UML Model to Fault Tree Model Transformation for Dependability Analysis

by
Zhao Zhao

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

Master of Applied Science
in Electrical and Computer Engineering

Department of System and Computer Engineering
Carleton University
Ottawa, ON, Canada

August 2014

ACKNOWLEDGEMENT

I would like to express my thank to all those who helped me during the writing of this thesis.

My deepest gratitude goes first and foremost to my thesis supervisor, Prof. Dorina Petriu. It was only through her valuable supervision and patient guidance that I was able to complete all the work of this thesis.

I would also like to thank my family, my father and my mother, and all my friends for their continuous support and encouragement.

ABSTRACT

Model-Driven Development (MDD) is a software development paradigm that has shifted the focus of software development from code to models. MDD's emphasis on models facilitates also the derivation of analysis models for different software non-functional properties (NFPs) from the software design models. Such analysis models are based on different formalisms, and can be used to verify the NFPs of the software under development starting in the early lifecycle stages.

This thesis proposes a model transformation to automatically generate Fault Tree models from UML models annotated with dependability annotations. Fault tree analysis is a top down deductive failure analysis model using both qualitative and quantitative analysis of undesired events of a system. It is used in safety and reliability engineering.

The main purpose of this work is to use a specialized model transformation language to transform UML Sequence Diagrams, along with UseCase Diagrams and Composite Structure Diagrams (extended with MARTE/DAM stereotypes) into Fault Tree Models. The transformation language used in this study is ATL (ATL Transformation Language), which is a hybrid language containing both declarative expressions and imperative blocks. This work focuses on the component level of a system. The transformation covers both hardware software, as well as their allocation within the system. With the input of structural and behavioural UML diagrams along with annotated data information, the output fault tree model could be directly used for qualitative and quantitative dependability analysis by using existing fault tree analysis tools.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE FRAGMENTS

ACRONYMS

ATL	Atlas Transformation Language
DAM	Dependability Analysis Modeling profile
ECORE	EMF Core meta-metamodel
EMF	Eclipse Modeling Framework
GQAM	Generic Quantitative Analysis Modeling
IDE	Integrated Development Environment
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MDA	Model Driven Architecture
MDD	Model Driven Development
MOF	Meta-Object Facility
NFP	Non-Functional Property
OCL	Object Constraint LanguageOMG Object Management Group
PAM	Performance Analysis Modeling
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QVT	Query/View/Transformations
RTES	Real-Time Embedded Systems
SPT	Schedulability, Performance and Time UML Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Document

CHAPTER 1. INTRODUCTION

1.1 Motivation and Objectives

Model Driven Development (MDD) is a software development paradigm in which the primary focus and the products of the development are models rather than code (i.e., computer programs). An important characteristic of MDD is that the code is automatically generated from the models. The advantage of MDD is that models are expressed using concepts that are much closer to the problem domain than to the underlying implementation technology, making the models easier to specify, understand, and maintain [SELIC2003].

By changing the focus of software development from code to models, MDD enable developers to verify the non-functional properties (NFP) of software (such as performance, availability, reliability, security, etc.) by transforming UML design specifications annotated with extra information to appropriate analysis models. Many modeling techniques as well as tools have been developed for the analysis of software's different non-functional properties (e.g., Markov chains, queueing networks, Petri nets, fault trees, etc). The research challenge is to bridge the gap between model-based development tools and different existing analysis tools by using model transformation techniques. The general processing for model-based analysis is shown in the Figure 1.1 below. The non-functional property of interest in this thesis is software dependability. The thesis addresses the problem of automatic derivation of Fault Tree models from a UML software model with dependability annotations, as discussed in the following sections.

