule implements the case in Algorithm 4-1, lines 44-45. The rule applies to triggers of transitions stereotyped with *DaStep* transitioning the SM from a failure state into a subsequent failure state. This kind of behaviour generates an event as a cause to the event translated from the subsequent failure state, which has an AND-gate. An AND-gate shows that two events need to occur in order to reach the subsequent failure state: the event generated here and the original event that leads to initially reaching this failure state.

```
rule errorTrans2event transform trigger:SysML!Trigger to event:CFTA!Event {
  guard {
    var transition = trigger.eContainer();
    return transition.hasStereotype("DaStep")
    and transition.source.isTypeOf(SysML!State)
    and transition.source.hasStereotype("DaStep")
    and transition.target.isTypeOf(SysML!State)
    and transition.target.hasStereotype("DaStep");
  }
  var transition = trigger.eContainer();
  var ft = transition.target.containingStateMachine().equivalent();
  var topEvent = transition.target.equivalent();
  event.name = nameElementWithParent(trigger.event.name, ft.name);
  event.probability = trigger.getPropFromDaStepAsDouble();
  topEvent.gate.type = CFTA!GateType#AND;
  topEvent.gate.events.add(event);
  ft.events.add(event);
}
```

**Code Fragment 5-9: Rule errorTrans2event**

#### 5.1.1.10 Rule transitionRetry2event

This rule implements Algorithm 4-1, lines 34-36 and 52-56 (Retry Failure Pattern (5). It applies to a trigger of a *DaStep* stereotyped transition with a counting effect, which models the behaviour of retrying an event for a certain number of times before terminating in a failure state. This trigger's transition is assumed to have the effect of updating the counter value. The transition that loops back to the state where the failure event occurred is identified based on the threshold guard. If a consecutive series of error events instances (equal in number with the threshold value) takes place, the SM moves to the failure state. An equivalent number of basic events are generated and joined under an AND-gate as the cause for reaching the failure state event. Note that no target model elements are generated if the number of error events is less than the threshold, because failure does not occur in this case.

```
rule transitionRetry2event transform trigger : SysML!Trigger
  to events : Sequence(EMFTA!Event) {
  guard {
    var transition = trigger.eContainer();
    return transition.hasStereotype("DaStep")
    and hasCounterEffect(transition.getEffect())
    and transition.target.isTypeOf(SysML!State)
    and transition.target.hasStereotype("DaStep");
  }
  var transition = trigger.eContainer();
  var ft = transition.containingStateMachine().equivalent();
  var topEvent = transition.getTransitionRetryFailureState().equivalent();
  var transition = trigger.eContainer();
  var target = transition.target;
  var counter:Integer;
  for (inTransition in target.outgoings) {
    if (inTransition.target <> target) {
      var inCounter = getGuardMatchingEffect(inTransition,
        transition.effect.getBodies().first());
      if (inCounter.isDefined() and inCounter.isInteger()) {
        counter = inCounter.asInteger();
        break;
      }
    }
  }
  if (counter > 0) {
    var parent = topEvent;
    if ( counter > 1 ) {
      parent = new CFTA!Event;
      parent.name = nameElementWithParent('AND', ft.name);
      parent.gate = new CFTA!Gate;
      parent.gate.type = CFTA!GateType#AND;
      topEvent.gate.events.add(parent);
      events.add(parent);
    }
    for (i in Sequence{1..counter}) {
      var event = new CFTA!Event;
      var name = trigger.event.name + '[' + i + ']';
      event.name = nameElementWithParent(name, ft.name);
      event.probability = trigger.getPropFromDaStepAsDouble();
      parent.gate.events.add(event);
      events.add(event);
    }
    ft.events.addAll(events);
  }
}
```

**Code Fragment 5-10: Rule transitionRetry2event**

#### 5.1.1.11 Rule trigger2failureevent

This rule implements the last part of the Algorithm 4-1, lines 47-51. It is a general rule for transforming the triggering of a failure transition going to a failure state into an FT event, if it has not been transformed by another rule. The condition “has not been transformed” is verified by using the built-in ETL attribute that maintains the transformations that have already been performed with the source and target elements. The generated event reflects whether the triggered event was a timeout event or received on a port. This event is set as the causing event for reaching the failure state if it has an OR-gate. Otherwise, it is set as the cause of an intermediate event with an OR-gate causing the event of reaching the failure state.

```
rule trigger2failureevent
  transform trigger : SysML!Trigger
  to event : CFTA!Event {
  guard {
    var transition = trigger.eContainer();
    return transition.hasStereotype("DaStep")
      and transition.getTarget().isTypeOf(SysML!State)
      and transition.getTarget().hasStereotype("DaStep")
      and not transTrace.transformations.exists(t | t.getSource() = trigger);
  }
  var transition = trigger.eContainer();
  var triggerEvent = trigger.getEvent();
  var ft = transition.containingStateMachine().equivalent();
  var topEvent = transition.getTarget().equivalent();
  if(triggerEvent.isTypeOf(SysML!TimeEvent)) {
    event.name = nameElementWithParent(triggerEvent.name, ft.name+"::Timeout");
  } else {
    event.name = nameElementWithParent(triggerEvent.name, ft.name);
  }
  if(not trigger.getPorts().isEmpty()) {
    event.name += " on " + trigger.getPorts().first().name;
  }
  event.probability = trigger.getPropFromDaStepAsDouble();
  if (topEvent.gate.type <> CFTA!GateType#OR) {
    var temp = topEvent.gate.events.selectOne(event | event.gate.isDefined()
      and event.gate.type = CFTA!GateType#OR);
    if (temp.isDefined()) {  topEvent = temp;  }
    else {
      var newIntermediateEvent = new CFTA!Event;
      newIntermediateEvent.name = "OR";
      newIntermediateEvent.gate = new CFTA!Gate;
      topEvent.gate.events.add(newIntermediateEvent);
      ft.events.add(newIntermediateEvent);
      topEvent = newIntermediateEvent;
    }
  }
  topEvent.gate.events.add(event);
  ft.events.add(event);
}
```

**Code Fragment 5-11: Rule trigger2failureevent**

#### 5.1.1.12 Rule port2port

This rule transforms a SysML port into a CFT port. The port should have a type which contains Flow properties. Flow properties have a direction attribute, which is used to identify the direction of the port. This rule is marked as lazy, which means it has to be

invoked by another part of the implementation. In this case, the invocation takes place in 5.1.1.2 rule.

```
@lazy
rule port2port
  transform src : SysML!Port
  to trgt : CFTA!Port {
  trgt.name=src.name;
  trgt.dir=CFTA!Direction#Out;
  var type = src.type;
  if ( type.isUndefined() ) { return; }
  var direction;
  for (attr in type.getAllAttributes()) {
    var dir = SysML!FlowProperty.all.selectOne(x |
        x.base_Property.qualifiedName.matches(attr.qualifiedName))
        .direction.asString();
    var mapdir;
    if (dir.matches("in")) {  mapdir = CFTA!Direction#In;  }
    else if (dir.matches("out")) {  mapdir = CFTA!Direction#Out;  }
    else {  mapdir = CFTA!Direction#InOut;  }
    if (not direction.isDefined()) {  direction=mapdir;  }
    else if (direction<>mapdir) {  direction=CFTA!Direction#InOut;  }
  }
  if (src.isConjugated()) {
    if (direction = CFTA!Direction#In) {
      direction=CFTA!Direction#Out;
    } else if (direction = CFTA!Direction#Out) {
      direction=CFTA!Direction#In;
    }
  }
  trgt.dir=direction;
}
```

**Code Fragment 5-12: Rule port2port**

## 5.2 Implementation of the transformation from CFT+SysML to SFT

This subsection provides the implementation of the second transformation, which synthesizes SFT models from the CFT models generated in the first transformation (see Section 5.1) and the original SysML annotated model. The trace model of the first transformation is also considered to relate when needed the CFT model elements with the SysML source elements.

### 5.2.1 Rules

The rules that compose the transformation are briefly described in this section.

#### 5.2.1.1 Rule basicEventOnPort2SFTIntermediateEvent

This rule implements a specific case in Algorithm 4-4 the *GenerateSFTEvent* at lines 10-46 satisfying the if-statements starting with the one on line 24. This rule is applied to CFT events that are triggered by an event received on a port, which signifies an event occurred in a connected component that led to sending a failure signal event. A CFT event is contained in CFT component that was generated from a SysML block, using the cross-model link trace, we obtain the block and its instances. An SFT event is generated per instance of the component.

```
rule basicEventOnPort2SFTIntermediateEvent
  transform cftEvent :CFTA!Event
  to duplicateEvents :Set(EMFTA!Event) {
  guard : cftEvent.isAtInternalPortTriggeringFailure ()
  var parts = getComponentFTBlockInstances(cftEvent.eContainer().eContainer());
  var srcEvents = cftEvent.getTraceSysMLElement().getTriggerSourceEvents();
  for (part in parts){
    var event = new EMFTA!Event;
    event.name = part.name + "::" + cftEvent.name;
    event.probability = cftEvent.probability;
    for (srcEvent in srcEvents ) {
      if (srcEvent.gate.isDefined()
        and srcEvent.gate.events.size() > 0) {
        event.gate = srcEvent.gate.getEquivalentGate();
        for (innerEvent in srcEvent.gate.events) {
          event.gate.events.addAll(innerEvent.equivalent());
        }
      }
    }
    duplicateEvents.add(event);
    var ownerFTAModel = part.getOwner().getBlockByClass().equivalent();
    ownerFTAModel.events.add(event);
  }
}
```

**Code Fragment 5-13: Rule basicEventOnPort2SFTIntermediateEvent**

#### 5.2.1.2 Rule cftEventNotAtPort2SFTEvent

This rule implements the general case in Algorithm 4-4 presented in function *GenerateSFTEvent* in lines 10-46 with the conditions at line 24 are not satisfied. This rule applies to the CFT events that are either internal events or events that result in sending failure events to the system's external ports. Instances of the block from which the containing CFT was generated are considered for SFT events generation. If a CFT event has causing events, they are iterated over, and their equivalent SFT events are added as causing events for the resultant SFT event.

```
rule cftEventNotAtPort2SFTEvent
  transform sourceEvent :CFTA!Event
  to duplicateEvents :Set(EMFTA!Event) {
  guard : not sourceEvent.isAtInternalPortTriggeringFailure ()
      and not sourceEvent.doesEventSendAtInternalPortTriggeringFailure ()
  var parts =getComponentFTBlockInstances(sourceEvent.eContainer().eContainer());
  for (part in parts){
    var owner = part.getOwner().equivalent();
    var event = new EMFTA!Event;
    event.name = part.name + "::" + sourceEvent.name;
    event.probability = sourceEvent.probability;
    duplicateEvents.add(event);
    if (sourceEvent.gate.isDefined()
      and sourceEvent.gate.events.size() > 0) {
      event.gate = sourceEvent.gate.getEquivalentGate();
      for (innerEvent in sourceEvent.gaate.events ) {
        event.gate.events.addAll(innerEvent.equivalent());
      }
    }
    var ownerFTAModel = part.getOwner().getBlockByClass().equivalent();
    ownerFTAModel.events.add(event);
  }
}
```

**Code Fragment 5-14: Rule cftEventNotAtPort2SFTEvent**

#### 5.2.1.3 Rule allocationSupplier2sftevent

The case in Algorithm 4-4 in lines 47-61 is implemented here. Also, it handles another case in lines 62-111 specifically lines 74 and 96-102. This rule applies to allocation relations,

where the supplier block does not have a failure behaviour described by a SM, but rather is stereotyped with failure information using *DaComponent* and has an item flow connection with the client. The connected ports between the supplier and the client are identified. At the client side, triggers that expect events on any of the previously identified ports are identified, and their corresponding CFT events are obtained. The SFT equivalent event is obtained, and an SFT event is generated for the failure of the supplier that is added as a causing event for the client's SFT event.

```
rule allocationSupplier2sftevent
  transform block : SysML!Block
  to event : EMFTA!Event {
  guard {
    return block.isAnAllocationSupplier() and not block.hasFailureBehaviour()
      and block.hasStereotype("DaComponent");
  }
  var prob = block.getFailureOccurrenceProb();
  event.name = block.base_class.name + "Failure";
  event.probability = prob;
  var relevantAllocations : Sequence(SysML!Allocate)
    = SysML!Allocate.all.select(allocation
      | allocation.base_Abstraction.supplier.contains(block.base_class));
  var clients = relevantAllocations.base_Abstraction.client;
  for (client in clients.flatten().asSet()) {
    if (client.hasFailureBehaviour()) {
      var ftamodel = client.classifierBehavior.getCFTAElement().first()
        .getEquivalent().first();
      var topEvents = ftamodel.getFTAModelRootEvents();
      for (topevent in topEvents) {
        var sftTopEvent = topevent.equivalent();
        if(sftTopEvent.isDefined()) {
          for (sftTopEventI in sftTopEvent) {
            sftTopEventI.gate.events.add(event);
            sftTopEventI.eContainer().events.add(event);
          }
        } else {
          var siblings = topevent.gate.events.equivalent();
          for (sibling in siblings) {
            var sftTopEvents = sibling.getParnetEventsByEvent();
            for (sftTopEventI in sftTopEvents) {
              sftTopEventI.gate.events.add(event);
              sftTopEventI.eContainer().events.add(event);
            }          }        }      }
    } else if (client.getBlockByClass().hasPorts()) {
      var ports = client.getOwnedAttributes().select(attribute |
        attribute.getAggregation() == SysML!AggregationKind#composite
        and attribute.isTypeOf(SysML!Port)
        and attribute.hasStereotype("ProxyPort"));
      for (port in ports) {
        var itemFlows : Sequence(SysML!ItemFlow) = SysML!ItemFlow.all.select(
          itemFlow | itemFlow.base_InformationFlow
            .getInformationSources().contains(port));
        for (itemFlow in itemFlows) {
          var targetPort
            = itemFlow.base_InformationFlow.getInformationTargets().first();
          var triggers = SysML!Trigger.all.select(trigger |
            trigger.hasStereotype("DaStep")
            and not trigger.getPorts().isEmpty()
            and trigger.getPorts().contains(targetPort));
          for (trigger in triggers) {
            var cftElement = trigger.getCFTAElement().first().getEquivalent();
            var sftEvent = cftElement.first().equivalent().first();
            if (sftEvent.gate.isUndefined()) {  sftEvent.gate = new EMFTA!Gate;  }
            sftEvent.gate.events.add(event);
            sftEvent.eContainer().events.add(event);
          }        }      }    }  }}
```

**Code Fragment 5-15: Rule allocationSupplier2sftevent**

#### 5.2.1.4 Rule mainblock2sftmodel

This rule considers SysML block that has parts but does not have on its own an identified failure behaviour nor stereotyped with failure information using DaComponent.

```
@lazy
rule mainblock2sftmodel
  transform block : SysML!Block
  to ftamodel : EMFTA!FTAModel {
  guard {
    return block.hasParts()
      and not block.hasFailureBehaviour()
      and not block.hasStereotype("DaComponent");
  }
  ftamodel.name = block.base_class.name;
}
```

**Code Fragment 5-16: Rule mainblock2sftmodel**

#### 5.2.1.5 Rule blockWithFailureInfo2sftevent

This rule implements the case in Algorithm 4-4, lines 62-111 in general, and in special the cases in lines 79-88 and 103-107. This rule is applied to blocks that have at least one instance, are stereotyped with failure information using *DaComponent* without failure behaviour SM and do not have internal parts. The goal of this rule is to add a failure event representing this block using the annotated information for each owner of the block instances. Owner blocks of the instances of this block are looped over obtaining the SFT FTA model equivalent to it. For each FTA model, its top events are obtained. Based on the failure information, if the resource has one instance, an SFT event is generated and added as a causing event for these top events. Whereas if it has multiple instances, an intermediate event is generated to organize them. The value of the attribute “minimum number of resources” is considered to model the cases that if one failure occurrence of the block will lead to the failure of the owner or if more than one failure occurrence is needed for the

owner to fail. The first case is modeled by joining an equivalent number of events to the multiplicity under an OR-gate of the intermediate event, while in the latter case events will be joined under an AND-gate of the intermediate event.

```
rule blockWithFailureInfo2sftevent
  transform block : SysML!Block
  to events : Set(EMFTA!Event) {
  guard {
    return not block.hasParts()
      and not block.hasFailureBehaviour()
      and block.hasStereotype("DaComponent")
      and block.getBlockInstances().size() > 0;
  }
  var prob = block.getFailureOccurrenceProb();
  var ownerInstancesMap=block.getBlockInstances().getInstancesGroupedByOwner();
  for (owner in ownerInstancesMap.keySet()) {
    var ftaModel = owner.equivalent();
    for (rootEvent in ftaModel.getFTAModelRootEvents()) {
      for (instance in ownerInstancesMap.get(owner)) {
        if (block.getResourceMultiplicity() > 1) {
          var intermediate = new EMFTA!Event;
          intermediate.name = instance.name + " All Failure";
          intermediate.gate = new EMFTA!Gate;
          if (block.getMinNumberOfResources() > 1) {
            intermediate.gate.type = EMFTA!GateType#OR;
          } else {
            intermediate.gate.type = EMFTA!GateType#AND;
          }
          var i :Integer = 1;
          while ( i <= block.getResourceMultiplicity()) {
            createSFTEvent(instance.name + i, prob, intermediate, ftaModel);
            i++;
          }
          rootEvent.gate.events.add(intermediate);
          events.add(intermediate);
          ftaModel.events.add(intermediate);
        } else {
          var basicEvent = createSFTEvent(instance.name + " Failure", prob,
            rootEvent, ftaModel);
          events.add(basicEvent);
        }
      }
    }
  }
}
```

**Code Fragment 5-17: Rule blockWithFailureInfo2sftevent**

#### 5.2.1.6 Rule blockAsSpareWithFailureInfo2sftevent

This rule implements a specific condition in Algorithm 4-4, where the value from line 73 satisfies the condition at line 75 and is used in lines 89-95. This rule covers the case of spares and their substitutions. It applies to blocks that do not have their failure behaviour specified with SM, which are stereotyped using *DaSpare* and have at least a single instance without internal parts. The block substitutions are obtained, and a number of SFT events are generated matching the value of the resource multiplicity. The supplier substitute also does not have a SM failure behaviour model, only failure information via *DaComponent* stereotype, and their failure is transformed into SFT model events using the 5.2.1.5 rule. Looping through the substitutions, we obtain the equivalent events for the supplier block; if these events have gates identified, the substitutions events are added as causing events. Otherwise, the top events of these supplier events are obtained, and the substitution events are joined under a new intermediate event with an AND-gate which is added as a causing event to the top events.

```
rule blockAsSpareWithFailureInfo2sftevent
  transform spare : SysML!Block
  to events : Set(EMFTA!Event) {
  guard {
    return not spare.hasParts()
      and not spare.hasFailureBehaviour()
      and spare.base_Class.hasStereotype("DaSpare")
      and spare.getBlockInstances().size() > 0;
  }
  var substitutions = spare.base_class.getSubstitutions();
  if (substitutions.isUndefined() or substitutions.isEmpty()) {  return;  }
  var i :Integer = 1;
  while ( i <= spare.getSpareResourceMultiplicity()) {
    var basicEvent = new EMFTA!Event;
    basicEvent.name = spare.base_class.name + " " + i + " Failure";
    basicEvent.probability = spare.getSpareFailureOccurrenceProb();
    events.add(basicEvent);
    i++;
  }
  for (substitution in substitutions) {
    var suppliers = substitution.supplier;
    for (supplier in suppliers) {
      var supplierEvents = supplier.getBlockByClass().equivalent();
      for (supplierEvent in supplierEvents) {
        if (supplierEvent.gate.isDefined()) {
          supplierEvent.gate.events.addAll(events);
        } else {
          var intermediate = new EMFTA!Event;
            intermediate.name = supplierEvent.name + " "
            + spare.base_class.name + " All Failure";
          intermediate.gate = new EMFTA!Gate;
          intermediate.gate.type = EMFTA!GateType#AND;
          var parentEvents = supplierEvent.eContainer().events.select(event
            | event.gate.isDefined()
            and event.gate.events.includes(supplierEvent));
          if (parentEvents.isDefined() and not parentEvents.isEmpty()) {
            for (parentEvent in parentEvents) {
              parentEvent.gate.events.remove(supplierEvent);
              parentEvent.gate.events.add(intermediate);
            }
          }
          intermediate.gate.events.add(supplierEvent);
          intermediate.gate.events.addAll(events);
          supplierEvent.eContainer().events.add(intermediate);
        }
        supplierEvent.eContainer().events.addAll(events);
      }
    }
  }
}
```

**Code Fragment 5-18: Rule blockAsSpareWithFailureInfo2sftevent**

# 6 Chapter: Feedback of results

This chapter describes the activity of updating the source model with the analysis results. This activity is performed following the analysis of the System-level Fault Tree (SFT) and Component-level Fault Tree (CFT) models. This is to integrate the results of the Fault Tree Analysis (FTA) from the CFT and SFT analysis models back into the SysML system model. This allows for maintaining the values in subsequent activities and permits sharing among interested stockholders.

The FTA quantitative information in terms of events probabilities is updated in the FT models provided. It uses the probabilities of the basic events that are obtained from safety information annotated in the SysML system model. It updates the probabilities of the non-basic events, which are to be feedback into the SysML system model.

## 6.1 Transformation trace model

As mentioned in the transformation details, a trace model is generated following each transformation step providing cross-model tracelinks. In this subsection, we introduce the trace model used in this work, inspired from (Kolovos, 2009) and modified for our needs. The metamodel is presented in Figure 6-1. The root element is called *Trace* and it contains any number of *TraceLink* elements. TraceLink consist of the attributes: a *ruleName* that stores the name of the rule that was executed, an *index* of the transformation in order of execution, a *source* of type ECore EObject to hold the source element that was transformed and a *targets* list of elements that were generated from executing the rule.