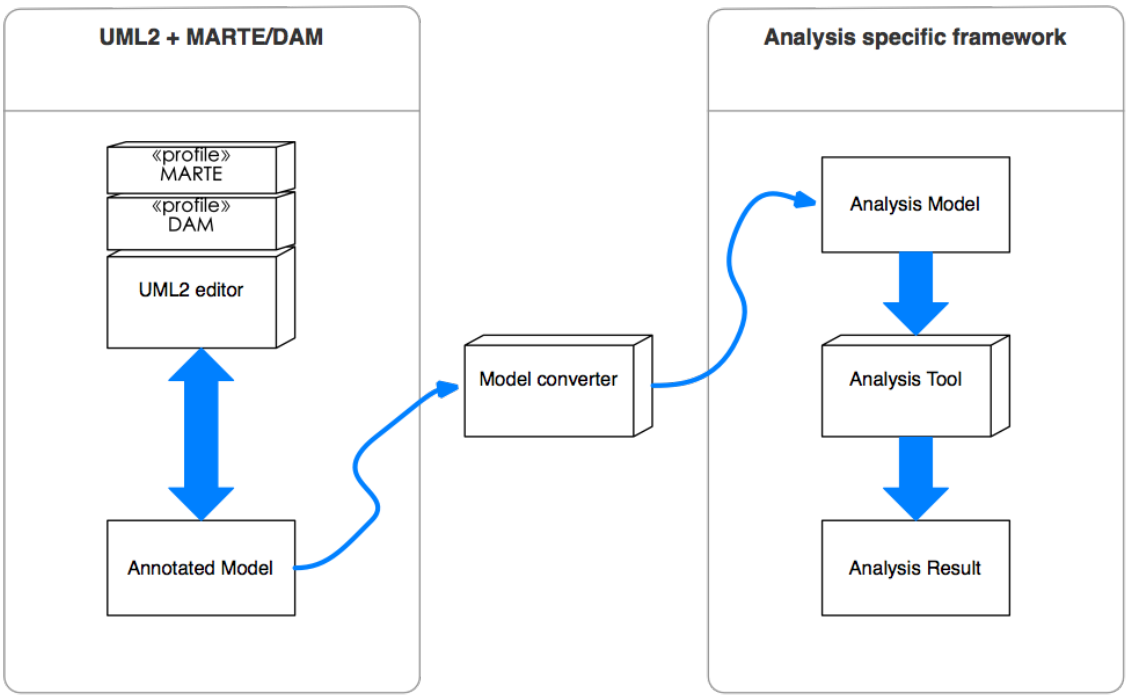


Figure 1.1 Processing schema for model-based analysis

Most approaches to model transformations have been based on general purpose programming languages, Java for instance, but more recently specialized model transformation languages have also been developed. In 2002, the Object Management Group (OMG) – the body that standardized UML - issued a Request for Proposal (RFP) on MOF Query/View/ Transformation to develop a standard transformation language compatible with other OMG standards (UML, MOF, OCL, etc.) Several replies to the RFP were given by a number of companies and research institutions; the replies have evolved into a MOF 2.0 QVT standard adopted by OMG for the specification of model transformations [MOF2011] . One of the model transformation languages proposed during this process was the Atlas Transformation Language (ATL) [ATL] developed by the ATLAS Group. ATL was released earlier than MOF QVT and has acquired a considerable group of users who helped apply ATL and evolve its toolset. This is the reason we have selected ATL for this thesis rather than MOF QVT.

ATL offers semantics that allow the specification of transformation rules. ATL transformations are much more easily maintained than those expressed in general-purpose programming languages, because rules are expressed at a higher level of abstraction, and thus can be added, removed or modified without affecting detailed implementation aspects. Such aspects are managed by the ATL engine, as opposed to model transformations developed in Java, which must explicitly address implementation details within their code. ATL also benefits from having a structure that can be understood much more easily among developers within a team, as opposed to Java where the transformation logic is more difficult to retrieve since it is deeply embedded within the code. Perhaps the most significant advantage of ATL is that it opens up the possibility of using formal proofs to verify transformation correctness through ATL's rule traceability capability. Formal proofs are not feasible with transformations written in general purpose programming languages.

The ATL language requires the definition and maintenance of source and target metamodels, to which the transformation model is tightly coupled. These challenges have given rise to research in developing design methodologies targeted to ATL and other rule-based transformation languages.

Before deriving fault tree models, we need to annotate UML design specifications with quantitative dependability attributes. We are using two existing UML profiles for this purpose: “The UML Profile for Modeling and Analysis of Real-Time Systems”(MARTE) [MARTE2011] and “The Dependability Analysis Model” (DAM) [BERNARDI2011], [BERNARDI2013].

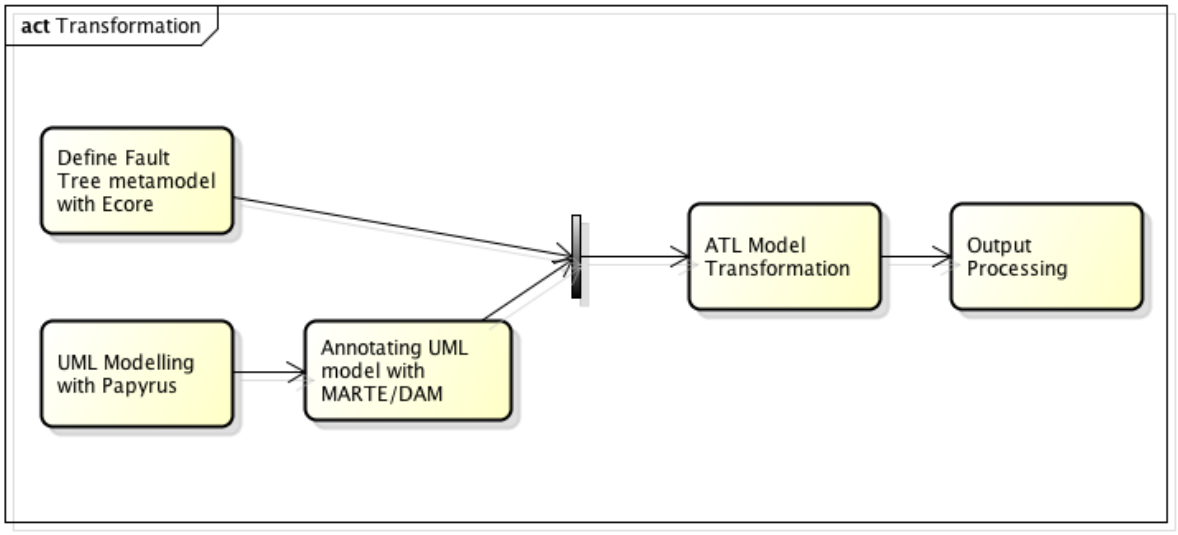


Figure 1.2 Activity of UML2FT transformation

Figure 1.1 shows the transformation process of UML+MARTE/DAM models to Fault Tree models. The main modeling tool used for producing UML Diagrams is Papyrus [PAPYRUS2011], which is an open-source tool based on EMF [EMF2013] and supports the MARTE profile. Additionally, a DAM profile application plugin [BERNARDI2011], [BERNARDI2013] that enable the use of DAM stereotypes in the Papyrus modeling environment is being added. The annotated UML model conforms to UML 2.4 Metamodel, which is an EMF registered package and can be directly utilized through EMF. The Fault Tree Metamodel, on the other hand, is created by using Ecore tools. The ATL transformation (presented in chapter 3) does its mapping from annotated UML model to the target Fault Tree model. The output Fault Tree model is then processed by a simple JAVA program to make it readable by the fault tree tool used in the thesis, FaultCAT, (which is introduced in section 2.1.3).

1.2 Thesis Contributions

The goal of this thesis is to define, implement and test an ATL transformation accepting as input a source model composed of UML sequence diagrams, UseCase diagrams and composite structure diagrams (with MARTE/DAM applied stereotypes) and generating as output a Fault Tree model. The steps for achieving this goal have resulted in the following contributions:

1) Transformation Mapping – The design of mapping rules of UML+ MARTE/DAM model elements to Fault Tree elements through the analysis of the problem domain.

2) ATL Transformation Implementation – The transformation mappings and transformation model of this study are implemented in an ATL project, which showcases the methodology explored in this thesis and forms the basis for future work. The test cases used to verify the correctness of the implementation are also described.

1.3 Thesis Contents

This thesis consists of 6 chapters, including this introductory chapter, and an appendix. Chapter 2 introduces the background material required for this work, which includes a basic description of Model-Driven Architecture (MDA), general knowledge of dependability analysis as well as some typical dependability modes, the introduction to model transformation, plus the modeling tool and transformation languages we used in this work. Also, some related works from literature are discussed and compared.

Chapter 3 described the design of the transformation from annotated UML to Fault Trees. It contains a detailed description of both metamodels of the source and target models,

and the main mappings from UML+MARTE/DAM models to Fault Tree models. The mapping rules of the transformation are presented separately.