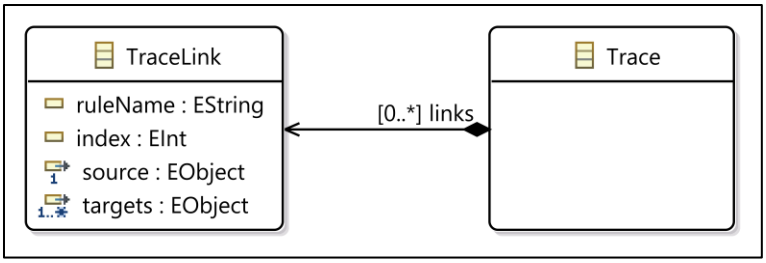



**Figure 6-1: Trace metamodel**

The goal here is to build a cross-model tracelinks between the source elements that results in the generation of target elements. The tracelinks provide traceability between source elements and target elements, that has a benefit of propagating changes between these models as used in (Mani et al., 2011). Also, tracelinks can be used to ensure a source element is transformed one time only. In our proposed approach, these tracelinks are beneficial when performing multiple transformations on related models.

As discussed earlier, an ETL module had an optional special block called *post* that gets executed after an ETL module transformation is finished. This block is utilized to obtain a built-in object prepared by ETL engine called *transTrace* which has an attribute called *transformations* that is a list of t executed transformations. Each transformation object contains a *source* attribute, a t*argets* collection attribute and a transformation rule *object*. In the post block, for each transformation object a *traceLink* element is created with the values of the transformation attributes with an index for the order of the transformation object.

A post block is executed after each of the transformations described in 5.1 and 5.2 for generating CFT and SFT models respectively. Thus, two trace models are generated for these two transformations.

## 6.2 Design

Utilizing the generated trace models from the transformations in 5.1 and 5.2, an algorithm was designed to allow to integrate the analysis results back into SysML system model. The objective is to reflect the events information from the analysis models to the corresponding elements of the source model that they were generated from.

The Algorithm 6-1 takes as input the SysML system model, the CFT models, the Trace model generated from the CFT model transformation, the SFT model and the second Trace model generated from the transformation of the SFT model. First, it iterates over all events in the CFT models and uses the first Trace model to obtain the SysML elements from which the events were generated, then reflects the event's value into the element. Second, it iterates over all the events in the SFT model using the second Trace to obtain the elements they were generated form. The SFT events are either generated from CFT events or SysML elements since the goal is to obtain the SysML elements. The latter case is satisfied while in the former case, the SysML element is obtained by visiting the first Trace model, then the element is updated.

**Algorithm 6-1: In place update for the SysML system source model**

```
1:  Input: System source model: System, CFT model for the system: CFT, SFT model
for the system: SFT, Trace model from CFT model generation: Trace, Trace model
from SFT model generation: TraceSecond
2:  Output: Update System source model: System
3:  cftModels = getListOfCFTModel()
4:  for each cftmodelinCFTModels do
5:      for each eventincftmodel do
6:          sysmlElement = Trace.findSysmlSourceElement(event)
7:          sysmlElement.DaStep.failure.probability = event.probability
8:      end for
9:  end for
10:  for each eventinSFT.allEvents do
11:      cftElement = TraceSecond.findCFTSourceElement(event)
12:      if isDefined then
13:          sysmlElement = Trace.findSysmlSourceElement(event)
14:      else
15:          sysmlElement = TraceSecond.findSysMLSourceElement(event)
16:      end if
17:      sysmlElement.DaStep.failure.probability = event.probability
18:  end for
```

## 6.3 Implementation

This section provides an implementation of the algorithm proposed in Algorithm 6-1. This activity is performed using the Epsilon Object Language (EOL) (Kolovos et al., 2006). EOL is the core language of the Epsilon model management framework. It is an imperative language based on a combination of JavaScript and OCL. Here we list the main two operations of the EOL module implementing the algorithm, and the complete code is found at (Al shboul, 2019).

The operation in Code Fragment 6-1 is invoked simply by passing a CFT model. It loops over all the model events that have a probability greater than 0, meaning its been initialized. For each event, we obtain the SysML element that is the source of it. If the probability of the source and the target element does not match, the value of the target element is applied

to the source element. Then a utility method is invoked to update the container of the SysML element.

```
operation crossModelCFTASysML(emftamodel) {
  for (ele in emftamodel.events) {
    if (ele.probability = 0) {
      continue;
    }
    var src = ele.getSysMLSourceElement();
    if (src.isUndefined() or src.isJoinNode()) {
      continue;
    }
    var prop = src.getStereotypeAttribute("DaStep","prob").asDouble();
    if (prop <> ele.probability) {
      var stereotype = src.getAppliedStereotypes()
        .selectOne(s: SysML!Stereotype | s.name = "DaStep");
      var result =  src.setValue(stereotype, "prob",
        ele.probability.asString());
      SysML.store();
    }
    checkTransition (src);
  }
}
```

**Code Fragment 6-1: Operation crossModelCFTASysML**

The operation Code Fragment 6-2 iterates over the events of the SFT model. For each event, first, it tries to obtain the CFT element that generated this event. If none is found, then it was generated from a SysML element directly. If it were generated from a CFT element, we would obtain the SysML element from which the CFT element was generated. Then we compare the probabilities of occurrence. If there is no match, the value from the SFT model element is applied to the SysML element. Then a utility method is invoked to update the container of the SysML element.

```
operation crossModelEMFTASysML() {
  for (ele in EMFTA!Event.all){
    var cfta = ele.getSourceElement();
    var src;
    if (cfta.isUndefined()) {
      src = ele.getSourceElement();
    } else {
      src = cfta.getSysMLSourceElement();
    }
    if (src.isUndefined()) {
      continue;
    }
    if (src.getStereotypeAttribute("DaStep","prob").isDefined()) {
      var prop = src.getStereotypeAttribute("DaStep","prob").asDouble();
      if (prop <> ele.probability){
        var stereotype = src.getAppliedStereotypes()
            .selectOne(s: SysML!Stereotype | s.name = "DaStep");
        var result =  src.setValue(stereotype, "prob",
            ele.probability.asString());
        stereotype = src.getAppliedStereotypes()
            .selectOne(s: SysML!Stereotype | s.name = "DaStep");
        SysML.store();
      }
      checkTransition (src);
    }
  }
}
```

**Code Fragment 6-2: Operation crossModelEMFTASysML**

This step performs an in-place update for the SysML system model, without generating any new models.

# 7 Chapter: FMEA model generation

In this chapter, we present the last objective of the thesis research, which aims at performing Failure Mode and Effects Analysis (FMEA), another safety analysis technique that usually accompanies Fault Tree Analysis (FTA). FMEA works in the opposite direction of FTA: it starts with potential failure modes at the component-level and considers their causes and effects at the local and system-level.

As already mentioned, the transformation extracts from the SysML source model only model elements related to failure, which are annotated with Dependability Analysis and Modeling (DAM) failure stereotypes. Source elements not describing failure are not transformed into FMEA model elements. The use of FMEA table is in the form of a table listing components, their failure modes, causes, local and system effects and the probability of occurrence. For this purpose, we use another model management language from Epsilon framework, called Epsilon Generation Language (EGL) (Rose et al., 2008), whose purpose is model-to-text transformation.

The proposed approach utilizes an annotated SysML source model that has gone through the previous activities of generating the Component-level Fault Tree (CFT) and System-level Fault Tree (SFT) models along with their cross-model tracelinks to generate the FMEA model.

## 7.1 FMEA target model

First, we will identify the target FMEA model/content. The proposed table header shown in Table 7-1 will consist of:

- *System*: The name of the system the FMEA is being generated for.

- *Component*: The name of the component being listed. A component could have multiple failure modes.
- *Failure mode:* The failure mode represents how a component can fail.
- *Failure cause:* The event that caused the failure mode.
- *Local effect:* The effect the failure has at the component level.
- *System effect:* The effect of the failure mode on the system containing the component.
- *Occurrence*: The probability of the failure occurrence.

**Table 7-1: FMEA table header**

| System: | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| **Component** | **Failure Mode** | **Failure Cause** | **Local Effects** | **System Effects** | **Occurrence (Prob)** |
| | | | | | |

## 7.2 Transformation

Based on the definition of the FMEA elements, Table 7-2 presents a mapping between selected elements from the models generated in the previous steps and the FMEA target elements. The input models include the system source model, the CFT models generated from the first transformation, the SFT model generated from the second transformation and the cross-model trace-links from these two transformations.

**Table 7-2: Mapping of source model into FMEA elements**

| **Source elements**<br><br>**Models: SysML, CFT, SFT, cross-model trace-links** | **FMEA target element** |
|---|---|
| SysML block with modeled failure behaviour | Component |

| | |
|---|---|
| SysML block annotated using DaComponent | Component |
| SysML trigger annotated with DaStep | Failure mode |
| SysML block's DaComponent (as a single failure mode) | Failure mode |
| SysML trigger on a port of timeout event | Cause |
| SysML failure state caused by a transition send event causing the trigger | Cause |
| SysML trigger by an internal event | Cause |
| CFT event that is caused by the equivalent basic event of the SysML element representing the failure mode. | Local effect |
| SFT event that has as a basic event the equivalent event of the SysML element representing the failure mode. | System effect |
| SysML trigger DaStep probability value | Occurrence (Prob) |
| SysML block DaComponent failure.OccurrenceProb | Occurrence (Prob) |

An algorithm was developed to guide the generation of the FMEA model. The system header field is populated by retrieving the value of the SysML model element name. The list of components of the system is retrieved. For each component, it either has a failure behaviour SM annotated with the DAM *DaStep* stereotype or failure information stereotyped at the block level using the DAM *DaComponent* stereotype.

For a component with failure behaviour, any trigger of a transition targeting a failure state is considered a failure mode. The naming of the failure mode in the case an event is a timeout event uses the format "Timeout <Event name> [on <Port name>]". To determine

the cause, we examine the origin of the trigger, if it was triggered on a port with a timeout event, then its set to *Input Omission*. If the trigger was on a port with a signal, then we track the transitions that send the same signal to a connected por. These transitions targets are set as the cause of the failure mode. If the trigger was neither of these two cases, then it is set to *Internal Failure*. To obtain the local effect, we obtain the component's corresponding CFT from which we get the trigger's generated events, then determine the top events that can be reached from it to be used as local effects. Similarly, the system effect is obtained but rather from the SFT model. The probability is obtained from the *DaStep* stereotype.

For a component without failure behaviour SM but with failure information using *DaComponent*, the failure mode is set as the event representing the failure of the component itself. An internal failure is considered as the cause because no behaviour was specified using a SM, only failure information using *DaComponent*. The local effect is ignored as the whole component is failing. For the system effect, SFT model is visited to obtain the events generated from a component failure info, then get the top events that are caused by the obtained components, which are considered the system effects.

**Algorithm 7-1: FMEA model Generation**

```
1:  Input: System source model: System, CFT model for the system: CFT, SFT model for the system: SFT, Trace
model from CFT model generation: Trace, Trace model from SFT model generation: TraceSecond
2:  Output: FMEA {{Component, {FailureMode, FailureCause, FailureLocalEffect, FailureSystemEffect,
FailureProbability}}}
3:  components = System.getAllComponents()
4:  for each component in components do
5:      Initialize FMEAComponent as {component, {FMEAFailureMode}}
6:      FMEAComponent.component = component.name
7:      if component.hasFailureBehaviorModel then
8:          stateMachine = block.behaviorModel
9:          failureTriggers = stateMachine.findAllTriggersWithStereotype(DaStep)
10:          for each failureTriggerinfailureTriggers do
11:              Initialize FMEAFailureMode as {failureMode, failureCause, failureLocalEffect, failureSystemEffect,
failureProbability}
12:              FMEAFailureMode.failureMode = failureTrigger.event.name
13:              if trigger.event.isTimeOut then
14:                  FMEAFailureMode.failureMode = ”Timeout ” + FMEAFailureMode.failureMode
15:                  if trigger.isOnPort then
16:                   FMEAFailureMode.failureMode = FMEAFailureMode.failureMode+”on”+trigger.getPort().name
17:                  end if
18:                  FMEAFailureMode.failureCause = ”Input omission”
19:              else if trigger.isOnPort and System.hasAssemblyConnections(trigger.onPort) then
20:                  sourceTransitions = System.findTransitionsWithEffectsSendSignal(trigger.onPort,
trigger.eventSignal)
21:                  FMEAFailureMode.failureCause = findAllTargetState(sourceTransitions)
22:              else
23:                  FMEAFailureMode.failureCause = ”Internal failure”
24:              end if
25:              cftModel = CFT.getByStatemachine(stateMachine)
26:              cftEvents = cftModel.findGeneratedEvents(trigger)
27:              FMEAFailureMode.failureLocalEffect = cftModel.findTopEvents(cftEvents)
28:              sftEvents = SFT.findGeneratedEvents(cftEvents)
29:              FMEAFailureMode.failureSystemEffect = SFT.findTopEvents(sftEvents)
30:              FMEAFailureMode.probability = trigger.DaStep.probability
31:          end for
32:      else if component.hasStereotype(DaComponent) then
33:          Initialize FMEAFailureMode as {failureMode, failureCause, failureLocalEffect, failureSystemEffect,
failureProbability}
34:          FMEAFailureMode.failureMode = component.name + ” Failure”
35:          FMEAFailureMode.failureCause = ”Internal failure”
36:          FMEAFailureMode.failureLocalEffect = ”Internal failure”
37:          sftEvents = SFT.getEvents(component)
38:          FMEAFailureMode.failureSystemEffect = SFT.findTopEvents(sftEvents)
39:          if component.DaComponent.multiplicity > 1 then
40:              FMEAFailureMode.probability = component.DaComponent.probability *
component.DaComponent.multiplicity
41:          else
42:              FMEAFailureMode.probability = component.DaComponent.probability
43:          end if
44:      else continue;
45:      end if
46:  end for
```

## 7.3 Implementation

The FMEA model generation was performed using the Epsilon Generation Language (EGL), which is a model-to-text transformation language. It is a template-based generator where the EGL code resembles the text that will be generated. An EGL code consists of two sections: a static section that appears as is on the generated artifact and a dynamic section that controls the content obtained from the input model. These two sections can overlap; this is seen as injecting the dynamic part within special identifiers for the EGL engine to realize and properly replace with content from the source model.

The target FMEA model is finally presented in a Microsoft Excel compatible document. For this end, Microsoft provides an XML format it accepts to be handled using Microsoft's widely used Excel software with tabular content (Microsoft Office Dev Center, 2018).

An EGL template has been developed in the thesis realizing the mapping and implementing the proposed algorithm for generating the FMEA model. The developed EGL template static section complies with the before mentioned target sheet structure and format. The dynamic section contains the implementation of the algorithm. The EGL template is presented in Appendix A with more details. The approach has been applied to the presented case studies in the following chapter.

# 8 Chapter: Case studies

This chapter presents three case studies taken from the literature and applies the proposed approach to each. We selected case studies from literature in order to avoid author bias and to make sure that our proposed method covers a variety of modeling features.

## 8.1 Tracking Processing Unit (TPU) case study

This case study was inspired by a tutorial (Höfig, 2016) presented at the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE, 2016). The system called Tracking Processing Unit (TPU) is a part of the Navigation and Primary Flight Display system within the Airbus A380 aircraft. TPU is used to compute the aerospace situation, which is composed of the current aircraft position (GPS data) and other aircraft information (sensor data).

### 8.1.1 Source Model

To perform safety analysis on the TPU with reference to the activities in the proposed approach provided in Figure 3-1, we start by performing architectural modeling and annotation. The TPU is composed of four block types with their instances:

- GPS Receiver: contains a proxy port for accepting delegated GPS signal, and providing redundant copy through two proxy ports. There is a single instance called R.
- RS224 Channel: accepts GPS signal input and provides an output through two proxy ports. There are two redundant instances, A and B.
- Channel Interface: accepts redundant input through 2 proxy ports and one output through a proxy port. There is a single instance called I.

- Position Processing Unit: accepts delegated Radar Sensor data and GPS signal data and provides aerospace situation data delegated to the TPU container component. There is a single instance called P.

There are also three proxy ports: GPSIn (accepting GPS signal input); SensorIn (accepting Radar sensor data input) and TPUOut (providing the aerospace situation). Each block defined above has its properties. To model this in SysML, we use a BDD, as shown in Figure 8-1.

The following step is to model the Internal Structure, that is, describing the interconnection among the blocks instances that compose the TPU, which are: a) an instance of the GPS Receiver, R; two instances of the RS224 Channel, A & B; an instance of the Channel Interface, I; and an instance of the Processing Unit, P.

Now, we shall illustrate the interconnection between the components of TPU and the information exchanged among them as flows:

- A flow item delegates the GPS signal from the TPU's GPSIn proxy port to the R's Rin proxy port.
- R's ROut proxy port is connected to the redundant A & B CIn proxy ports via realizing item flows.
- A & B COut proxy ports are connected to I's IInA and IInB proxy port with a realizing item flow connecting A's COut to IInA and B's COut to IInB.

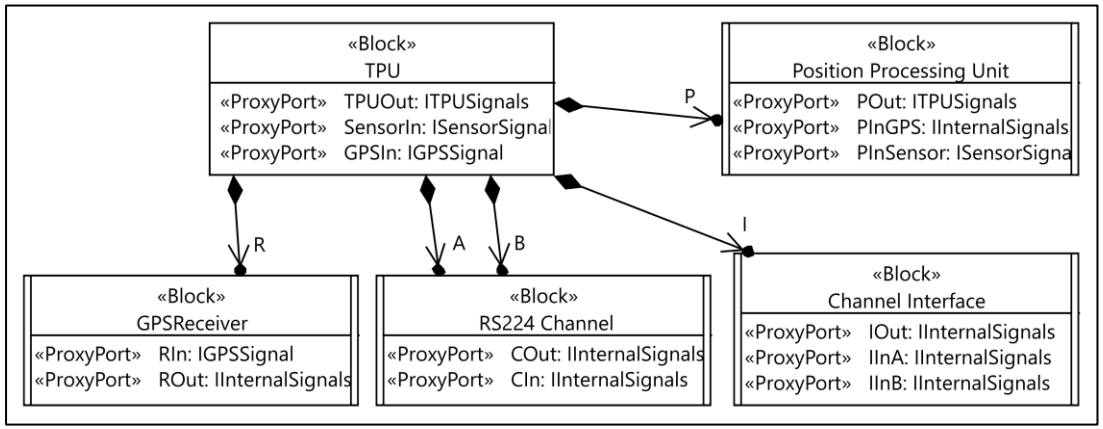



**Figure 8-1: TPU BDD**

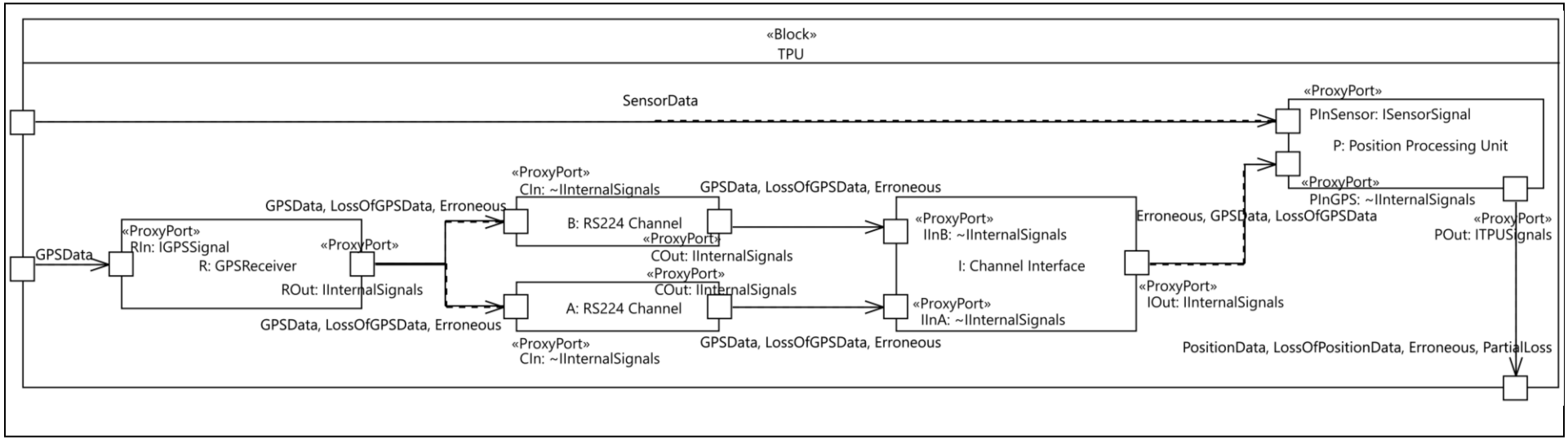



**Figure 8-2: TPU IBD**

- I's IOut proxy port is connected to P's PInGPS proxy port conveying GPSData from I. P receives another input delegated from the TPU to P's PInSensor, conveying the radar sensor.
- Except for the two input connectors, radar sensor data and GPS data, each connector is realized with the failure signals that each component can produce.

The SysML model obtained from this activity is an IBD shown in Figure 8-2.

Next, the behaviour of each block/component is modeled with SysML State Machine Diagrams (SMDs). For the case study on hand, four SMDs specify the behaviour of each block/component type. Figure 8-3 depicts the behaviour of the GPS Receiver component, Figure 8-4 depicts the RS 224 channel behaviour. Figure 8-5 shows the behaviour of the channel interface, which is an example of modeling a part that accepts a replicated input on two ports. In this part's behaviour, one non-failure input is sufficient to proceed with the nominal behaviour. For this component to fail on the input, both ports must receive a failure signal. This behaviour is modeled using a fork for the parallel behaviour of accepting the two inputs, and a join on the receiving of a failure signal forcing the wait of another failure signal or a successful signal.