Chapter 4 is a detailed description of the ATL implementation of UML+MARTE/DAM models to Fault Tree models, which includes the transformation model and process, the ATL run configurations and the output process method.

Chapter 5 presents several test cases and two complete case studies in which all the main aspects of this model transformation was covered and tested.

The 6th chapter is the conclusion which presents the main accomplishments of our work along with the limitations and some possible areas for future work.

CHAPTER 2. BACKGROUND AND STATE OF THE ART

2.1 Background Review

2.1.1 Model-Driven Architecture Overview

2.1.1.1. Introduction to Model Driven Architecture

Model-driven architecture (MDA) is OMG's vision for the model-driven development of software systems by using OMG standards (). MDA was first released by the Object Management Group (OMG) in 2001 [MDA2001]. A more detailed definition of MDA, presented in [MDA2003] was adopted by OMG in 2003. Basically, MDA is an integrated framework that promotes and supports the concept of model-driven engineering with OMG standards such as UML, OCL, MOF, QVT. The Unified Modeling Language (UML) is amongst the most popular modeling standards adopted by OMG.

The overall goal of the MDA framework is the strict separation between the functionality of a software system and its platform-specific implementation. This separation allows for the reuse of one software system on different platforms, as for example on Java or .NET-based frameworks. To achieve this aim, different levels of abstraction are required, made available by different models. The MDA provides an approach in which systems are specified independently of the platform supporting it. It also provides an approach for specifying platforms and for transforming the platform-independent specification into a platform-specific one. The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns [MDA2003].

MDA supports the concept of Model Driven Engineering (MDE), which includes Model-Driven Development (MDD) but also extends the use of models as described below. MDE has become a major part of today's software engineering field. With MDE, models are being considered as the key artifact of the entire engineering process [MDE]. By definition, a model is regarded as an abstraction from reality, including only the relevant aspects of a system. It is therefore only natural to represent the details of a complex software system by means of a model. Models are no longer restricted to documentation and representation purposes but they can be used for a variety of important engineering tasks, such as:

a) Model Testing: Error at the source code level is not easy to locate or to eliminate. Testing at model level eases the detection of errors and provides testing facilities at a higher level of abstraction.

b) Code generation: The (semi-)automatic generation of programming code is an essential part of the model-driven approach. Instead of using models only for communication purpose, they are directly transformed into code fragments.

c) Documentation: After all, models may still be the primary source for the documentation of a software system. As the complexity of software systems is constantly rising and the integration of different technologies is becoming more important, models yield a good option for specifying such comprehensive systems. Like most technical innovations, also the model-driven approach is in need of a sound basis by means of standardized technologies.

d) Verification of non-functional properties. The software models used for development can be transformed into software analysis models expressed in different formalisms

(such a Markov chains, queueing networks, petri nets, fault tree, etc.) which can be used for the analysis of NFPs

e) Model transformation: In general, models have diverse application domains, different levels of abstraction and may be defined based on different modeling languages. The aim of a model transformation is to automatically transform one model into another type of model.

A particularly important part of model-driven engineering is the notion of model transformation. There is a specific standard language for model transformations that has been defined by OMG called QVT, which is briefly described in the next few sections.

2.1.1.2. The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is one of those most widely-used modeling languages for software development. It was specified by the Object Management Group (OMG) and it has become the standard since its first release. UML is a graphical language which may be used to model complex, object-oriented systems. The UML provides different perspectives containing standardized modeling elements which allow for a holistic specification of a system under development.

The UML provides 13 different diagram types which represent different perspectives of the system under development. A UML diagram is created by some UML modeling tool. A brief summary of the most important diagram types is given below. A differentiation between structure diagrams and behavior diagrams is drawn at the highest level.

A structure diagram is needed to capture the structure of a system. For example, fixed system components and their respective properties may easily be modelled using a UML class diagram. The structure digram type includes: Class diagram, Object diagram, Package diagram, Component diagram, Deployment diagram, and Composite structure diagram.