The behaviour of the Processing Unit is depicted in Figure 8-6. This part shows an interesting business logic: it accepts two different inputs, and both are required in order to proceed with the nominal behaviour. This component may fail in the input accepting state in three situations. In the first situation two failures occur, one input times out and the other is a failure signal. The second situation is when one of the two before-mentioned failures occur.

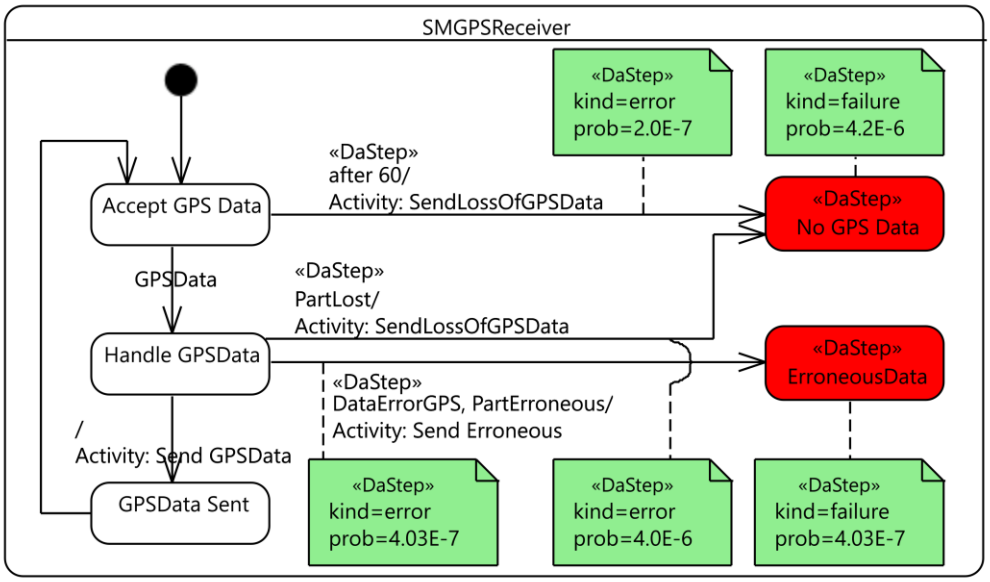



**Figure 8-3: GPS Receiver State Machine Diagram**

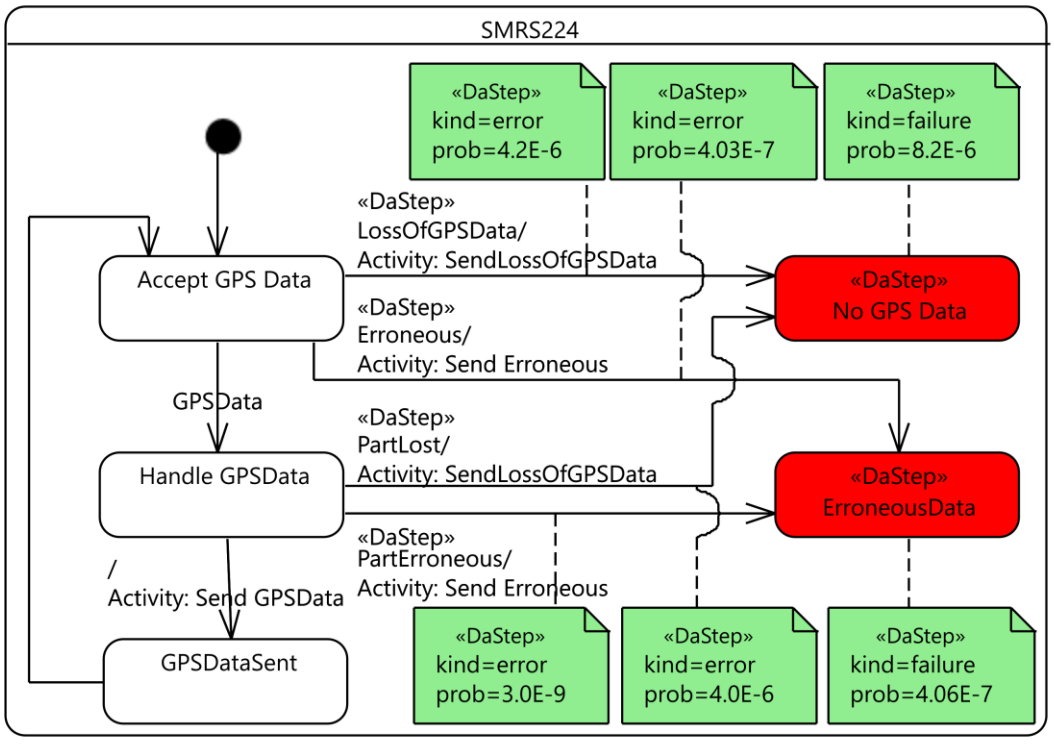



**Figure 8-4: Channel State Machine**

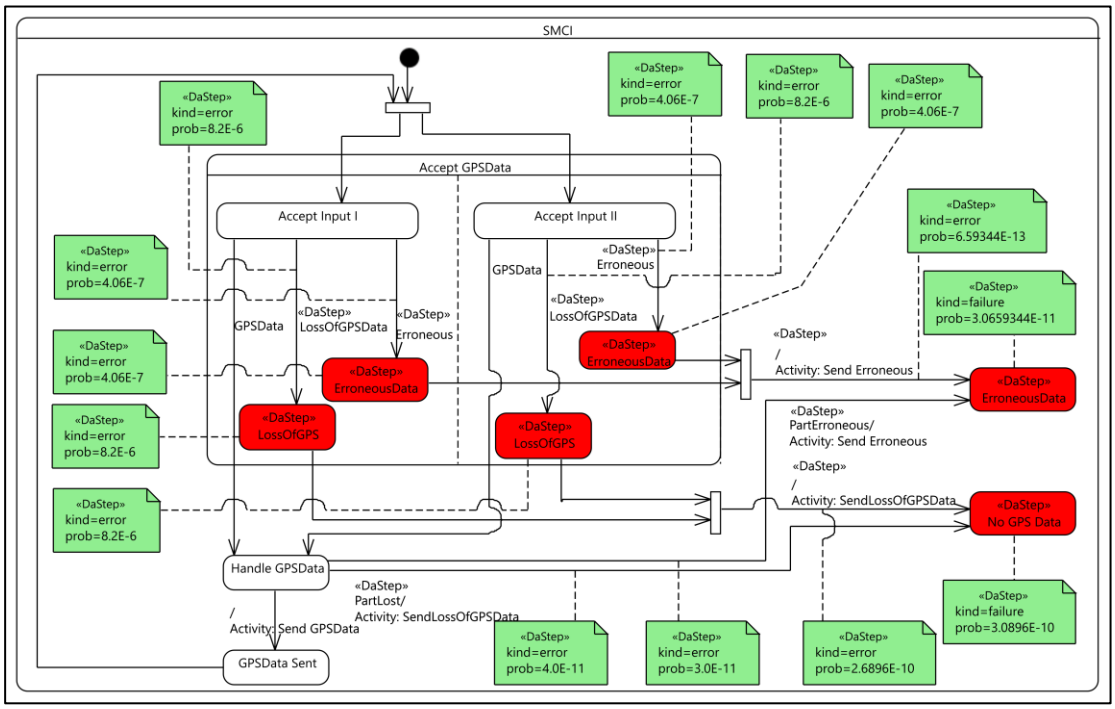



**Figure 8-5: Channel Interface State Machine**

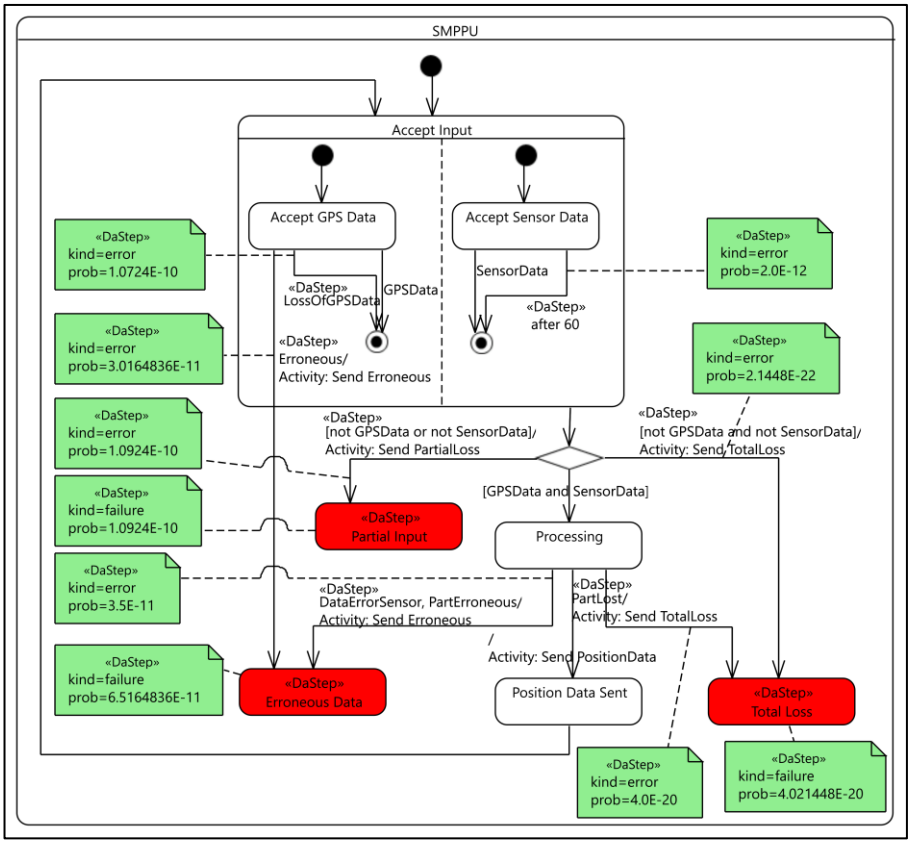



**Figure 8-6: Processing Unit State Machine**

Lastly, when one input receives a specific signal that promotes the input as invalid, disregarding the other input. We propose using a composite state with two regions, each corresponding to one distinct input and the possible behaviour associated with that input. Once both regions have reached their own final state, a completion transition is carried out into the choice node. The choice node outgoing transitions have guard constraints to select possible paths. In order to proceed with the nominal behaviour, both signals need to be valid. If both inputs are invalid, then the components fail, changing its state to the failure state TotalLoss, if one is valid and the other is invalid, then it proceeds to another failure state, PartialInput.

The diagrams in Figure 8-3 to Figure 8-6 represent the structure and behaviour of a source model annotated with safety characteristics needed to perform safety analysis. As shown, it is sufficient to use standard elements of SysML stereotyped with the DAM profile, an extension of the standard MARTE profile. A specific approach for modeling some interesting safety concepts are provided for multiple redundant inputs and multiple different inputs.

### 8.1.2 CFT Model

After the system has been modeled and annotated with failure information, we perform the SysML to CFT transformation. Figure 8-7 shows the mapping of the behaviour of the GPS Receiver component to a CFT model.

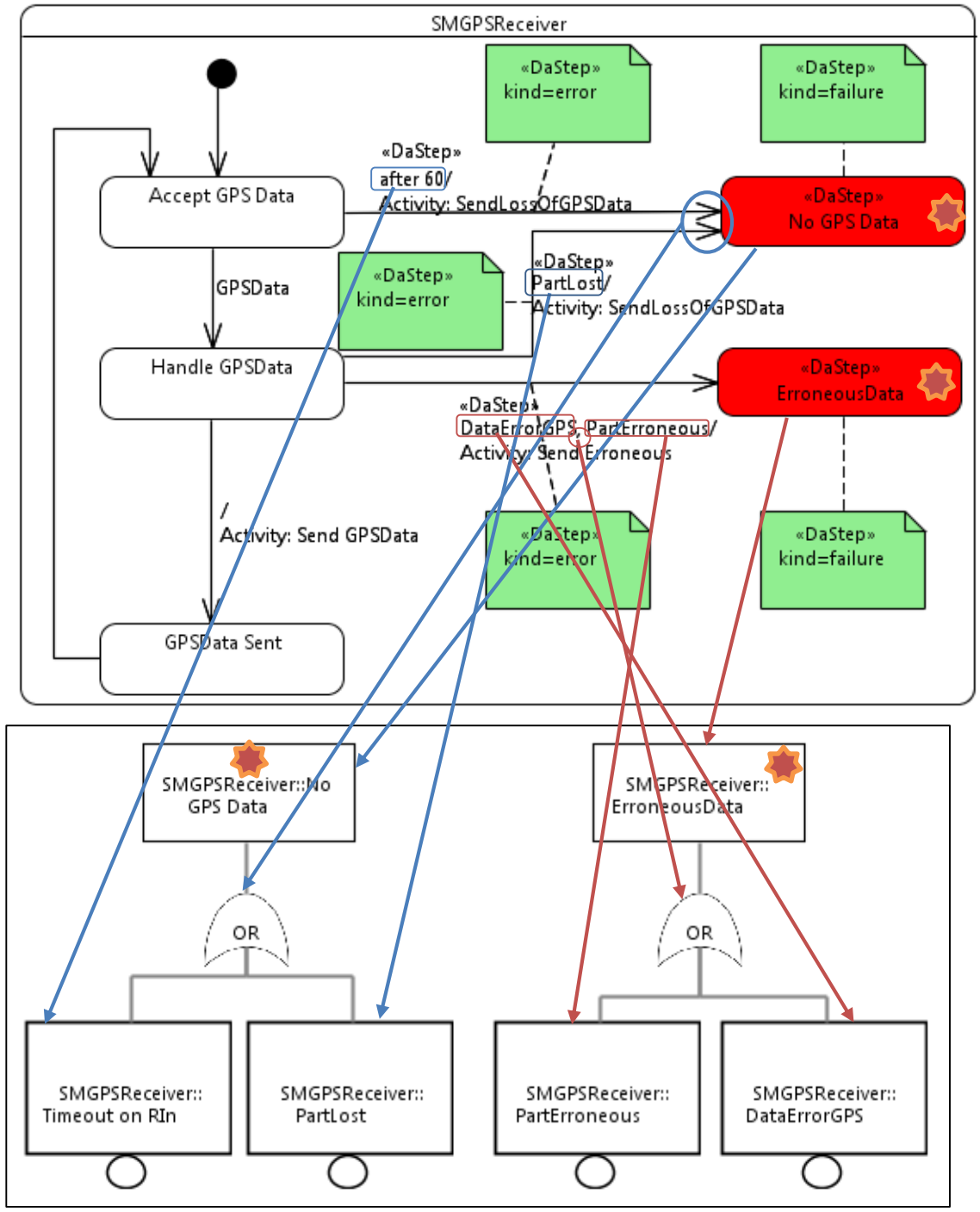



**Figure 8-7: Mapping between SysML and CFT models**

The Channel component generated the CFT model shown in Figure 8-8. Figure 8-9 depicts the CFT for the Channel interface component. Figure 8-10 shows the CFT for the Processing unit.

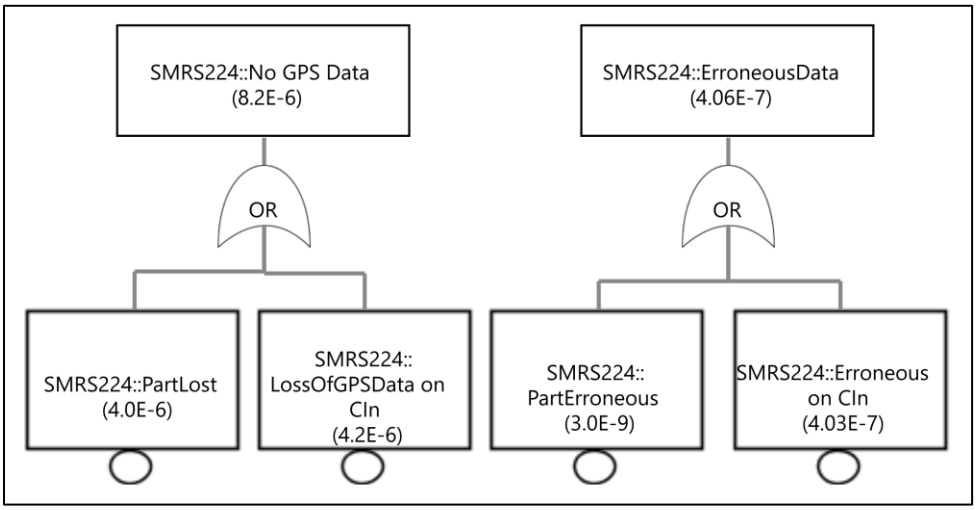



**Figure 8-8: CFT model: Channel**

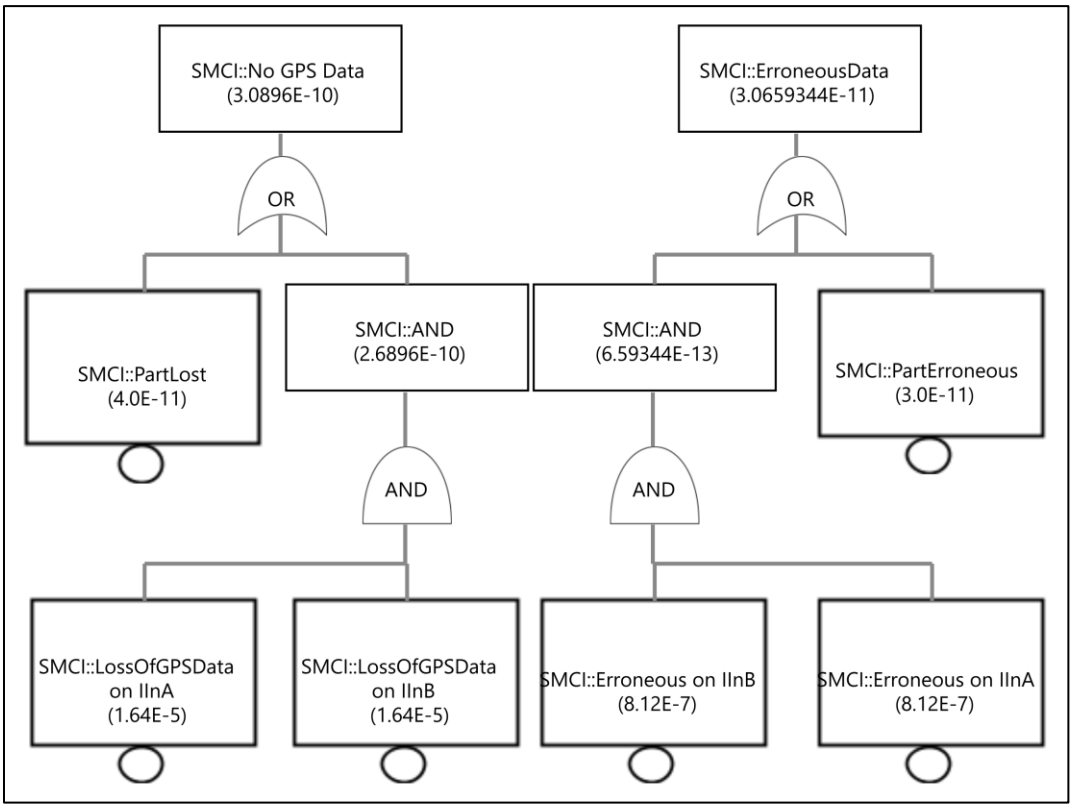



**Figure 8-9: CFT model: Channel interface**

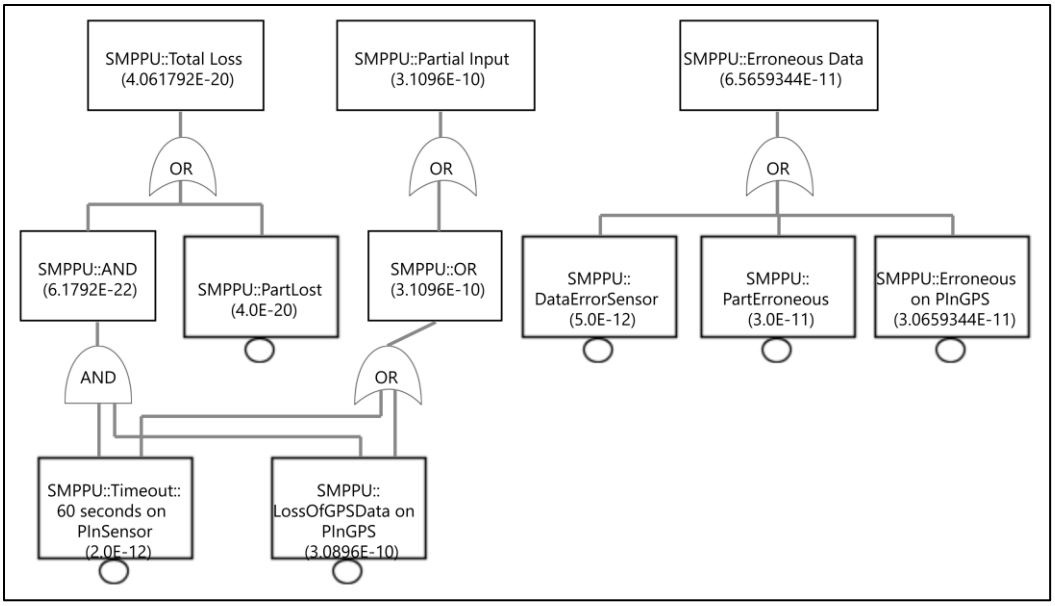



**Figure 8-10: CFT model: Processing unit**

### 8.1.3 SFT model

Figure 8-11 shows the mapping between the SysML source model internal structure depicted as IBD, the intermediate CFTs of the GPS Receiver and the Channel components and the SFT for the Erroneous Data top event. The black text boxes follow the naming convention of the events, and the direction of the arrows represent the use of an element as an input to the identified transformation and the outgoing arrow for the elements generated from the transformation. The complete SFT model with all top events is presented in Figure 8-12.

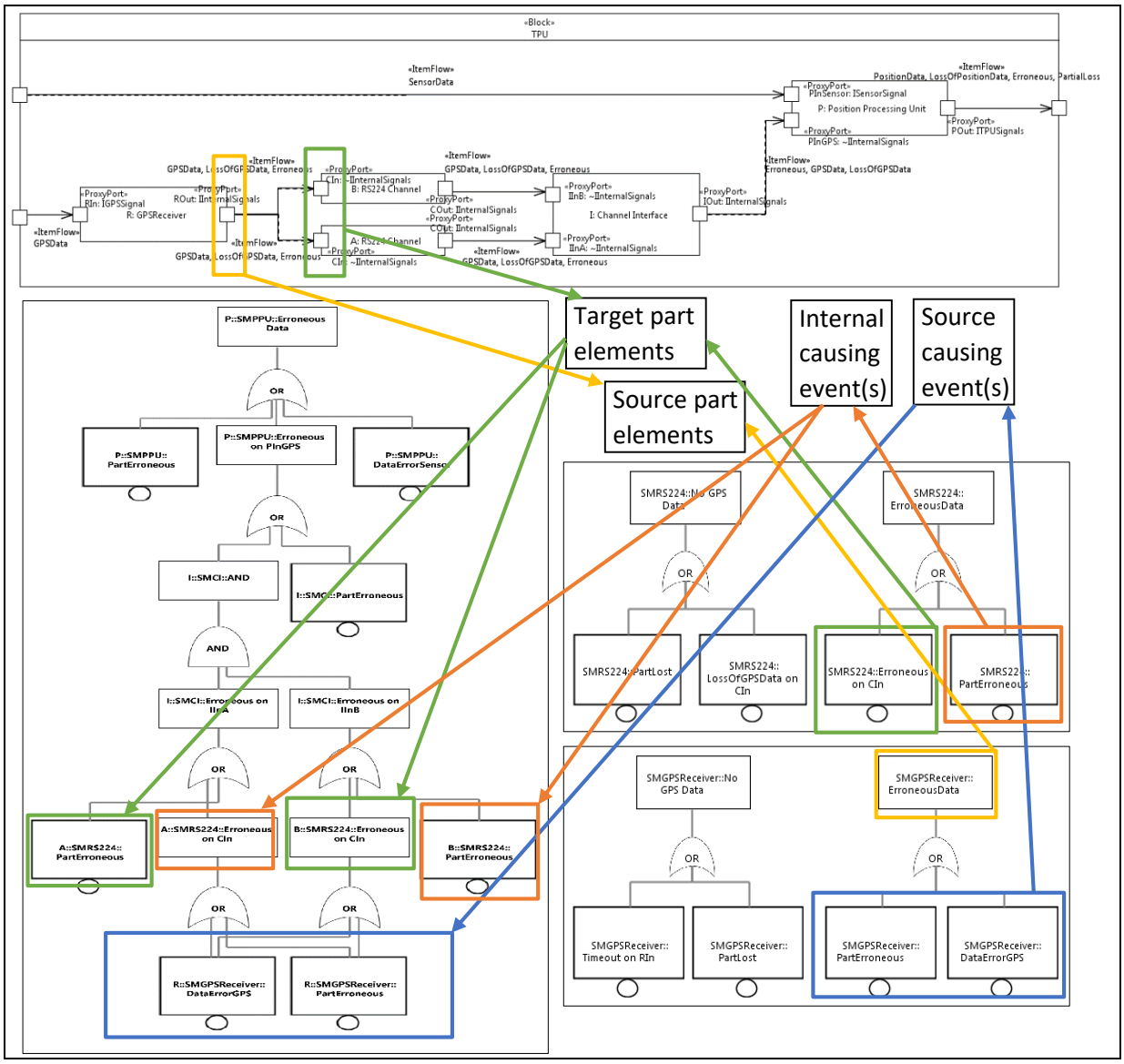



**Figure 8-11: Component-level CFT to system-level FT mapping**

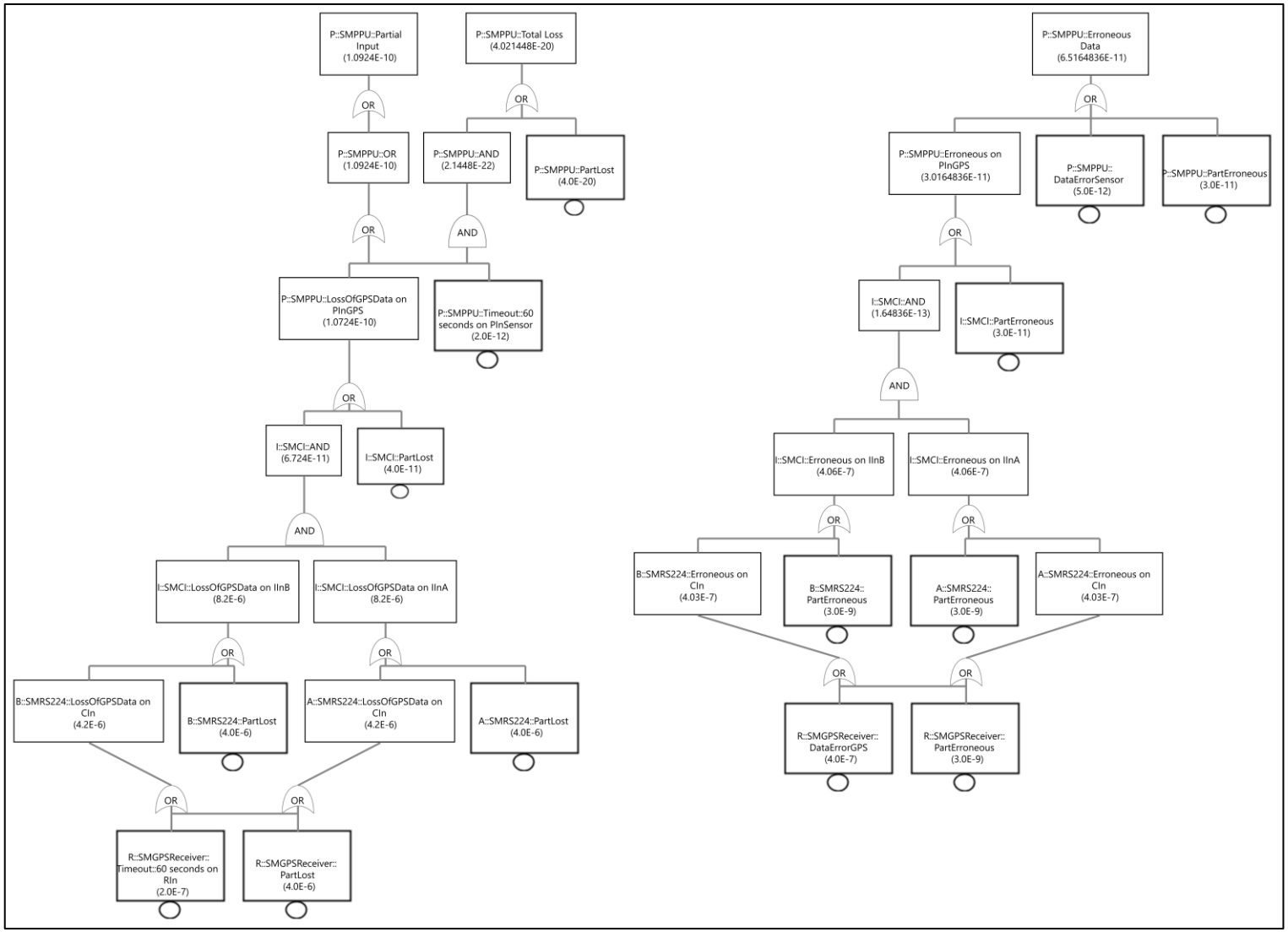



**Figure 8-12: The synthesized SFT for the system.**

### 8.1.4 Analysis

After generating the FT at the system level which represents the failure behaviour of the system, FTA can be used to analyze the safety properties of the system. The *qualitative evaluation* of the FT provides a cut set for the system failure top event.

A cut set contains the set of basic events whose simultaneous occurrence can cause the top unwanted event to occur. The basic events are events that cannot be further decomposed into a more basic event due to their nature or to the limited information available.

In Figure 8-13, three nested tables show multiple cut sets for the system, which correspond to the identified three top failure events for the TPU: nested table A) represents the cut sets for the TPU Total Loss of Position Data (TLPD) event; table B) represents the cut sets for the TPU Partial Loss of Position Data (PLPD) event; and C) represents the TPU's Erroneous Position Data (EPD) event.

Nested table A lists the cut sets of the FT with TLPD as a top event. Cut set 0, contains the event P::SMPPU::PartLost only, which is the failure of block instance P, denoting that the occurrence of this event will lead to the occurrence of the top event TLPD. The failure of P is seen as a single point of failure. Considering Cut set 1, consisting of the two events 'R::SMGPSReceiver::Timeout::60 seconds on Rin' and 'P::SMPPU::Timeout::60 seconds on PInSensor', which represent not receiving the GPS signal data and not receiving the Sensor signal data respectively. Hence, for TLPD to occur as a result of cut set 1, both of these events have to coincide.

**A) TLPD**

| Identifier | Probability |
|---|---|
| **Cutset #0** | 8.0E-18 |
| R::SMGPSReceiver::PartLost | 0.000004 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2E-12 |
| **Cutset #1** | 3.2E-23 |
| B::SMRS224::PartLost | 0.000004 |
| A::SMRS224::PartLost | 0.000004 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2E-12 |
| **Cutset #2** | 4.0E-19 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 0.0000002 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2E-12 |
| **Cutset #3** | 8.0E-23 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2E-12 |
| I::SMCI::PartLost | 4E-11 |
| **Cutset #4** | 4.0E-20 |
| P::SMPPU::PartLost | 4E-20 |

**B) PLPD**

| Identifier | Probability |
|---|---|
| Cutset #0 | 4.0E-6 |
| R::SMGPSReceiver::PartLost | 0.000004 |
| Cutset #1 | 4.0E-11 |
| I::SMCI::PartLost | 4E-11 |
| Cutset #2 | 1.6E-11 |
| B::SMRS224::PartLost | 0.000004 |
| A::SMRS224::PartLost | 0.000004 |
| Cutset #3 | 2.0E-12 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2E-12 |
| Cutset #4 | 2.0E-7 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 0.0000002 |

**C) EPD**

| Identifier | Probability |
|---|---|
| Cutset #0 | 9.0E-18 |
| B::SMRS224::PartErroneous | 0.000000003 |
| A::SMRS224::PartErroneous | 0.000000003 |
| Cutset #1 | 3.0E-11 |
| P::SMPPU::PartErroneous | 3E-11 |
| Cutset #2 | 3.0E-9 |
| R::SMGPSReceiver::PartErroneous | 0.000000003 |
| Cutset #3 | 3.0E-11 |
| I::SMCI::PartErroneous | 3E-11 |
| Cutset #4 | 5.0E-12 |
| P::SMPPU::DataErrorSensor | 5E-12 |
| Cutset #5 | 4.0E-7 |
| R::SMGPSReceiver::DataErrorGPS | 0.0000004 |

**Figure 8-13: The cut sets for the case study system**

Useful information can be obtained from the cut sets. A type of information is the prioritization of the cut sets that can be obtained based on the number of events in each cut set. In this example, cut set 0 will have the highest priority because it has a single event. Cut sets 1, 2, and 4 would have a lower priority for containing two events each, and the remaining cut set 3 will have the least priority for having three events. Another beneficial information that can be obtained is the significance of the contribution of a specific basic event toward the top event. As shown in the nested table A, the event 'P::SMPPU::Timeout::60 seconds on PInSensor' is a participant in 4 of the 5 cut sets leading to the TLPD top event.

Nested table B lists the cut sets of the FT with PLPD as a top event. Four of the five sets contain a single event, and one contains two events. Comparing to the TLPD, the PLPD is more likely to occur due to the nature of its cut sets.

Nested table C, contains the cutsets of the FT with EPD as a top event. All the cutsets except the last contain a single event, meaning that the occurrence of any of those events can cause the top event EPD to occur. As for the last cutset, it contains two events that need both to occur in order to lead to the top event.

### 8.1.5 Using redundant components

An enhancement to the system model is proposed as an alternative design. The FTA will be used to evaluate the two designs providing valuable information to increase the confidence in selecting one of them. The alternative design proposes to add a redundant GPS Receiver (R2) component. The modeling of this alternative design includes the following changes:

1) The modification of the BDD by adding another association instance (see Figure 8-14).

2) The modification of the IBD to represent the interconnection of the new instance within the internal structure of the TPU, as shown in Figure 8-15. As for the behaviour of the system, that is modeled using state machines, no modification is needed as the behaviour of the block that is being replicated is already provided.

Because only the structure view is modified, and the behaviour model has not been changed, we need to perform only the second transformation step; the synthesizing of the SFT model. Once the transformation is complete, the newly synthesized SFT model is shown in Figure 8-16.

A qualitative evaluation of the alternative structure of the case study is presented in Figure 8-17. The proposed alternative structure of the system can be evaluated using these resulting cut sets qualitatively for the same top events. Nested table A contains the cutsets for the top event TLPD. Each cutset contains the least number of events if occurred leads to the occurrence of the top event, the TLPD in this case. Eight out of the ten cutsets generated for revised design contain three events, whereas in the original design, one out of five cutsets had three events and three out of five had two events. A statement can be made that the alternate TPU design is now more resilient to the TLPD failure event.

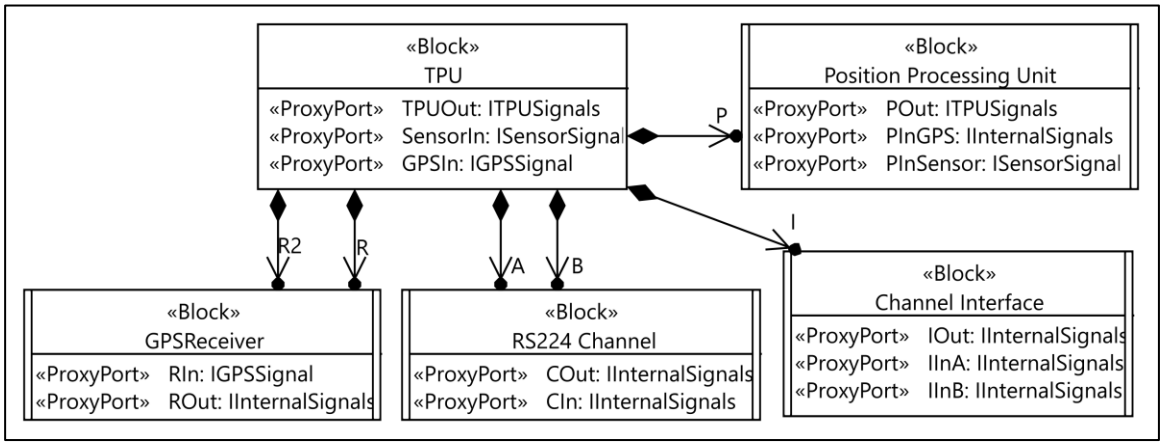



**Figure 8-14: A BDD for an alternative structure**

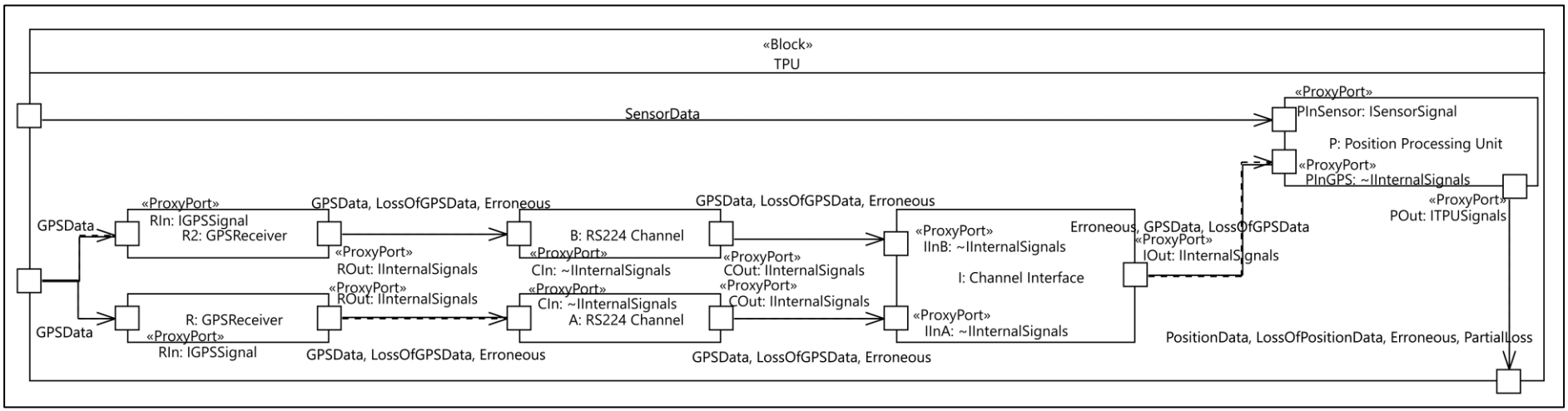



**Figure 8-15: An IBD for an alternative structure**

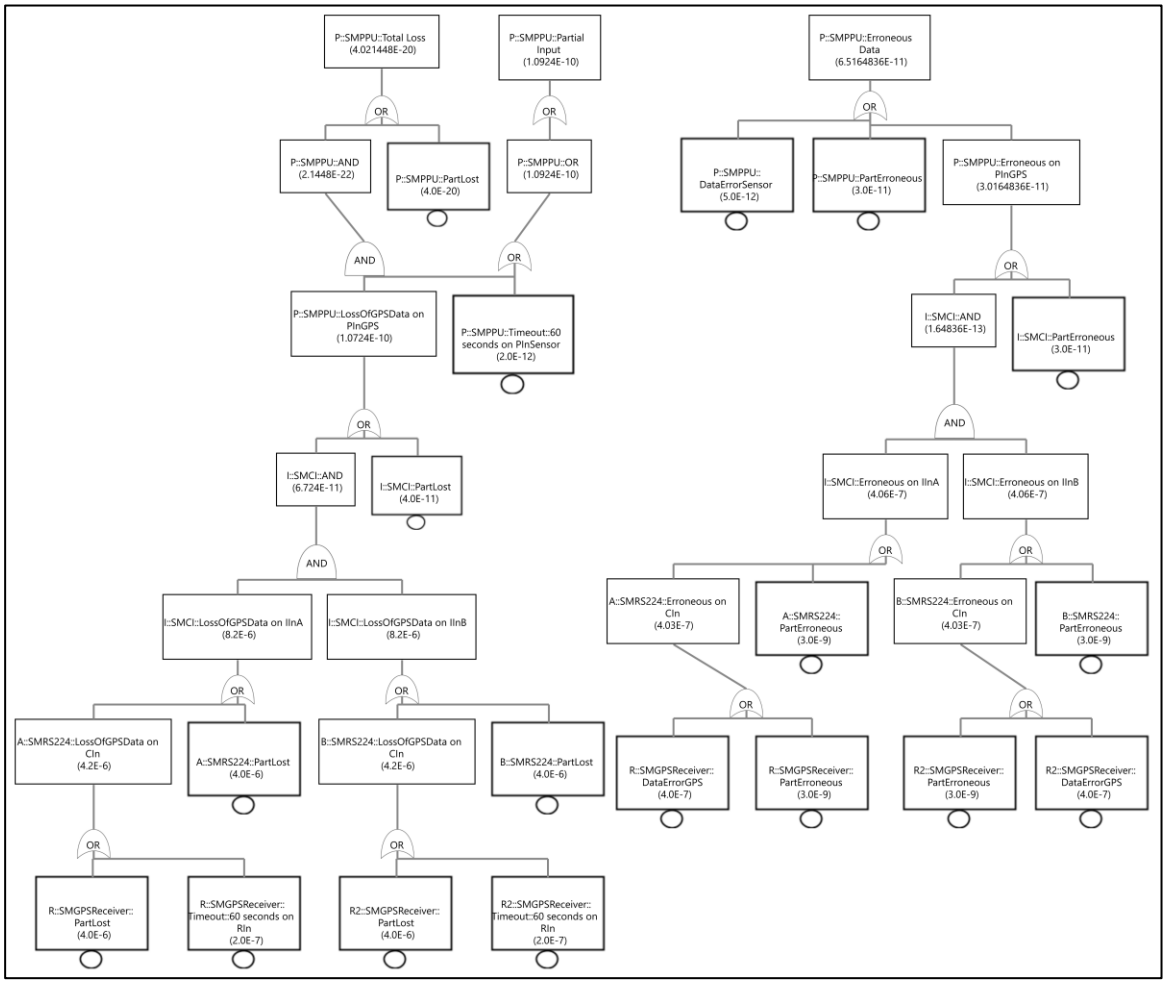



**Figure 8-16: FT for the alternative structure of the case study**

For another top event, the PLPD, a list of Cut sets is provided in the nested table B. When compared with the original design FTA analysis, a similar result can be concluded based on the number of events contained in the alternative design compared to the original design, which also favors the alternative design.

This comparison between the original and the revised designs show the usefulness of the proposed approach for evaluating a system with safety concerns and evaluating possible alternative designs helping in taking informative and quick design decisions.