Behavior diagram capture the behavior of a system , subsystem or component. Modeling different states and state transitions of system components or specifying the interaction between the user and the system are only two of many modeling possibilities. The behavior diagram type includes for example UseCase diagram, StateChart, Activity diagram, Sequence diagram, Timing diagram, Communication diagram, and Interaction overview diagram. The basic description of each kind of diagram are shown in the table below [UML Diagrams]:

Class diagram	Describe the structure of a system by showing its classes, attributes, and operations, as well as the relationship among them.
Object diagram	Represents instances of classes and their relationships; Typically, an object diagram describes special cases of class diagrams or communication diagrams.
Package diagram	Show how elements of model are organized into packages, and the dependencies between packages. Different packages may be clustered and structured within a package diagram.
Component diagram	Describes the components that make up an application, or system. The components, their relationships, interaction, and their interfaces are described.

Table 2.1: Introduction of UML diagrams

Deployment diagram	Describe the architecture of a system, which including hardware and/or software nodes, and the middleware connecting them.
Composite structure diagram.	Describe the internal structures of a classifier (such as use case, class, or component), includes the interactions of a classifier to other parts of a system.
UseCase diagram	Captures all functionalities that a system offers to its users. Shows use case, actor and their interrelationship.
State Machine Diagram	Capture the internal states of certain system components including all possible state changes.
Activity diagram	Describes the workflow of activity or action. Shows the overall control and/or data flow.
Sequence diagram	Describes the sequential logic of inter object communication, shows how component roles are communicating with one another and the message order.
Communication diagram	Typically describes the structural organisation of objects which are sending or receiving messages.
Interaction Overview Diagram	A variety of activity diagrams which overviewing the system's control flow or business processes. Each activity within the diagram could represent another interaction diagram.

Table 2.1: Introduction of UML diagrams continued

The main UML 2.0 diagrams we use in our transformation are sequence diagram, UseCase diagram and Composite structure diagram. Sequence diagrams are being used to capture most of the elements that may fail during the behavior of the system, including each execution period of each component and the message or signal passing between components.

Each UseCase diagram is being mapped to one or more interactions (e.g., sequence diagrams). For models containing multiple use cases, we considered the occurrence possibility of different use cases in our calculation process, which is being detailed discussed in section 4.4.2. Composite structure diagram are being used to show the relationship of each lifeline to its represented component, along with the software component to hardware component allocation.

2.1.1.3. Metamodels

In the context of software development, a model is used to represent a software system. The model itself contains only the relevant parts of the real system and is therefore referred to as an abstraction. The four-layer metamodel hierarchy by the OMG (see Figure 2.1) illustrates the different abstraction layers. The system can be found on the lowest layer M0. Consequently, the model which represents the system is placed on the next higher level, referred as M1.

A model usually contains different model elements which together form the modeled system. These elements that compose a model are formally specified by a so-called metamodel. All model elements are instances of metamodel elements. The metamodel is placed on the M2 layer of the four-layer hierarchy and will be explained in the next paragraph. The last level M3 represents the meta-metamodel, as described below.