**A) TLPD**

| Identifier | Probability |
|---|---|
| Cutset #0 | 1.60E-24 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| B::SMRS224::PartLost | 4.00E-06 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #1 | 3.2E-23 |
| R::SMGPSReceiver::PartLost | 4.00E-06 |
| R2::SMGPSReceiver::PartLost | 4.00E-06 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #2 | 1.60E-24 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| R2::SMGPSReceiver::PartLost | 4.00E-06 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #3 | 3.2E-23 |
| R::SMGPSReceiver::PartLost | 4.00E-06 |
| B::SMRS224::PartLost | 4.00E-06 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #4 | 1.60E-24 |
| R2::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| A::SMRS224::PartLost | 4.00E-06 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #5 | 3.2E-23 |
| B::SMRS224::PartLost | 4.00E-06 |
| A::SMRS224::PartLost | 4.00E-06 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #6 | 4.0E-20 |
| P::SMPPU::PartLost | 4.00E-20 |
| Cutset #7 | 8.0E-23 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| I::SMCI::PartLost | 4.00E-11 |
| Cutset #8 | 1.60E-24 |
| R::SMGPSReceiver::PartLost | 4.00E-06 |
| R2::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #9 | 8.00E-26 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| R2::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #10 | 3.2E-23 |

**B) PLPD**

| Identifier | Probability |
|---|---|
| Cutset #0 | 8.00E-13 |
| R::SMGPSReceiver::PartLost | 4.00E-06 |
| R2::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| Cutset #1 | 4.00E-14 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| R2::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| Cutset #2 | 1.60E-11 |
| R::SMGPSReceiver::PartLost | 4.00E-06 |
| B::SMRS224::PartLost | 4.00E-06 |
| Cutset #3 | 1.60E-11 |
| B::SMRS224::PartLost | 4.00E-06 |
| A::SMRS224::PartLost | 4.00E-06 |
| Cutset #4 | 1.60E-11 |
| R2::SMGPSReceiver::PartLost | 4.00E-06 |
| A::SMRS224::PartLost | 4.00E-06 |
| Cutset #5 | 2.00E-12 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |
| Cutset #6 | 4.00E-11 |
| I::SMCI::PartLost | 4.00E-11 |
| Cutset #7 | 1.60E-11 |
| R::SMGPSReceiver::PartLost | 4.00E-06 |
| R2::SMGPSReceiver::PartLost | 4.00E-06 |
| Cutset #8 | 8.00E-13 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| B::SMRS224::PartLost | 4.00E-06 |
| Cutset #9 | 8.00E-13 |
| R2::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |
| A::SMRS224::PartLost | 4.00E-06 |
| Cutset #10 | 8.00E-13 |
| R::SMGPSReceiver::Timeout::60 seconds on RIn | 2.00E-07 |

| | |
|---|---|
| R2::SMGPSReceiver::PartLost | 4.00E-06 |
| A::SMRS224::PartLost | 4.00E-06 |
| P::SMPPU::Timeout::60 seconds on PInSensor | 2.00E-12 |

| | |
|---|---|
| R2::SMGPSReceiver::PartLost | 4.00E-06 |

**C) EPD**

| Identifier | Probability |
|---|---|
| Cutset #0 | 1.60E-13 |
| R2::SMGPSReceiver::DataErrorGPS | 4.00E-07 |
| R::SMGPSReceiver::DataErrorGPS | 4.00E-07 |
| Cutset #1 | 1.20E-15 |
| B::SMRS224::PartErroneous | 3.00E-09 |
| R::SMGPSReceiver::DataErrorGPS | 4.00E-07 |
| Cutset #2 | 9.00E-18 |
| R::SMGPSReceiver::PartErroneous | 3.00E-09 |
| R2::SMGPSReceiver::PartErroneous | 3.00E-09 |
| Cutset #3 | 9.00E-18 |
| R2::SMGPSReceiver::PartErroneous | 3.00E-09 |
| A::SMRS224::PartErroneous | 3.00E-09 |
| Cutset #4 | 1.20E-15 |
| R2::SMGPSReceiver::DataErrorGPS | 4.00E-07 |
| R::SMGPSReceiver::PartErroneous | 3.00E-09 |
| Cutset #5 | 1.20E-15 |
| R2::SMGPSReceiver::DataErrorGPS | 4.00E-07 |
| A::SMRS224::PartErroneous | 3.00E-09 |
| Cutset #6 | 9.00E-18 |
| B::SMRS224::PartErroneous | 3.00E-09 |
| R::SMGPSReceiver::PartErroneous | 3.00E-09 |
| Cutset #7 | 3.00E-11 |
| I::SMCI::PartErroneous | 3.00E-11 |
| Cutset #8 | 9.00E-18 |
| B::SMRS224::PartErroneous | 3.00E-09 |
| A::SMRS224::PartErroneous | 3.00E-09 |
| Cutset #9 | 5.00E-12 |
| P::SMPPU::DataErrorSensor | 5.00E-12 |
| Cutset #10 | 3.00E-11 |
| P::SMPPU::PartErroneous | 3.00E-11 |
| Cutset #11 | 1.20E-15 |
| R2::SMGPSReceiver::PartErroneous | 3.00E-09 |
| R::SMGPSReceiver::DataErrorGPS | 4.00E-07 |

**Figure 8-17: The cut sets for the case study system alternative structure**

### 8.1.6 FMEA model

In this subsection, we present the activity of applying FMEA to the TPU case study. The following table represents the FMEA results. The table list all the components of the TPU in the first column. The second column represents the failure modes of each component. The third column lists identified failure causes for the failure modes. The fourth column provides identified local effects, that is at the component level itself. The fifth column list system-level effect of the failure mode. The last column contains the occurrence probability of the failure mode.

**Table 8-1: FMEA model for the TPU case study**

| **System:** | | **TrackingProcessingUnit** | | | |
|---|---|---|---|---|---|
| | | | | | |
| **Component** | **Failure Mode** | **Failure Cause** | **Local Effects** | **System Effects** | **Occurrence (Prob)** |
| GPSReceiver | Timeout 60 seconds on RIn | Omission of Input | No GPS Data | Total Loss, Partial Input | 2.0E-7 |
| | DataErrorGPS | Internal failure | ErroneousData | Erroneous Data | 4.0E-7 |
| | PartErroneous | Internal failure | ErroneousData | Erroneous Data | 3.0E-9 |
| | PartLost | Internal failure | No GPS Data | Total Loss, Partial Input | 4.0E-6 |
| RS224 Channel | LossOfGPSData | SMGPSReceiver ::No GPS Data | No GPS Data | Total Loss, Partial Input | 4.2E-6 |
| | Erroneous | SMGPSReceiver ::ErroneousData | ErroneousData | Erroneous Data | 4.03E-7 |
| | PartErroneous | Internal failure | ErroneousData | Erroneous Data | 3.0E-9 |
| | PartLost | Internal failure | No GPS Data | Total Loss, Partial Input | 4.0E-6 |
| Channel Interface | PartErroneous | Internal failure | ErroneousData | Erroneous Data | 3.0E-11 |
| | Erroneous | SMRS224::Erro neousData | ErroneousData | Erroneous Data | 4.06E-7 |
| | PartLost | Internal failure | No GPS Data | Total Loss, Partial Input | 4.0E-11 |
| | LossOfGPSData | SMRS224::No GPS Data | No GPS Data | Total Loss, Partial Input | 8.2E-6 |
| | Erroneous | SMRS224::Erro neousData | ErroneousData | Erroneous Data | 4.06E-7 |
| | LossOfGPSData | SMRS224::No GPS Data | No GPS Data | Total Loss, Partial Input | 8.2E-6 |
| Position Processing Unit | PartLost | Internal failure | No GPS Data | Total Loss, Partial Input | 4.0E-20 |
| | Timeout 60 seconds on PInSensor | Omission of Input | Partial Input, Total Loss | Total Loss, Partial Input | 2.0E-12 |

| | | | | | |
|---|---|---|---|---|---|
| | DataErrorSensor | Internal failure | Erroneous Data | Erroneous Data | 5.0E-12 |
| | PartErroneous | Internal failure | ErroneousData | Erroneous Data | 3.0E-11 |
| | LossOfGPSData | SMCI::No GPS Data | Partial Input, Total Loss | Total Loss, Partial Input | 1.0724E-10 |
| | Erroneous | SMCI::ErroneousData | Erroneous Data | Erroneous Data | 3.0164836E-11 |

## 8.2 Elevator Case study

The Elevator case study is inspired from (Mustafiz, 2010), where an approach for the requirements elicitation and analysis of safety-critical systems is proposed, concentrating on getting a complete comprehension of the components and system failures. Taking the requirements elicitation and analysis results from (Mustafiz, 2010), we proceed with the design phase to model the system in SysML. For the design of the SysML model, we looked at the UML model presented in (Gomaa, 2000) for an elevator system. With the system modeled in SysML, our proposed approach can be applied to perform safety analysis of the system.

The elevator system in this section consists of the following: a) an elevator Control System (CS); b) a single elevator cabin, c) a motor controlled by the CS to move the cabin up and down between floors; d) a cabin door controlled by the CS to open and close it: e) a set of sensors to provide the CS with the required information; f) buttons on each floor to request the elevator; g) floor buttons inside the cabin to request a specific floor.

### 8.2.1 Source Model

The structural view of the model is represented by a BDD diagram specifying the blocks of the system and their relations (see Figure 8-18). Blocks are created for each component of the system. Hardware and software components are modeled as separate blocks; the allocation relation is used to depict the software to hardware allocation as a pair. The

following block pairs have been identified: {Control System, Processor}; {Motor SW, Motor}; {Door SW, Door}; {Emergency Brakes IF (EBIF), Emergency Brakes (EB)}; {Call Button IF, Call button}; {At Floor Sensor IF, At Floor Sensor}; {Approaching Floor Sensor IF, Approaching Floor Sensor}; {Weight Sensor IF, Weight Sensor}.

The component failure information is annotated in the model using the *DaComponent* stereotype. The internal structure of the Elevator system is modeled showing the interconnections between its parts. The Floor Button IF, the Call Button IF, the Approaching Floor Sensor IF, the At Floor Sensor IF and the Weight Sensor IF all provides regular input to the CS. Each has a single port and connected to a specific port at the CS via a realizing item flow showing the signals flowing from those parts into the CS.

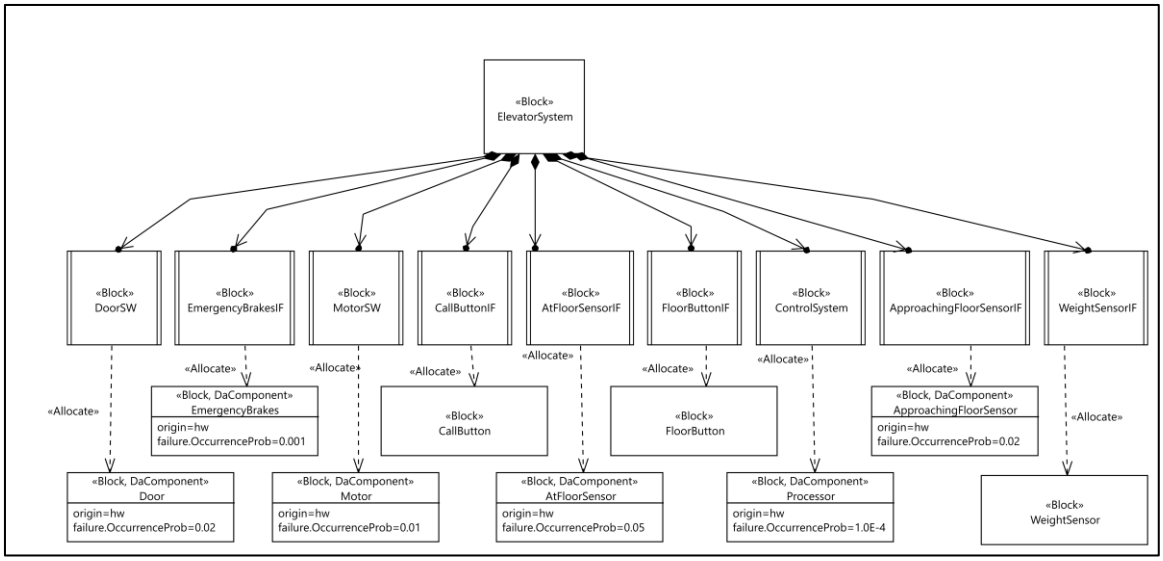



**Figure 8-18: Elevator System BDD**

The Motor SW, the Door SW and the Emergency Brakes IF each have two ports, both of which are connected to the CS via item flows showing the possible signal flowing in and out of each part. One of the ports is for accepting input from the CS and the other to provide the CS with the output.

The IBD in Figure 8-19 depicts the internal structure of the Elevator System.

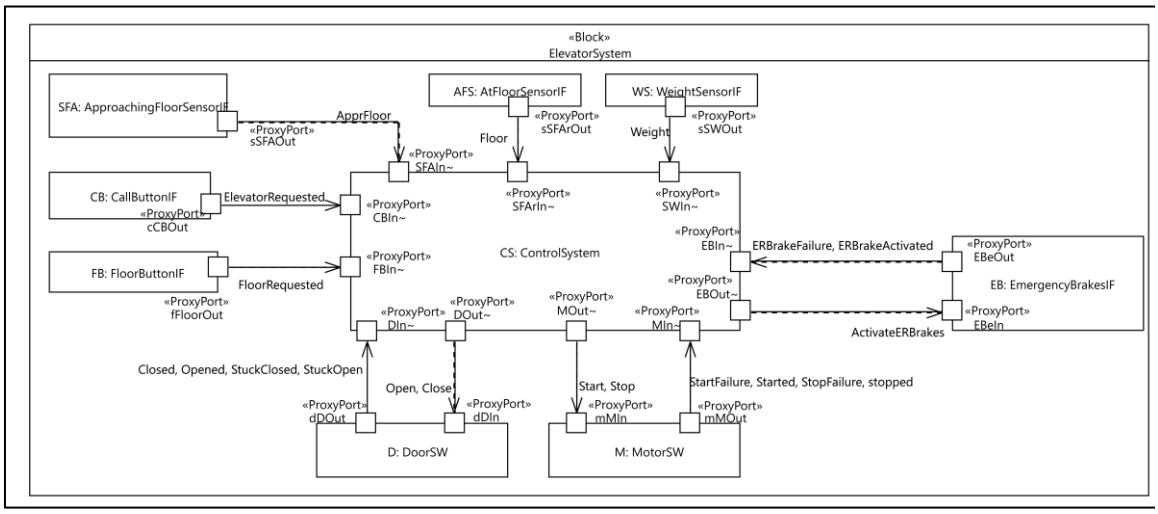



**Figure 8-19: Elevator System IBD**

Next, we model the behaviour view of the elevator system model. The CS, the Motor SW, the Door SW, and the EBIF have their behaviour specified in the requirements phase, which is modeled using SMs.

The Motor SW SM represents the normal behaviour of initially being idle until it receives the Start signal, then moves into Starting state, sends a “motor started” signal and then goes into Running state. Once it receives a Stop signal, it goes to Stopping state, sends a motor stopped signal upon completion and becomes idle. There are two failures modeled in the SM. The first is when the motor is starting: if after a specific time it does not start, a certain number of retries to start the motor are performed until it goes to normal behaviour or it reaches a threshold which sends a failure signal out. The other failure may occur when the motor is stopping, which is more critical. No retries are performed, and a failure signal is sent upon timing out while waiting for the motor to stop. The Motor SW state machine is depicted in Figure 8-20.

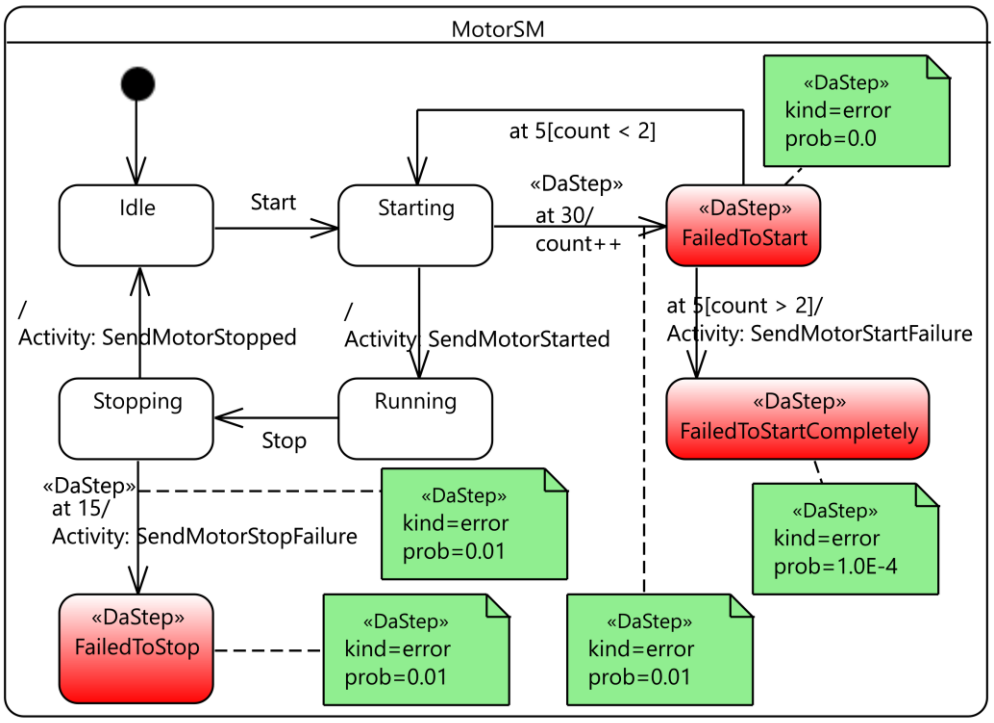



**Figure 8-20: Motor SW behavioural view**

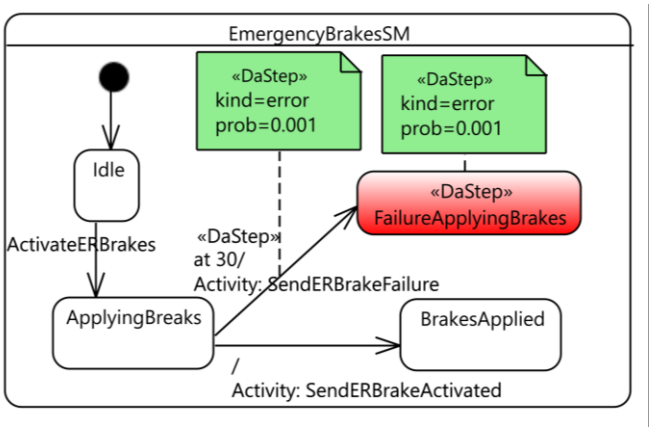



**Figure 8-21: EBIF behaviour view**

The EBIF normal behaviour consists of initially being idle until it receives an activate signal, where the emergency brakes are being applied; upon completion, it sends an activated signal. A failure behaviour is modeled when a timeout occurs upon applying the emergency brakes, which send a failure signal out. Figure 8-21 depicts this SM.

The normal behaviour of Door SW is moving between four states: Open, ClosingDoor, Closed, OpenningDoor. A failure may take place while closing the door; if a timeout occurs a retry is triggered for a certain threshold; if surpassed, a failure signal is sent out. Another failure behaviour is modeled while opening the door, which triggers retries of opening the door until a certain threshold is reached, at which time a failure signal is sent out. The described behaviour of the Door SW is presented as a state machine in Figure 8-22.

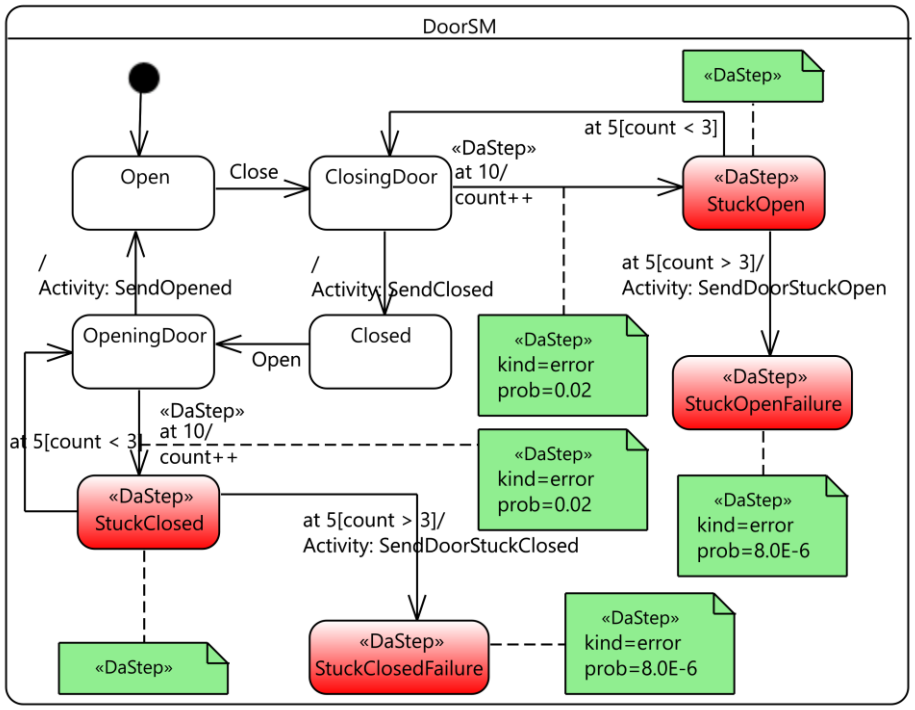



**Figure 8-22: Door SW behaviour view**

The CS, as the control component, has a more complex behaviour thank the Door, presented in Figure 8-23. It starts in an idle state with the cabin stopped at a floor, the door is open, and the motor is stopped. Closing the doors is triggered by pressing either a floor button or a call button for a specific floor. Once the doors close completely, the starting of the motor is triggered. Successful starting of the motor means that the cabin is moving, and the CS begins waiting for a signal signifying the approaching of the requested floor, which

then triggers stopping the motor. After the motor stopped and the destined floor is detected, the door is requested to open; when it is open, the elevator is back to idle state.

Some failures have been identified within this behaviour that has been added to the behavioural view. When the doors are closing, a failure signal can be received at the port with the Door IF. Also, a timeout can occur while waiting for the weight at the port with the Weight sensor. Starting the motor can result in receiving a failure signal from the port with the Motor IF. While the cabin is moving and the CS is waiting for the signal of approaching the destination, a timeout can occur along with passing the destination without prior notification of approaching it. Once the destination has been detected and the motor is requested to stop, a failure signal can be received from the port with the Motor IF, or a timeout can occur waiting for the signal of arriving at the destined floor. Both failures will trigger the activation of the emergency brakes. Activating the emergency brakes can fail with a failure signal or be completed with another signal, both received from the port with the EBIF. Each of the signals received while in Motor Stop Failure state leads to a different failure state. While opening the door, a failure of not being able to open the door can be received from the port with the Door IF.

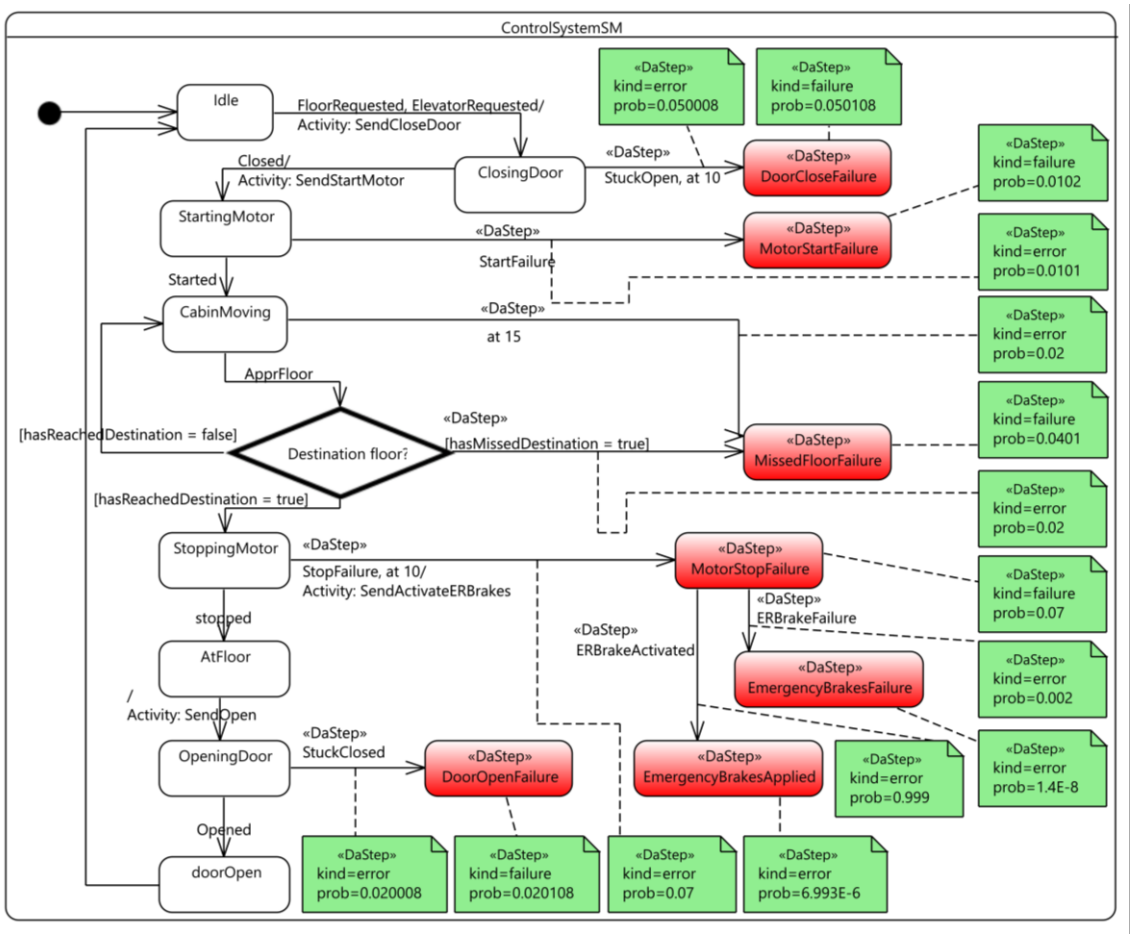



**Figure 8-23: The Control System behaviour view**

### 8.2.2 CFT Model

The following step is to perform the transformation of the SysML model behavioural view in CFT model. CFTs are generated for the components that have failure behaviour modeled and annotated. The results are shown in Figure 8-24, Figure 8-25, Figure 8-26, and Figure 8-27.

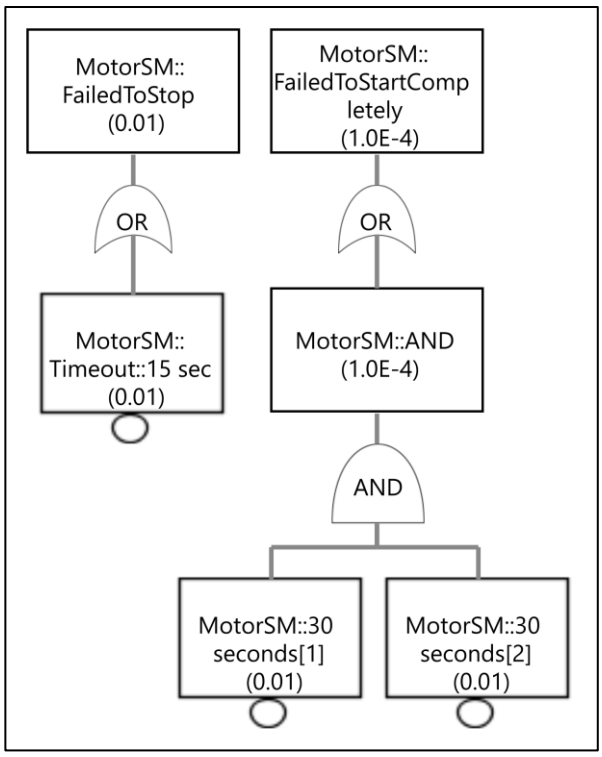



**Figure 8-24: CFT model: Motor**

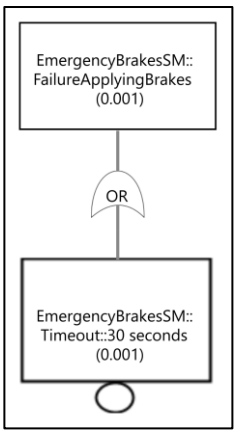



**Figure 8-25: CFT model: Emergency Brake**

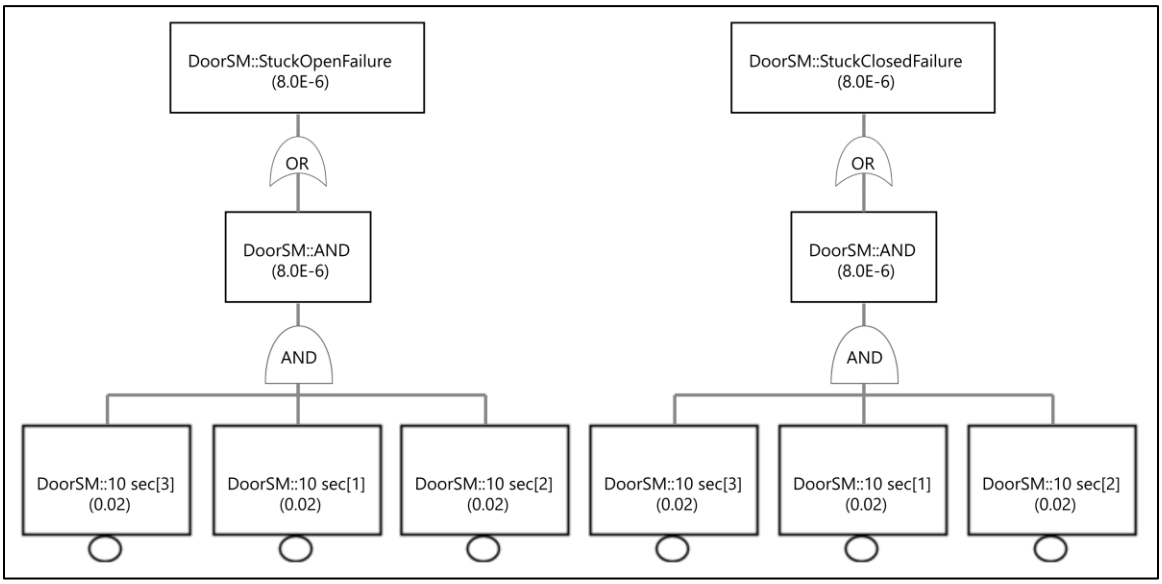



**Figure 8-26: CFT model: Door**

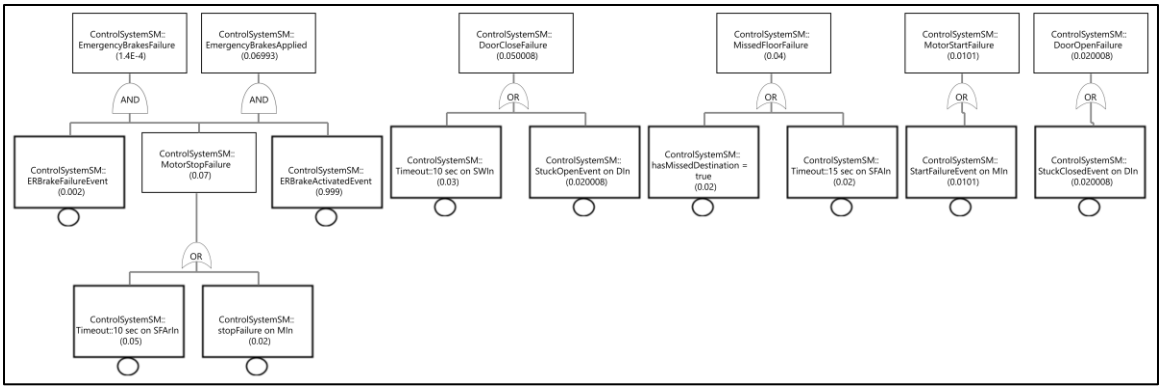



**Figure 8-27: CFT model: Control System**

Also, this transformation generates a trace model relating the SysML source elements to the CFT target elements.

### 8.2.3 SFT Model

After the component fault trees have been generated, we perform the second transformation step synthesizing an SFT model. This step takes as input the system SysML model, the CFT model, and the first transformation trace model.

The synthesized SFT model is presented in Figure 8-28.

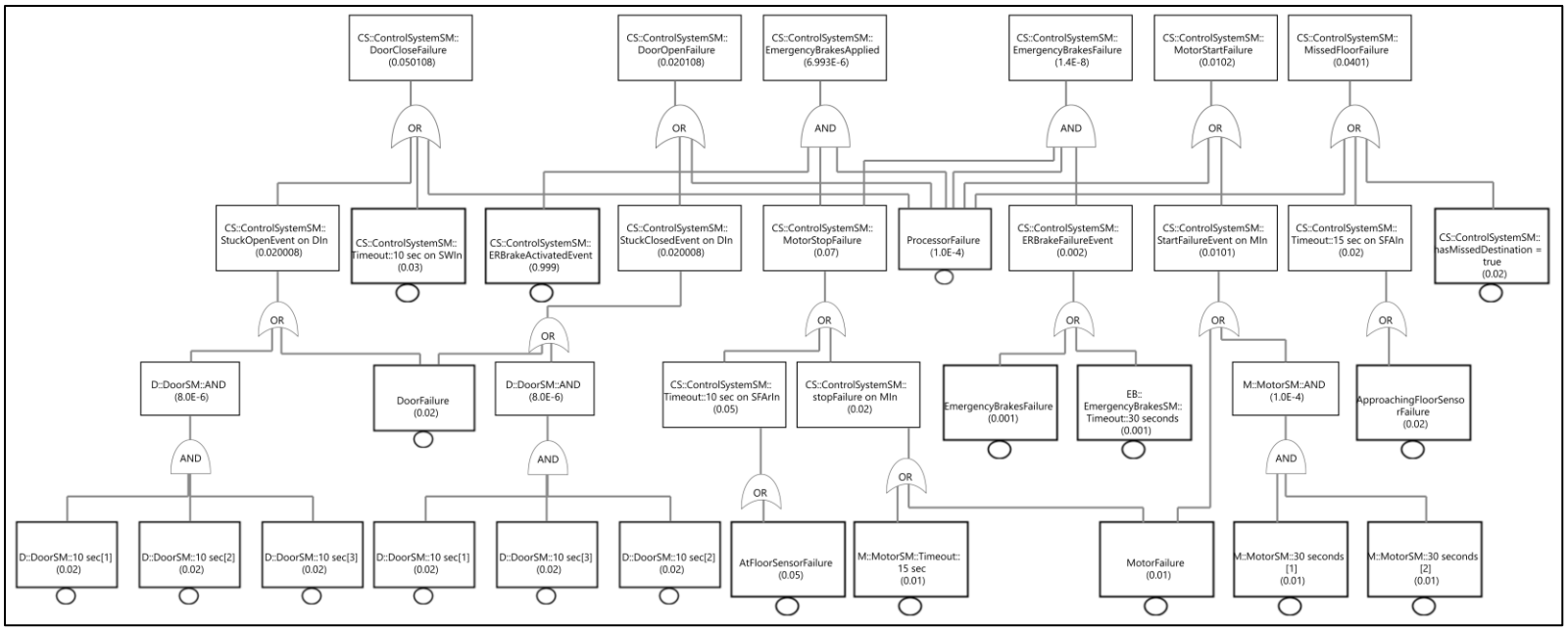



**Figure 8-28: Elevator System SFT model**

### 8.2.4 FMEA model

Now, we present the application of FMEA to the Elevator case study. Table 8-2 represents the FMEA results in tabular format. The first column list all the components of the Elevator system. The second column represents the failure modes of each component. The identified failure causes for the failure modes are in the third column. The identified local effects at the component level are provided in the fourth column. The fifth column list system-level effect of the failure mode. The last column contains the occurrence probability of the failure mode.

**Table 8-2: FMEA model for the Elevator System case study**

| **System:** | | **ElevatorSystem** | | | |
|---|---|---|---|---|---|
| | | | | | |
| **Component** | **Failure Mode** | **Failure Cause** | **Local Effects** | **System Effects** | **Occurrence (Prob)** |
| MotorSW | Timeout 15 sec | Omission of Input | FailedToStop | EmergencyBrakesApplied, EmergencyBrakesFailure | 0.01 |
| | Timeout 30 sec | Omission of Input | FailedToStartCompletely | MotorStartFailure | 0.01 |
| DoorSW | Timeout 10 sec | Omission of Input | StuckClosedFailure | DoorCloseFailure | 0.02 |
| | Timeout 10 sec | Omission of Input | StuckClosedFailure | DoorCloseFailure | 0.02 |
| ControlSystem | StuckOpenEvent | DoorSM::StuckOpenFailure | DoorCloseFailure | DoorCloseFailure | 0.020008 |
| | Timeout 10 sec on SWIn | Omission of Input | DoorCloseFailure | DoorCloseFailure | 0.03 |
| | stopFailure | MotorSM::FailedToStop | EmergencyBrakesApplied, EmergencyBrakesFailure | EmergencyBrakesApplied, EmergencyBrakesFailure | 0.02 |
| | Timeout 10 sec on SFArIn | Omission of Input | EmergencyBrakesApplied, EmergencyBrakesFailure | EmergencyBrakesApplied, EmergencyBrakesFailure | 0.05 |
| | StartFailureEvent | MotorSM::FailedToStartCompletely | MotorStartFailure | MotorStartFailure | 0.0101 |
| | StuckClosedEvent | DoorSM::StuckClosedFailure | DoorOpenFailure | DoorOpenFailure | 0.020008 |
| | ERBrakeFailureEvent | EmergencyBrakesSM::FailureApplyingBrakes | EmergencyBrakesFailure | EmergencyBrakesFailure | 0.002 |
| | ERBrakeActivatedEvent | EmergencyBrakesSM::BrakesApplied | EmergencyBrakesApplied | EmergencyBrakesApplied | 0.999 |

| | | | | | |
|---|---|---|---|---|---|
| | Timeout 15 sec on SFAIn | Omission of Input | MissedFloorFailure | MissedFloorFailure | 0.02 |
| EmergencyBrakesIF | Timeout 30 sec | Omission of Input | FailureApplyingBrakes | EmergencyBrakesFailure | 0.001 |
| Motor | Motor Failure | Internal failure | NA | EmergencyBrakesApplied, EmergencyBrakesFailure, MotorStartFailure | 0.01 |
| Door | Door Failure | Internal failure | NA | DoorCloseFailure, DoorOpenFailure | 0.02 |
| ApproachingFloorSensor | ApproachingFloorSensor Failure | Internal failure | NA | MissedFloorFailure | 0.02 |
| AtFloorSensor | AtFloorSensor Failure | Internal failure | NA | EmergencyBrakesApplied, EmergencyBrakesFailure | 0.05 |
| Processor | Processor Failure | Internal failure | NA | EmergencyBrakesApplied, DoorCloseFailure, EmergencyBrakesFailure, DoorOpenFailure, MissedFloorFailure, MotorStartFailure | 1.0E-4 |
| EmergencyBrakes | EmergencyBrakes Failure | Internal failure | NA | EmergencyBrakesFailure | 0.001 |

## 8.3 Mission Avionics System (MAS) case study

This case study was adapted from a highly referenced literature work (Dugan et al., 1992) in the area of dynamic fault trees. The system is a highly redundant system called Mission Avionics System (MAS). The system consists of four main components: a) Vehicle Management System (VMS) which provide airframe control and utility management system and control; b) Crew Station (CS) that displays information to the pilot and control the crew station activity; c) Mission and System Management (MSM) used to allocate resources for real-time control functions; and d) Scene and Obstacle following (SO)

component. Also, the system has a memory component. All of these components are connected using a set of buses.

The CS and the MSM both are assigned two dedicated identical processors, one active and the other as a hot spare. The VMS and SO require more processing power, so are assigned two sets of processors. The SO set has replicated processors, while the VMS set has triplicated processors. The memory has also replicated components. The buses are triplicated, with one set connecting the memory with the components other than the VMS called Background Data Bus (BDB), another set connecting the VMS with the other processers called Mission Management Bus (MMB), and the last connecting the VMS sets called Vehicle Management Bus (VMSBus).

### 8.3.1 Source Model

We modeled the described system in SysML, which only provides a structural view of the system. The system is presented as a BDD in Figure 8-29.

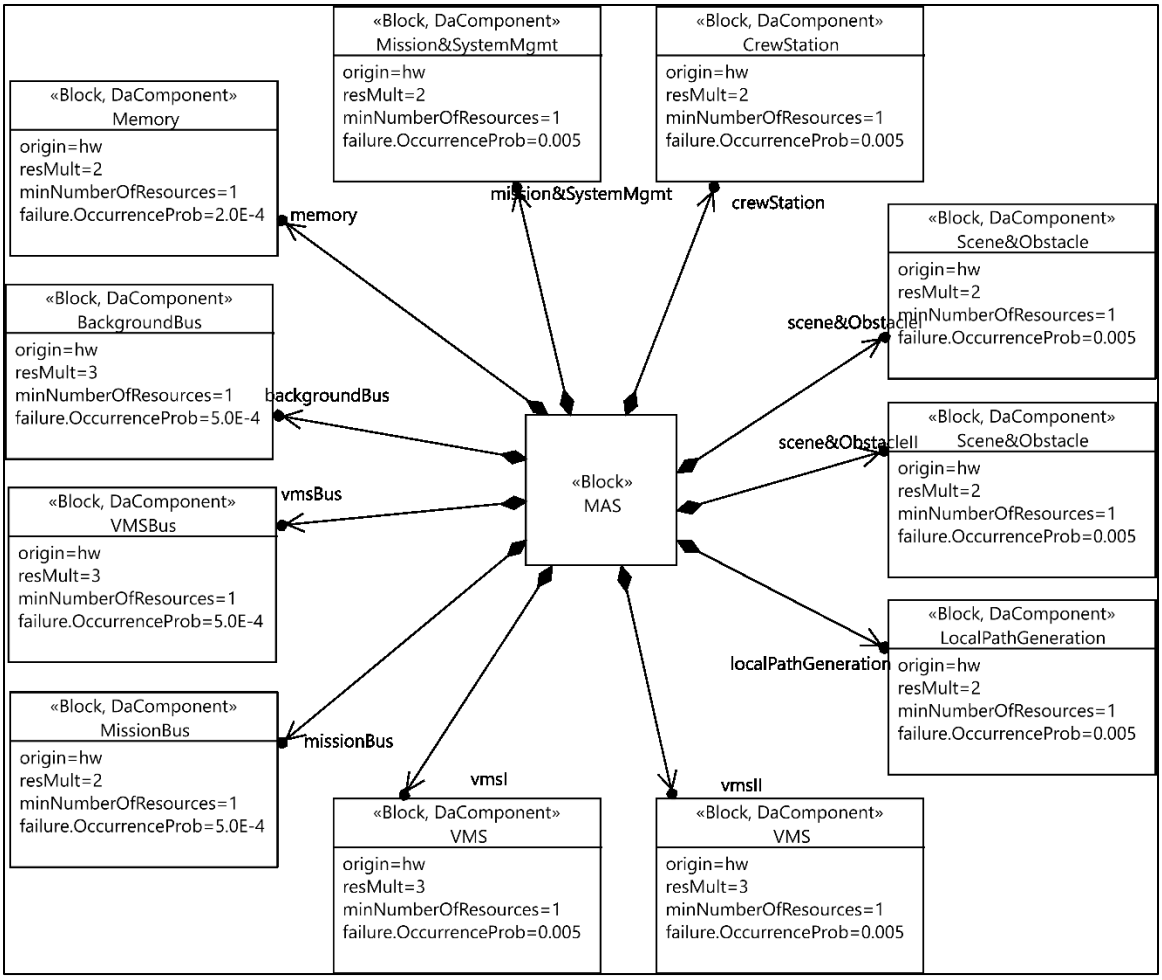



**Figure 8-29: MAS SysML BDD**

### 8.3.2 SFT Model

The first transformation step is not presented due to the nature of the system that it only provides a structural view. If the first transformation is executed, it will result in empty CFT models. Performing the second transformation on the MAS to synthesize an SFT model, results in the model presented in Figure 8-30. This SFT obtained by our approach is equivalent to the FT presented in (Dugan et al., 1992) for the MAS system.