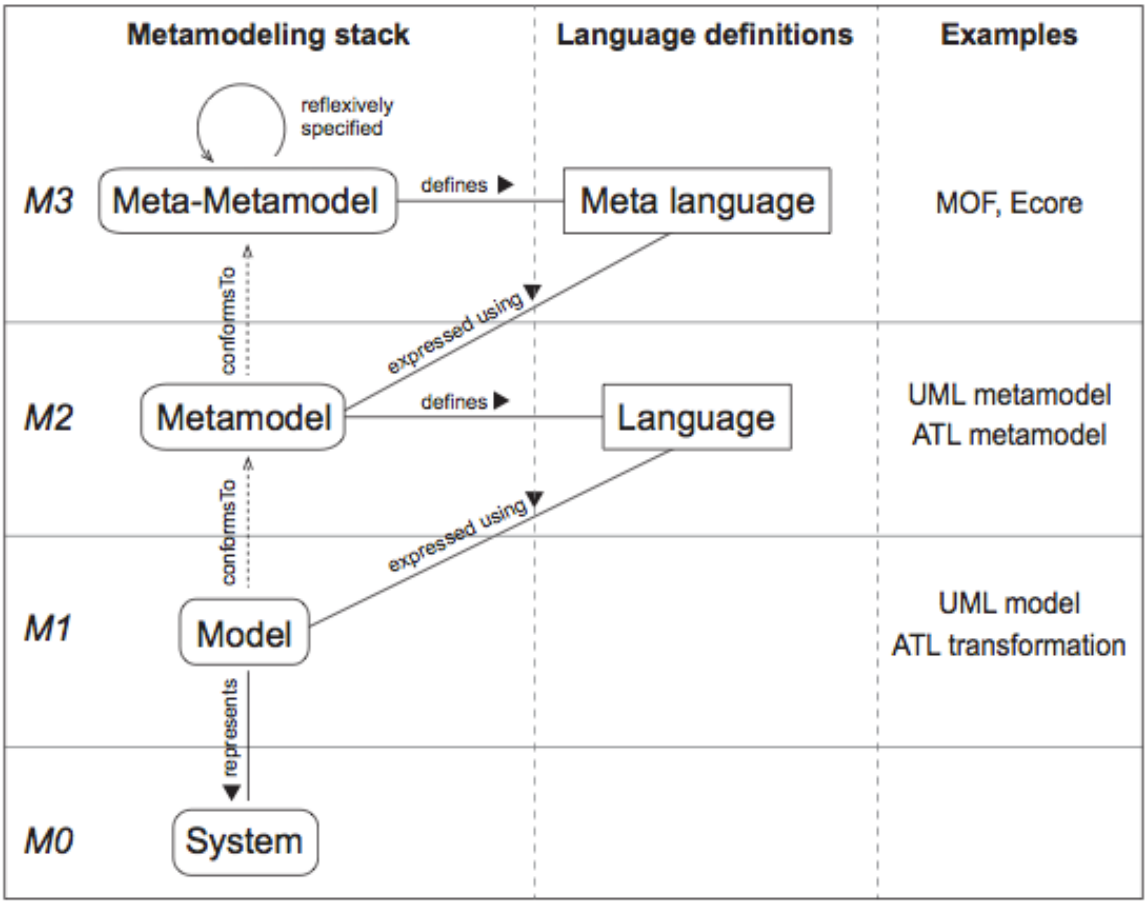


Figure 2.1: Basic structure of the four-layer metamodel hierarchy

The metamodel define the abstract syntax of the modeling language used to express a model, meaning that the metamodel comprises all language concepts and constructs that can be used in a model. It is also said that the model conforms to the metamodel. Apart from the supported concepts, the metamodel also specifies the way how these concepts are interconnected and which properties are available. Models in a given modeling language are therefore instances of the language metamodel. A model consists of model elements which are in turn instantiation of metamodel elements. Metamodels are important in the field of Model-Driven Engineering since the entire software development process is based on and guided by models. The advantages of metamodels are as follows [Metamodel]:

a) Metamodels provide a standardized and concise definition of model elements. Only those language constructs, references, and properties that are present in the metamodel can be used in the derived models. By this, a common understanding of the particular modeling domain is possible. Moreover, uncontrolled model variations can be easily prevented.

b) Metamodels are the basis for the specification of models. Therefore, models can be checked for syntactic validity as they have to conform to their respective metamodels. Metamodels are thus an efficient control mechanism which eases the task of model development.

c) A metamodel is not necessarily a rigid construct that cannot be modified at a later stage. Modeling languages like the UML offer different alternatives on how to extend or specialize a metamodel in a controlled and language-inherent manner. Nevertheless, the original metamodel should be as stable as possible in order to provide a consistent basis for all conforming models.

Models are specified by means of metamodels, so the next logical step is to examine how the concepts of metamodels are defined. As a metamodel itself is again a model, there must exist some superior level which specifies the model elements of the meta- model. This highest level is called meta-metamodel , situated on the M3 layer, on top of the M2 layer.

As shown in Figure 2.1, the meta-metamodel defines a meta language. All model elements that are used within a metamodel are specified by means of meta-metamodel elements, or in other words, every metamodel conforms to some meta-metamodel. The two most prominent meta-metamodels are briefly described below.

a) The Meta-Object Facility (MOF) is a specification by OMG and it is a standard for the definition of metamodels. MOF is particularly suitable for the definition of metamodels that represent object-oriented concepts and systems, like for example the UML. As MOF was introduced by the OMG, it is the proposed meta-metamodel within the MDA approach and the basis for all modeling concepts of the OMG.