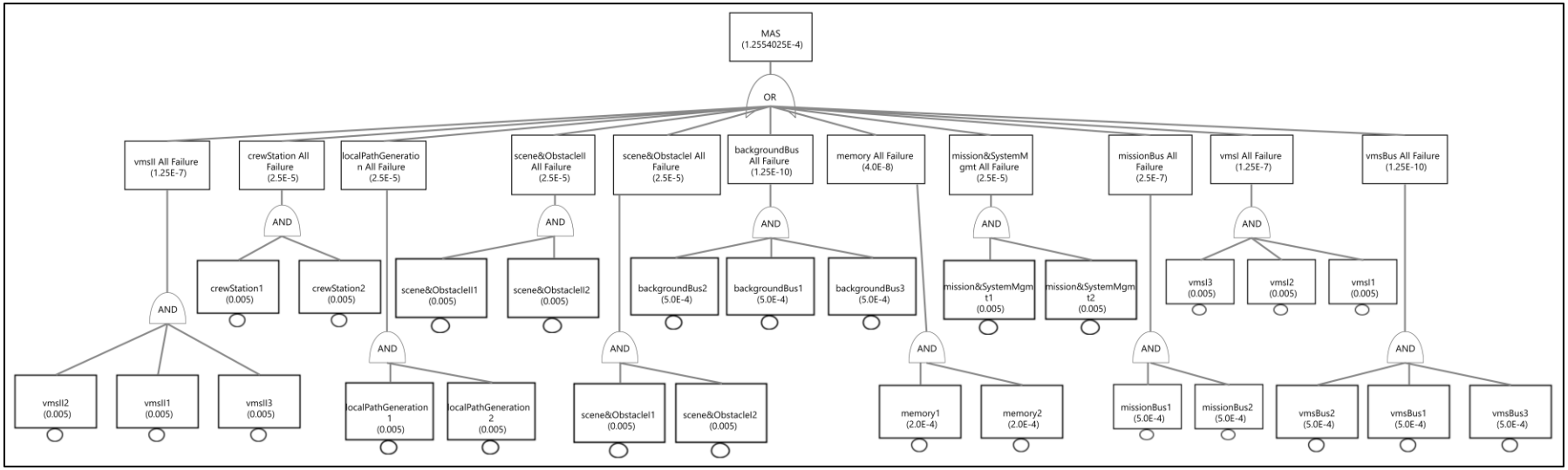



**Figure 8-30: MAS SFT model**

### 8.3.3 FMEA model

In this subsection, we apply FMEA to the MAS case study. The FMEA table is presented in Table 8-3. All the components of the MAS are listed in the first column. The second column represents the failure modes of each component. The third column lists identified failure causes for the failure modes. The fourth column provides identified local effects, that is at the component level itself. The fifth column list system-level effect of the failure mode. The last column contains the occurrence probability of the failure mode.

**Table 8-3: FMEA model for the MAS case study**

| **System:** | | **MissionAvionicsSystem** | | | |
|---|---|---|---|---|---|
| | | | | | |
| **Component** | **Failure Mode** | **Failure Cause** | **Local Effects** | **System Effects** | **Occurrence (Prob)** |
| Memory | Memory Failure * 2 | Internal failure | NA | MAS Failure | 4.0E-4 |
| VMSBus | VMSBus Failure * 3 | Internal failure | NA | MAS Failure | 0.0015 |
| MissionBus | MissionBus Failure * 2 | Internal failure | NA | MAS Failure | 0.001 |
| VMS | VMS Failure * 3 | Internal failure | NA | MAS Failure | 0.015 |
| LocalPathGeneration | LocalPathGeneration Failure * 2 | Internal failure | NA | MAS Failure | 0.01 |
| Scene&Obstacle | Scene&Obstacle Failure * 2 | Internal failure | NA | MAS Failure | 0.01 |
| CrewStation | CrewStation Failure * 2 | Internal failure | NA | MAS Failure | 0.01 |
| Mission&SystemMgmt | Mission&SystemMgmt Failure * 2 | Internal failure | NA | MAS Failure | 0.01 |
| BackgroundBus | BackgroundBus Failure * 3 | Internal failure | NA | MAS Failure | 0.0015 |

# 9 Chapter: Conclusions

In this thesis, an approach has been proposed to perform Model-Driven Safety Engineering (MDSE) of Safety Critical Systems (SCS) by integrating the Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA) safety analysis techniques within a Model Driven Engineering (MDE) methodology. The first contribution is the definition of failure mapping patterns between the failure-related elements of the source and target models. The purpose of the mapping patterns is to answer an important question when designing a transformation: "what needs to be transformed into what". This is shifting the focus from the implementation of the mapping with transformation languages (which introduce their own constraints) to the definition of the mapping itself.

Another novelty is that our approach uses, besides a structural view of the system, a behavioural view represented with state machines for modeling the component behaviour, where normal and failure behaviour are combined. Our approach extracts automatically the failure behavioural and structural information from the source model (based on the failure mapping patterns), in order to transform it into fault trees, which ignore the normal behaviour. Many works in literature are using a structural view and flow between blocks, but not state machines. Using state machines has the advantage of being able to define behaviour precisely, as detailed as necessary.

Our approach pays attention to a feature that was identified as missing in most of the other approaches, according to relevant surveys: feeding back the fault tree analysis results to the SysML model. This way, the designers can have access to safety information in the system model.

Furthermore, an intermediate model, Component-level Fault Tree (CFT), is proposed to be generated at the component level, allowing for reuse of modeling and analysis results. Subsequently, the System-level Fault Tree (SFT) model is generated by composing the CFT models. If changes occur in the system, only the CFT for changed components needs to be rebuilt from scratch, while the other can be reused.

Usually, the FMEA model is built by hand by human safety experts in very early stages of the system development. Later on, when the development evolves, the original FMEA model will fall out-of-synch. Our approach proposes to automatically generate the FMEA model from the current system source model, and the generated CFT and SFT models. This allows to re-synchronize the original system model with the fault-tree models and FMEA model at any phase during the development process, without error-prone human intervention.

These transformations were implemented using a model management framework called Epsilon (Kolovos et al., 2017), which is a family of model-based languages specialized for different tasks. We used the Epsilon Transformation Language (ETL) to implement the CFT and SFT model-to-model transformations. The first transformation to generate CFT consists of twelve rules and eleven operations, while the second transformation synthesizing SFT consists of six rules and twenty-three operations. The feedback of results for updating the source model after doing FTA analysis is implemented in the Epsilon Object Language (EOL). The FMEA model generation was realized using the Epsilon Generation Language (EGL) model-to-text transformation language. The implementation of these transformations has been orchestrated automatically using ANT. Finally, we evaluate the proposed approach against three different case studies taken from literature,

in order to avoid designer bias and to make sure that our method covers necessary modeling features.

## 9.1 Limitations

Even though the proposed approach accomplished its target, some limitations have been identified due to multiple factors. This subsection lists these limitations.

The tool EMFTA used to perform FTA implements the traditional fault tree model based on standards such as ARP4761. This fact limits the generated FTA models to using only standard FT, even though there exist some useful extended FT models. The lack of free open source tools to perform analysis on such extended models was the reason for relying on EMFTA.

The list of failure-mapping patterns introduced in section 4.1 may not be complete. These patterns were extracted from models described in the literature to ensure that necessary features are covered. Other patterns may be defined later and added to the pattern collection. However, this will require to extend the transformation algorithms.

Another point is related to the modeling language and the profiles used in the proposed approach. SysML seems to be gaining momentum, being used in literature and industry, but there exist some other modeling languages that may be used for SCS where safety analysis is required.

## 9.2 Future work

This subsection provides some directions and possible extensions to the proposed approach.

- The proposed approach requires some specific details to be found in the system source model. Some validations and warnings have been placed in the implementation, but a more comprehensive validation can enhance the process. A solution would be to use the Epsilon Validation Language (EVL), which allows to define a set of constraints over a model in order to validate it.
- Safety analysis could be integrated into the model-driven development process with other NFP analyses, such as performance, schedulability, reliability, availability, etc.
- A more practical solution for the users would be to implement the approach as a plugin for Papyrus, giving a more user-friendly experience. Also, it would allow for better integration with the FT analysis tool, the EMFTA.
- Support for model management with different types of files for source and target models, transformation files, analysis models, results could be developed in the future.

# Appendices

## Appendix A . FMEA Generation Template

This appendix provides the complete implementation of the Failure Modeling and Effects Analysis (FMEA) generation using Epsilon Generation Language (EGL). The code fragments from Appendix Code Fragment A-1 to Appendix Code Fragment A-7 present the EGL template. A-1 contains only static content that complies with the target spreadsheet application. A-2 also consist of static content except for line 67 which include an EGL dynamic statement wrapped in `[%= %]`. Appendix Code Fragment A-3 includes a mixed part of static and dynamic content, where all the dynamic content is wrapped within `[% %]` and `[%= %]`. The remaining of the code fragments of the template contain dynamic content consisting of functions used in section.

```
<?xml version="1.0"?>
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
          xmlns:o="urn:schemas-microsoft-com:office:office"
          xmlns:x="urn:schemas-microsoft-com:office:excel"
          xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
          xmlns:html="http://www.w3.org/TR/REC-html40">
  <DocumentProperties xmlns="urn:schemas-microsoft-com:office:office">
    <LastAuthor/>   <Created/>  <LastSaved/>  <Version>14.0</Version>
  </DocumentProperties>
  <OfficeDocumentSettings xmlns="urn:schemas-microsoft-com:office:office">
    <AllowPNG/> </OfficeDocumentSettings>
  <ExcelWorkbook xmlns="urn:schemas-microsoft-com:office:excel">
    <WindowHeight>16380</WindowHeight>
    <WindowWidth>38400</WindowWidth>
    <WindowTopX>0</WindowTopX>
    <WindowTopY>0</WindowTopY>
    <Date1904/>
    <ProtectStructure>False</ProtectStructure>
    <ProtectWindows>False</ProtectWindows>
  </ExcelWorkbook>
  <Styles>
    <Style ss:ID="Default" ss:Name="Normal">
      <Alignment ss:Vertical="Bottom"/>
      <Font ss:FontName="Calibri" x:Family="Swiss" ss:Size="12"
ss:Color="#000000"/>
      <Borders/>      <Interior/>      <NumberFormat/>      <Protection/>
    </Style>
    <Style ss:ID="s62">
      <Font ss:FontName="Calibri" x:Family="Swiss" ss:Size="12"
ss:Color="#000000"/>
    </Style>
    <Style ss:ID="s67">
      <Alignment ss:Horizontal="Left" ss:Vertical="Bottom" ss:WrapText="1"/>
      <Font ss:FontName="Calibri" x:Family="Swiss" ss:Size="12"
ss:Color="#000000"
          ss:Bold="1"/>
    </Style>
    <Style ss:ID="s69">
      <Alignment ss:Vertical="Bottom" ss:WrapText="1"/>
      <Font ss:FontName="Calibri" x:Family="Swiss" ss:Size="12"
ss:Color="#000000"
          ss:Bold="1"/>
    </Style>
    <Style ss:ID="s70">
      <Font ss:FontName="Calibri" x:Family="Swiss" ss:Size="12"
ss:Color="#000000"
          ss:Bold="1"/>
    </Style>
```

**Appendix Code Fragment A-1: FMEA Epsilon EGL Template (1 of 7)**

```
    <Style ss:ID="s71">
      <Alignment ss:Horizontal="Left" ss:Vertical="Bottom"/>
      <Font ss:FontName="Calibri" x:Family="Swiss" ss:Size="12"
ss:Color="#FFFFFF"
          ss:Bold="1"/>
      <Interior ss:Color="#333333" ss:Pattern="Solid"/>
    </Style>
    <Style ss:ID="s72">
      <Alignment ss:Vertical="Top" ss:WrapText="1"/>
      <Font ss:FontName="Calibri" x:Family="Swiss" ss:Size="12"
ss:Color="#000000"/>
    </Style>
  </Styles>
<Worksheet ss:Name="FMEA">
  <Table ss:ExpandedColumnCount="6"
       x:FullColumns="1"
    x:FullRows="1"
    ss:StyleID="s62"
    ss:DefaultColumnWidth="150"
    ss:DefaultRowHeight="15">
    <Row>
      <Cell ss:MergeAcross="1" ss:StyleID="s67"><Data ss:Type="String">
        System:
     </Data></Cell>
      <Cell ss:MergeAcross="3" ss:StyleID="s67"><Data ss:Type="String">
        [%=SysML!Model.all().first().name%]
      </Data></Cell>
    </Row>
    <Row>
      <Cell ss:StyleID="s67"/>
      <Cell ss:StyleID="s67"/>
      <Cell ss:StyleID="s67"/>
      <Cell ss:StyleID="s69"/>
      <Cell ss:StyleID="s70"/>
      <Cell ss:StyleID="s69"/>
    </Row>
    <Row>
      <Cell ss:StyleID="s71"><Data ss:Type="String">Component</Data></Cell>
      <Cell ss:StyleID="s71"><Data ss:Type="String"> Failure
Mode</Data></Cell>
      <Cell ss:StyleID="s71"><Data ss:Type="String"> Failure
Cause</Data></Cell>
      <Cell ss:StyleID="s71"><Data ss:Type="String"> Local
Effects</Data></Cell>
      <Cell ss:StyleID="s71"><Data ss:Type="String"> System
Effects</Data></Cell>
      <Cell ss:StyleID="s71"><Data ss:Type="String">Occurrence
(Prob)</Data></Cell>
    </Row>
```

**Appendix Code Fragment A-2: FMEA Epsilon EGL Template (2 of 7)**

```
[%for (component in getAllComponents()) {
      var failureModes = getFailureModes(component);
      if (failureModes.isUndefined() or failureModes.size() < 1) {
            continue;
      }
%]
    <Row ss:AutoFitHeight="1">
      <Cell ss:StyleID="s72" ss:MergeDown="3"><Data ss:Type="String">
        [%=component.name%]</Data></Cell>
    [%for (failureMode in failureModes) { %]
      <Cell ss:StyleID="s72" ss:Index="2"><Data ss:Type="String">
        [%=failureMode.getName()%]</Data></Cell>
      <Cell ss:StyleID="s72" ss:Index="3"><Data ss:Type="String">
        [%=failureMode.getCause()%]</Data></Cell>
      <Cell ss:StyleID="s72" ss:Index="4"><Data ss:Type="String">
        [%=failureMode.getLocalEffect()%]</Data></Cell>
      <Cell ss:StyleID="s72" ss:Index="5"><Data ss:Type="String">
        [%=failureMode.getSystemEffect()%]</Data></Cell>
      <Cell ss:StyleID="s72" ss:Index="6"><Data ss:Type="String">
        [%=failureMode.getProbability()%]</Data></Cell>
      [%  if(failureModes.size() > 1 and hasMore) { %]
    </Row>
    <Row ss:AutoFitHeight="1">
        [%}
      }%]
    </Row>
  [%}%]
  </Table>
  <WorksheetOptions xmlns="urn:schemas-microsoft-com:office:excel">
    <Print>
      <ValidPrinterInfo/>
      <PaperSizeIndex>9</PaperSizeIndex>
      <HorizontalResolution>-4</HorizontalResolution>
      <VerticalResolution>-4</VerticalResolution>
    </Print>
    <PageLayoutZoom>0</PageLayoutZoom>
    <Selected/>
    <Panes>
      <Pane>
        <Number>3</Number>
        <ActiveRow>0</ActiveRow>
        <ActiveCol>2</ActiveCol>
      </Pane>
    </Panes>
    <ProtectObjects>False</ProtectObjects>
    <ProtectScenarios>False</ProtectScenarios>
  </WorksheetOptions>
</Worksheet>
</Workbook>
```

**Appendix Code Fragment A-3: FMEA Epsilon EGL Template (3 of 7)**

```
operation getAllComponents() {
  var result : Sequence;
  for(blk in SysML!Block.all) {
    var base = blk.base_class;
    if(base.hasFailureBehaviour()
      or base.hasStereotype("DaComponent")){
      result.add(base);
    }
  }
  return result;
}
operation getFailureModes(component : SysML!Class) {
  var result : OrderedSet;
  if (component.hasFailureBehaviour()) {
    var stateMachine = component.classifierBehavior;
    var triggers = SysML!Trigger.all.select(trgr
      | trgr.isContainedInStatemachine(stateMachine)
        and trgr.hasStereotype("DaStap"));
    for(trgr in triggers) {
      result.add(trgr);
    }
  } else {
    result.add(component);
  }
  return result;
}
operation SysML!Trigger getName() : String {
  var result = self.getEvent().name;
  if (self.getEvent().isTypeOf(SysML!TimeEvent) ) {
    result = "Timeout " + self.getEvent().name;
    if (self.getPorts().notEmpty()) {
      result = result + " on " + self.getPorts().first().name;
    }
    self.~cause = "Omission of Input";
  }
  return result;
}
operation SysML!Class getName() :String {
  var result :String = self.name + " Failure";
  if (self. getResourceMultiplicity() > 1) {
    result = result.concat(" * " + self. getResourceMultiplicity());
  }
  return result;
}
```

**Appendix Code Fragment A-4: FMEA Epsilon EGL Template (4 of 7)**

```
operation SysML!Trigger getCause() {
  var result;
  if (not self.getPorts().isEmpty()) {
    if (self.getEvent().isTypeOf(SysML!TimeEvent)) {
      result = "Omission of Input";
    } else {
      var states = self.getTriggerSources();
      for (state in states) {
        result = state.eContainer.eContainer.name + "::" + state.name;
        if (hasMore) {
          result += ", ";
        }
      }
    }
  } else {
    result = "Internal failure";
  }
  return result;
}
operation SysML!Class getCause() {
  return "Internal failure";
}
operation SysML!Trigger getTriggerSources () : Set(SysML!State) {
  var resultStates :Set(SysML!State);
  var resultEvents :Set(CFTA!Event);
  var transitions = SysML!Transition.all.select(transition |
    transition.getEffect().isDefined()
    and transition.getEffect().isTypeOf(SysML!Activity));
  var sourceEnds = getTriggerPortMatchingConnectorSourceEnd(
    self.getPorts().first());
  for (sourceEnd in sourceEnds) {
    var sourcetrns = transitions.select(transition |
      not transition.effect.getOwnedElements().select(effect |
        effect.isTypeOf(SysML!SendSignalAction)
        and effect.onPort == sourceEnd.getRole()
        and effect.signal = self.event.signal).isEmpty());
    if (not sourcetrns.isEmpty()) {
      for (sourcetrn in sourcetrns) {
        resultStates.addAll(SysML!State.all.select(state
          | state.incomings.includes(sourcetrn)));
      }
    }
  }
  return resultStates;
}
```

**Appendix Code Fragment A-5: FMEA Epsilon EGL Template (5 of 7)**

```
operation SysML!Trigger getLocalEffect() : String {
  var basic = self.getCFTABySysMLElement();
  var effects = basic.getFailureModeLocalEffect();
  var result = "";
  for(effect in effects) {
    var src = effect.getSysMLByCFTAElement();
    result = result.concat(src.name);
    if(hasMore) {
      result = result.concat(",&#13;");
    }
  }
  return result;
}
operation SysML!Class getLocalEffect() : String {
  return "NA";
}
operation SysML!Trigger getSystemEffect() {
    var result = "";
    var sfta = self.getSFTAElement();
    var effects = sfta.getFailureModeSystemEffect();
    for (effect in effects) {
      var cfta = effect.getCFTABySFTAElement();
      var eff = cfta.getSysMLByCFTAElement();
      result = result.concat(eff.name);
      if(hasMore) {
        result = result.concat(",&#13;");
      }
    }
    if (result.length() == 0) {
        result = "Error!";
    }
    return result;
}
operation SysML!Class getSystemEffect() {
    var result = "";
    var sfta = self.getBlockByClass().getSFTAElement();
    var effects = sfta.getFailureModeSystemEffect();
    for (effect in effects) {
      var cfta = effect.getCFTABySFTAElement();
      if (cfta.isUndefined()) {
        result = effect.name.concat(" Failure");
      } else {
        result = result.concat(cfta.getSysMLByCFTAElement().name);
      }
      if(hasMore) {
        result = result.concat(",&#13;");
      }
    }
    if (result.length() == 0) {
        result = "Error!";
    }
    return result;
}
```

**Appendix Code Fragment A-6: FMEA Epsilon EGL Template (6 of 7)**

```
operation getTriggerPortMatchingConnectorSourceEnd (port: SysML!Port)
  : Set(SysML!Port) {
  var connectors = SysML!Connector.all.select(connector
    | not connector.getEnds().isEmpty()
    and connector.getKind() == SysML!ConnectorKind#assembly);
  var sources : Set (SysML!Port);
  for (connector in connectors) {
    for (end in connector.getEnds()) {
      if (end.getRole() = port) {
        sources.add(connector.getEnds().selectOne(inEnd|inEnd <> end));
      }
    }
  }
  return sources;
}

operation SysML!Trigger getProbability() {
  return self.getStereotypeAttribute("DaStep","prob").asDouble();
}

operation SysML!Class getProbability() {
  var result = self.getStereotypeAttribute("DaComponent",
      "failure.OccurrenceProb").asDouble();
  if (self. getResourceMultiplicity() > 1) {
    result = result * self. getResourceMultiplicity();
  }
  return result;
}

operation SysML!Class getResourceMultiplicity () : Any {
  return self.getStereotypeAttribute("DaComponent", "resMult");
}
```

**Appendix Code Fragment A-7: FMEA Epsilon EGL Template (7 of 7)**

## Bibliography


Adler, R., Domis, D., Höfig, K., Kemmann, S., Kuhn, T., Schwinn, J.-P., Trapp, M., 2010. Integration of Component Fault Trees into the UML, in: Models in Software Engineering. Presented at the International Conference on Model Driven Engineering Languages and Systems, Springer, Berlin, Heidelberg, pp. 312–327. https://doi.org/10.1007/978-3-642-21210-9_30

Al shboul, B., 2019. Bashar Al shboul PhD Thesis Implementation [WWW Document]. GitHub. URL https://github.com/basharaa/SysMLTransformation (accessed 4.6.19).