2) Ecore is a meta-metamodel used in the Eclipse Modeling Framework [EMF2013]. The idea of Ecore is to have a Java-based implementation of the most important MOF components. Therefore, the basic language elements of Ecore and MOF are very similar.

As depicted in Figure 2.1, the four-layer hierarchy ends with layer M3. The reason for this is that a meta-metamodel, like MOF, is reflexively specified. This means that every language concept used on the M3 layer is again defined by itself. No further abstraction layer is needed which helps in keeping the hierarchy manageable.

2.1.1.4. UML Profiles

A profile in the Unified Modeling Language (UML) provides a standard extension mechanism for customising UML models for specific domains or platforms. The profile extension mechanism should allow refining the standard semantics in a stringently additive manner, avoiding contradicting the standard semantics of the language [Profile].

UML profiles allows to extend the UML metamodel for different technological platforms and modeling domains. A profile is a language-inherent extension mechanism and stereotypes as well as tagged values constitute the heart of the profile concept. Stereotypes may be seen as specialized metaclasses for either introducing new modeling concepts or re-

stricting existing ones. Tagged values play the role of stereotype attributes, being are used for introducing new features for existing modeling concepts.

A UML profile contains all stereotype definitions, tagged values and possible constraints that are needed for defining a specific extension of the reference metamodel. Each stereotype is related by an “extend” relationship to a metaclass from the reference metamodel or inherits from another stereotype. A UML profile is a package indicated by the keyword «profile», which is grouping a number of related stereotype definitions, as well as their attributes and constraints.

The Profiles package in the meta-metamodel of the UML contains all required metaclasses that are needed for the definition of a profile. The main class Profile inherits from the class Package. The relationship between a Profile instance and its contained Stereotypes is established via a containment relation named ownedStereotype. The class ProfileApplication that is associated with the class Package is used to demonstrate which profiles are applied to a package. It is important to note that a profile must always extend a reference metamodel (e.g., the UML metamodel) which is in turn conforming to MOF. This dependency is illustrated by the association named metamodelReference.

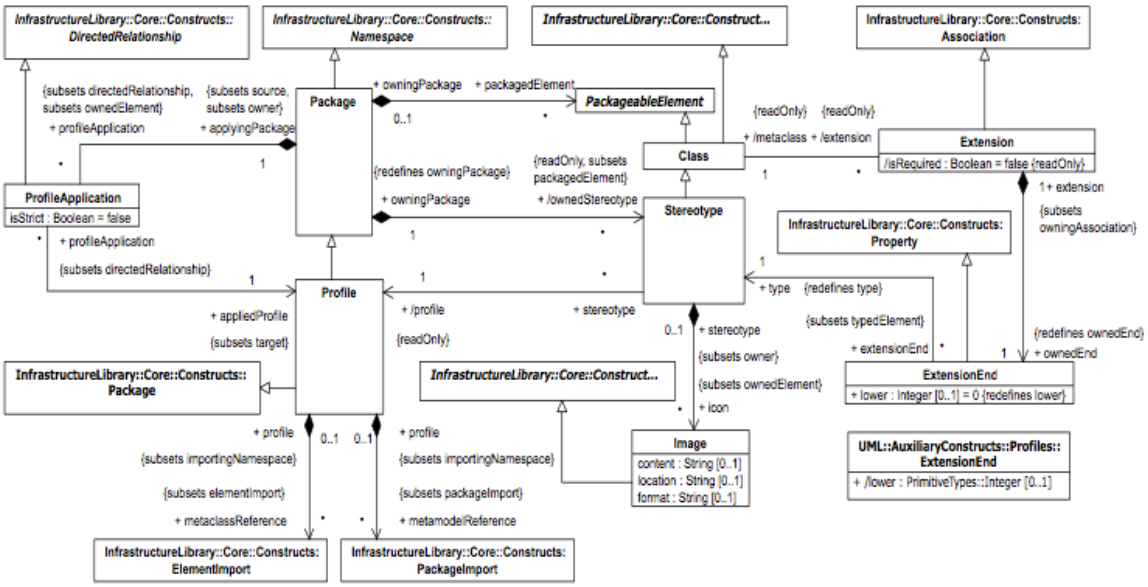


Figure 2.2: The elements defined in the Profiles package

STEREOTYPES

Stereotypes are the main component of a UML profile. The concept of stereotype is used to specialize metaclasses with respect to some domain, platform or some specific functionality. By applying a stereotype to some metaclass, the class gets bound to a defined purpose or a usage context. Moreover, the existing metaclasses may be extended by new meta attributes or restricted by user- defined constraints. Apart from the Stereotype metaclass itself, all existing UML metaclasses may be specialized by some stereotypes. The metaclasses are extended on the M2 layer (meta-model layer) of the four-layer architecture, as Figure 2.2. Stereotypes may be instantiated and used like normal model elements of the M1 model layer.