ARTEMIS-JU-100022, 2017. CHESS - Composition with guarantees for High-integrity Embedded Software components aSsembly [WWW Document]. URL http://www.chess-project.org/ (accessed 10.24.17).

Avižienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. Dependable Secure Comput. IEEE Trans. On 1, 11–33. https://doi.org/10.1109/TDSC.2004.2

Berardinelli, L., Bernardi, S., Cortellessa, V., Merseguer, J., 2009. UML Profiles for Non-functional Properties at Work: Analyzing Reliability, Availability and Performance., in: NFPinDSML@ MoDELS.

Bernardi, S., Merseguer, J., Petriu, D.C., 2013. Model-Driven Dependability Assessment of Software Systems. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-39512-3

Bernardi, S., Merseguer, J., Petriu, D.C., 2012. Dependability modeling and analysis of software systems specified with UML. ACM Comput. Surv. 45, 1–48. https://doi.org/10.1145/2379776.2379778

Berres, A., Schumann, H., 2016. Automatic generation of fault trees: A survey on methods and approaches, in: Risk, Reliability and Safety: Innovating Theory and Practice. CRC Press, pp. 2485–2492. https://doi.org/10.1201/9781315374987-377

Biggs, G., Sakamoto, T., Kotoku, T., 2014. A profile and tool for modelling safety information with design information in SysML. Softw. Syst. Model. 15, 147–178. https://doi.org/10.1007/s10270-014-0400-x

Bowles, J.B., Wan, C., 2001. Software failure modes and effects analysis for a small embedded control system, in: Reliability and Maintainability Symposium, 2001. Proceedings. Annual. IEEE, pp. 1–6.

Briones, J.F., Miguel, M.Á. de, Silva, J.P., Alonso, A., 2007. Application of Safety Analyses in Model Driven Development, in: Obermaisser, R., Nah, Y., Puschner, P., Rammig, F.J. (Eds.), Software Technologies for Embedded and Ubiquitous Systems, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 93–104. https://doi.org/10.1007/978-3-540-75664-4_10

CENELEC - EN 50129, 2018. Railway applications – Communication, signalling and processing systems – Safety related electronic systems for signalling [WWW Document]. URL https://standards.globalspec.com/std/1266373/en-50129 (accessed 4.3.19).

Chen, B., Avrunin, G.S., Clarke, L.A., Osterweil, L.J., 2006. Automatic fault tree derivation from little-jil process definitions, in: Software Process Change. Springer, pp. 150–158.

Ciardo, G., Lindemann, C., 1993. Analysis of deterministic and stochastic Petri nets, in: Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop On. IEEE, pp. 160–169.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2017. The Maude System [WWW Document]. URL http://maude.cs.illinois.edu/w/index.php?title=The_Maude_System (accessed 7.7.17).

CMU-SEI, 2017. EMFTA: EMF-based Fault-Tree Analysis Tool [WWW Document]. URL https://github.com/cmu-sei/emfta (accessed 9.20.17).

D'Ambrogio, A., Iazeolla, G., Mirandola, R., 2002. A method for the prediction of software reliability, in: Proceedings of the 6-Th IASTED Software Engineering and Applications Conference (SEA2002), Cambridge, MA.

David, P., Idasiak, V., Kratz, F., 2010. Reliability study of complex physical systems using SysML. Reliab. Eng. Syst. Saf. 95, 431–450. https://doi.org/10.1016/j.ress.2009.11.015

David, P., Idasiak, V., Kratz, F., 2009. Improving reliability studies with SysML, in: Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual. Presented at the Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual, pp. 527–532. https://doi.org/10.1109/RAMS.2009.4914731

Deji, P.D., 2016. Derivation of Failure Mode and Effects Analysis (FMEA) Table from UML Software Model by Epsilon Model Transformation (MASc Thesis).

Domis, D., Trapp, M., 2008. Integrating Safety Analyses and Component-Based Design, in: Harrison, M.D., Sujan, M.-A. (Eds.), Computer Safety, Reliability, and Security, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 58–71. https://doi.org/10.1007/978-3-540-87698-4_8

Douglass, B.P., 2009. Analyze system safety using UML within the IBM Rational Rhapsody environment. IBM Ration. White Pap. IBM Softw. Group.

Dugan, J.B., Bavuso, S.J., Boyd, M.A., 1992. Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans. Reliab. 41, 363–377. https://doi.org/10.1109/24.159800

Dugan, J.B., Sullivan, K.J., Coppit, D., Sullivan, K.J., Dugan, J.B., 1999. The Galileo Fault Tree Analysis Tool, in: In Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, pp. 232–235.

Ericson, C.A., 2011. Concise Encyclopedia of System Safety: Definition of Terms and Concepts. Wiley, Hoboken, UNITED STATES.

ESI ITI GmbH, 2017. SimulationX [WWW Document]. URL https://www.simulationx.com/simulation-software.html (accessed 12.11.17).

Feiler, P., Delange, J., 2017. Automated Fault Tree Analysis from AADL Models. ACM SIGAda Ada Lett. 36, 39–46. https://doi.org/10.1145/3092893.3092900

Flores, M., Malin, J.T., 2013. Failure Modes and Effects Analysis (FMEA) Assistant Tool Feasibility Study. Presented at the 6th International Association for the Advancement of Space Safety Conference: Safety Is Not An Option, Montreal, Canada.

Friedenthal, S., Moore, A., Steiner, R., 2014. A Practical Guide to SysML: The Systems Modeling Language, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. https://doi.org/10.1016/c2013-0-14457-1

Gherbi, A., Khendek, F., 2006. From UML/SPT models to schedulability analysis: a metamodel-based transformation, in: Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06). Presented at the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06), pp. 8 pp.-. https://doi.org/10.1109/ISORC.2006.37

Gomaa, H., 2000. Designing Concurrent, Distributed, and Real-Time Applications with Uml, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gómez-Martínez, E., Rodríguez, R.J., Elorza, L.E., Rezabal, M.I., Earle, C.B., 2014. Model-Based Verification of Safety Contracts, in: Canal, C., Idani, A. (Eds.), Software Engineering and Formal Methods, Lecture Notes in Computer Science. Springer International Publishing, pp. 101–115. https://doi.org/10.1007/978-3-319-15201-1_7

Grant, E., Datta, T. (Eds.), 2015. Roadmap to a DO-178C Formal Model-Based Software Engineering Methodology, in: Proceedings of the International MultiConference of Engineers and Computer Scientists 2015 Vol I, IMECS 2015, March 18 - 20, 2015, Hong Kong, International MultiConference of Engineers and Computer Scientists, IMECS 2015. IAENG, Hong Kong, p. 6.

Grunske, L., Kaiser, B., 2005. An Automated Dependability Analysis Method for COTS-Based Systems, in: Franch, X., Port, D. (Eds.), COTS-Based Software Systems, Lecture Notes in Computer Science. Presented at the International Conference on COTS-Based Software Systems, Springer, Berlin, Heidelberg, Berlin, Heidelberg, pp. 178–190. https://doi.org/10.1007/978-3-540-30587-3_28

GSFC, 1996. Performing a Failure Mode and Effects Analysis.

Harel, D., 1987. Statecharts: a visual formalism for complex systems. Sci. Comput. Program. 8, 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

Hassan, A., Goseva-Popstojanova, K., Ammar, H., 2005. UML based severity analysis methodology, in: Reliability and Maintainability Symposium, 2005. Proceedings. Annual. Presented at the Reliability and Maintainability Symposium, 2005. Proceedings. Annual, pp. 158–164. https://doi.org/10.1109/RAMS.2005.1408355

Hause, M.C., Thom, F., 2007. An Integrated Safety Strategy to Model Driven Development with SysML, in: 2007 2nd Institution of Engineering and Technology International Conference on System Safety. Presented at the 2007 2nd Institution of Engineering and Technology International Conference on System Safety, pp. 124–129.

Helle, P., 2012. Automatic SysML-based Safety Analysis, in: Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems, ACES-MB '12. ACM, New York, NY, USA, pp. 19–24. https://doi.org/10.1145/2432631.2432635

Höfig, K., 2016. Tutorial 9: Dependability Analysis in the Context of Component-Based System Architectures.

Höfig, K., Zeller, M., 2016. SpARTA - State-Aware Fault Tree Analysis.

Höfig, K., Zeller, M., Grunske, L., 2014. MetaFMEA-A framework for reusable FMEAs, in: Model-Based Safety and Assessment. Springer, pp. 110–122.
Höfig, K., Zeller, M., Heilmann, R., 2015. ALFRED: A methodology to enable component fault trees for layered architectures. Presented at the 41st Euromicro Conference on Software Engineering and Advanced Applications, IEEE, pp. 167–176. https://doi.org/10.1109/SEAA.2015.26
IBM, 2017. IBM - Rational Software Architect Designer [WWW Document]. URL https://www.ibm.com/software/products/en/ratsadesigner (accessed 10.19.17).
IEC 61508, 2005. Functional safety of electrical/electronic/programmable electronic safety-related systems [WWW Document]. URL https://webstore.iec.ch/publication/5514 (accessed 4.3.19).
ISO 26262, 2018. Road vehicles -- Functional safety [WWW Document]. ISO. URL http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/83/68383.html (accessed 4.3.19).
ISSRE, 2016. 27th International Symposium on Software Reliability Engineering, Ottawa, Canada | ISSRE 2016 [WWW Document]. URL http://2016.issre.net/ (accessed 12.7.16).
Joshi, A., Vestal, S., Binns, P., 2007. Automatic generation of static fault trees from AADL models, in: Workshop on Architecting Dependable Systems of The 37th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks, Edinburgh, UK.
Kaâniche, M., Laprie, J.-C., Blanquart, J.-P., 2002. A framework for dependability engineering of critical computing systems. Saf. Sci. 40, 731–752.
Kaiser, B., Gramlich, C., Förster, M., 2007. State/event fault trees—A safety analysis model for software-controlled systems. Reliab. Eng. Syst. Saf., SAFECOMP 2004, the 23rd International Conference on Computer Safety, Reliability and Security 92, 1521–1537. https://doi.org/10.1016/j.ress.2006.10.010
Kaiser, B., Liggesmeyer, P., Mäckel, O., 2003. A New Component Concept for Fault Trees, in: Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software, SCS '03. Australian Computer Society, Inc., Darlinghurst, Australia, pp. 37–46.
Kim, J., Ghang, S., Lee, E., 2012. Run-time fault detection using automatically generated fault tree based on UML, in: Convergence and Hybrid Information Technology. Springer, pp. 426–435.
Knight, J., 2012. Fundamentals of Dependable Computing for Software Engineers. CRC Press.
Kolovos, D., 2009. SimpleTrace.emf [WWW Document]. URL https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/examples/org.eclipse.epsilon.examples.metamodels/SimpleTrace.emf (accessed 4.7.19).
Kolovos, D., Rose, L., Paige, R., Garcıa-Domınguez, A., 2017. The epsilon book. The Eclipse Foundation.
Kolovos, D.S., Paige, R.F., Polack, F.A.C., 2008. The Epsilon Transformation Language, in: Vallecillo, A., Gray, J., Pierantonio, A. (Eds.), Theory and Practice of Model Transformations, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 46–60.

Kolovos, D.S., Paige, R.F., Polack, F.A.C., 2006. The Epsilon Object Language (EOL), in: Rensink, A., Warmer, J. (Eds.), Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 128–142.
LASER, 2017. LASER: Little-JIL [WWW Document]. URL http://laser.cs.umass.edu/tools/littlejil.shtml (accessed 10.31.17).
Malhotra, M., Trivedi, K.S., 1995. Dependability modeling using Petri-nets. IEEE Trans. Reliab. 44, 428–440. https://doi.org/10.1109/24.406578
Mani, N., Petriu, D.C., Woodside, M., 2011. Studying the Impact of Design Patterns on the Performance Analysis of Service Oriented Architecture, in: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. Presented at the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 12–19. https://doi.org/10.1109/SEAA.2011.13
Mhenni, F., Nguyen, N., Choley, J.Y., 2018. SafeSysE: A Safety Analysis Integration in Systems Engineering Approach. IEEE Syst. J. 12, 161–172. https://doi.org/10.1109/JSYST.2016.2547460
Microsoft Office Dev Center, 2018. Structure of a SpreadsheetML document (Open XML SDK) [WWW Document]. URL https://docs.microsoft.com/en-us/office/open-xml/structure-of-a-spreadsheetml-document (accessed 3.6.19).
Miguel, M.A.D., Briones, J.F., Silva, J.P., Alonso, A., 2008. Integration of safety analysis in model-driven software development. IET Softw. 2, 260–280. https://doi.org/10.1049/iet-sen:20070050
MIL-STD-1316E, 1998. Safety Criteria for Fuze Design.
Modelica Association, 2017. Modelica and the Modelica Association [WWW Document]. URL https://www.modelica.org/ (accessed 12.11.17).
Mohrle, F., Zeller, M., Hofig, K., Rothfelder, M., Liggesmeyer, P., 2015. Automated compositional safety analysis using component fault trees. Presented at the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 152–159. https://doi.org/10.1109/ISSREW.2015.7392061
Montecchi, L., Lollini, P., Bondavalli, A., 2011. An Intermediate Dependability Model for state-based dependability analysis (Technical No. RCL101115). University of Florence, Dip. Sistemi Informatica, RCL group.
Müller, M., Roth, M., Lindemann, U., 2016. The hazard analysis profile: Linking safety analysis and SysML. Presented at the Annual IEEE Systems Conference (SysCon), pp. 1–7. https://doi.org/10.1109/SYSCON.2016.7490532
Mustafiz, S., 2010. Dependability-oriented Model-driven Requirements Engineering for Reactive Systems (Ph.D. Dissertation). McGill University, Montreal, Que., Canada, Canada.
Nair, S., De La Vara, J.L., Sabetzadeh, M., Briand, L., 2014. An extended systematic literature review on provision of evidence for safety certification. Inf. Softw. Technol. 56, 689–717. https://doi.org/10.1016/j.infsof.2014.03.001
No Magic, Inc., 2017. MagicDraw [WWW Document]. URL https://www.nomagic.com/products/magicdraw (accessed 7.17.17).
Obeo, 2017. ATL: ATLAS Transformation Language [WWW Document]. URL https://eclipse.org/atl/ (accessed 7.26.17).

Oliveira, A.L. d, Braga, R.T.V., Masiero, P.C., Papadopoulos, Y., Habli, I., Kelly, T., 2014. A Model-Based Approach to Support the Automatic Safety Analysis of Multiple Product Line Products, in: 2014 Brazilian Symposium on Computing Systems Engineering (SBESC). Presented at the 2014 Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 7–12. https://doi.org/10.1109/SBESC.2014.20

OMG, 2018. What is SysML? | OMG SysML [WWW Document]. URL http://www.omgsysml.org/what-is-sysml.htm (accessed 4.30.18).

OMG, 2015a. OMG Systems Modeling Language (OMG SysML™) V1.4 (Standard).

OMG, 2015b. OMG Unified Modeling Language (OMG UML), v2.5 (Standard).

OMG, 2014. Model Driven Architecture (MDA) Guide revision 2.0.

OMG, 2011. UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) (Standard).

Pai, G.J., Dugan, J.B., 2002. Automatic synthesis of dynamic fault trees from UML system models, in: 13th International Symposium on Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. Presented at the 13th International Symposium on Software Reliability Engineering, 2002. ISSRE 2003. Proceedings, pp. 243–254. https://doi.org/10.1109/ISSRE.2002.1173261

Papadopoulos, Y., Walker, M., Parker, D., Rüde, E., Hamann, R., Uhlig, A., Grätz, U., Lien, R., 2011. Engineering failure analysis and design optimisation with HiP-HOPS. Eng. Fail. Anal. 18, 590–608. https://doi.org/10.1016/j.engfailanal.2010.09.025

Python Software Foundation, 2017. Python [WWW Document]. Python.org. URL https://www.python.org/ (accessed 10.17.17).

Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C., 2008. The Epsilon Generation Language, in: Schieferdecker, I., Hartman, A. (Eds.), Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–16.

RTCA, 2011. Software considerations in airborne systems and equipment certification. RTCA, Washington D.C.

RTCA, 1992. Software considerations in airborne systems and equipment certification. RTCA, Inc., Washington, D.C.

Russo, S., Scippacercola, F., 2016. Model-Based Software Engineering and Certification: Some Open Issues, in: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Presented at the 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 237–240. https://doi.org/10.1109/ISSREW.2016.24

SAE-ARP4754, 1996. Certification Considerations for Highly-Integrated Or Complex Aircraft Systems (Aerospace Standard) - SAE Mobilus [WWW Document]. URL https://saemobilus.sae.org/content/arp4754 (accessed 4.3.19).

SAE-ARP4761, 1996. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. USA Eng. Soc. Adv. Mobil. Land Sea Air Space. https://doi.org/10.4271/ARP4761

Sherer, S.A., 1988. Methodology for the assessment of software risk (Ph.D. Dissertation). University of Pennsylvania.

Stamatelatos, M., Vesely, W., Dugan, J., Fragola, J., Minarick, J., Railsback, J., 2002. Fault Tree Handbook with Aerospace Applications. NASA Wash. DC 218.
The Eclipse Foundation, 2017a. Papyrus [WWW Document]. URL https://eclipse.org/papyrus/ (accessed 7.15.17).
The Eclipse Foundation, 2017b. Eclipse [WWW Document]. URL https://www.eclipse.org/home/index.php (accessed 10.17.17).
The Eclipse Foundation, 2017c. Emfatic [WWW Document]. URL https://www.eclipse.org/emfatic/ (accessed 7.17.17).
Towhidnejad, M., Wallace, D.R., Gallo, Jr., A.M., 2003. Validation of object oriented software design with fault tree analysis, in: Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard. Presented at the Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard, pp. 209–215. https://doi.org/10.1109/SEW.2003.1270745
Wang, D., Pan, J., Avrunin, G.S., Clarke, L.A., Chen, B., 2010. An Automatic Failure Mode and Effect Analysis Technique for Processes Defined in the Little-JIL Process Definition Language., in: SEKE. Presented at the The 22nd International Conference on Software Engineering and Knowledge Engineering, San Francisco Bay, USA, pp. 765–770.
Wu, J., Yue, T., Ali, S., Zhang, H., 2015. A modeling methodology to facilitate safety-oriented architecture design of industrial avionics software: SAFETY ARCHITECTURE DESIGN OF AVIONICS SOFTWARE. Softw. Pract. Exp. 45, 893–924. https://doi.org/10.1002/spe.2281
Xiang, J., Yanoo, K., Maeno, Y., Tadano, K., 2011. Automatic Synthesis of Static Fault Trees from System Models, in: 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement. Presented at the 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement, pp. 127–136. https://doi.org/10.1109/SSIRI.2011.32
Xiao, N., Huang, H.-Z., Li, Y., He, L., Jin, T., 2011. Multiple failure modes analysis and weighted risk priority number evaluation in FMEA. Eng. Fail. Anal. 18, 1162–1170. https://doi.org/10.1016/j.engfailanal.2011.02.004
Yakymets, N., Perin, M., Lanusse, A., 2015. Model-driven multi-level safety analysis of critical systems, in: Systems Conference (SysCon), 2015 9th Annual IEEE International. Presented at the Systems Conference (SysCon), 2015 9th Annual IEEE International, pp. 570–577. https://doi.org/10.1109/SYSCON.2015.7116812
Zeller, M., Ratiu, D., Höfig, K., 2016. Towards the Adoption of Model-Based Engineering for the Development of Safety-Critical Systems in Industrial Practice, in: Computer Safety, Reliability, and Security. Presented at the International Conference on Computer Safety, Reliability, and Security, Springer, Cham, pp. 322–333. https://doi.org/DOI: 10.1007/978-3-319-45480-1_26
Zhao, Z., 2014. UML Model to Fault Tree Model Transformation for Dependability Analysis (MASc Thesis). Carleton University.
Ziani, A., Hamid, B., Bruel, J., 2012. A Model-Driven Engineering Framework for Fault Tolerance in Dependable Embedded Systems Design, in: 2012 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA). Presented at the 2012 38th EUROMICRO Conference on Software

Engineering and Advanced Applications (SEAA), pp. 166–169. https://doi.org/10.1109/SEAA.2012.47
Zornoza Moreno, E., 2018. Model-based approach for automatic generation of IEC-61025 standard compliant fault trees (MASc Thesis).
Zoughbi, G., Briand, L., Labiche, Y., 2011. Modeling safety and airworthiness (RTCA DO-178B) information: conceptual model and UML profile. Softw. Syst. Model. 10, 337–367. https://doi.org/10.1007/s10270-010-0164-x