Appearance and declaration of a stereotype: a stereotype is depicted as a rectangle (like classes in a class diagram) and is composed of three sections. The first section includes the keyword «stereotype» in a pair of guillemets, followed by the name of the stereotype. The second segment contains the attributes of a stereotype, called tag definitions. An instantiated

tag definition is called tagged value. The third section may be used to define possible operations of a stereotype.

The metaclass Stereotype inherits from the metaclass Class, see Figure 2.2. By this, stereotypes can form a generalization hierarchy with abstract super-stereotypes and associated sub-stereotypes. A derived stereotype inherits all extension relations, tag definitions and constraints from its super-stereotype.

MARTE PROFILE:

After adopting the UML standard and its new release of UML 2, the UML modeling language has been widely used in the developing many of the time or resource critical systems [MARTE]. However, while a useful tool, a consensus emerged that UML lacked a few key features necessary for the development of real-time and embedded systems (RTES). For instance, UML is lacking a quantifiable notion of time and resources, which are needed for use in the real-time and embedded systems domain [MARTE]. Fortunately, the extensibility feature of UML allows for the definition of a new profile MARTE, without using heavyweight extensions.

MARTE is a UML Profile that supports specifications, designs, and verification/validation stages of RTES models [MARTE]. As shown in Figure 2.3, it is intended as a replacement for the UML SPT profile, of which MARTE borrows many constructs. SPT [SPT2005], which was defined for older UML versions (1.X), is lacking mechanisms for extending or refining those constructs for more specific need, hence the formulation of a new UML Profile for UML 2.X versions was necessary [MARTE].

MARTE defines the foundation for model-based description of both the software and hardware aspects of a RTES, including the relationships between them. These concepts can then be used for modeling and performance/schedulability analysis. This is done through annotations (i.e., stereotypes, tagged values and constraints) applied to existing UML models.

The main benefits of MARTE are [MARTE]:

a) Provide an ordinary way for modeling both hardware and software aspects of a RTES, which could improve the communication between developers.

b) Enable the interoperability between developing tools which are being used for specification, design, verification, code generation, etc.

c) Support the construction of models that could be used to make quantitative predictions related to real-time or embedded features of systems for both hardware and software characteristic. MARTE has two main concerns, one is for modeling the feature of real-time and embedded systems and the other is to annotate application models so that it could have the ability to support the analysis of non-functional properties [MARTE].

Figure 2.4 shows the architecture of the MARTE Profile. MARTE contains three main parts: *Marte Foundations*, *Marte Design Model* and *Marte Analysis Model*. The last part defines profiles for two different types of analysis: schedulability and performance analysis. The concepts common to these two quantitative analyses are contained in the *Generic Quantitative Analysis Model* (GQAM) sub-profile, which is in turn specialized by SAM and PAM. The MARTE sub-profiles important for this work are GQAM and PAM.

GQAM Resource Concepts. A resource is based on the abstract *Resource* class defined in GRM and contains common features such as scheduling discipline, multiplicity, services. Important resources in GQAM are: a) *ExecutionHost*: processor or other computing device; b) *CommunicationsHost*: communication network or bus; c) *SchedulableResource*: software process managed by the operating system; d) *CommunicationChannel*: logical channel that conveys messages.

GQAM Behaviour/Scenario Concepts. The class *BehaviorScenario* describes a behavior triggered by an event. *Scenarios* define execution paths with externally visible end points. Each scenario is composed from scenario *Steps* joined by predecessor-successor relationships, which may include fork/join, branch/merge and loops. A step may represent an elementary operation or a whole sub-scenario. A specialized step, *CommunicationStep*, defines the conveyance of a message. *Resource* usage is attached to behaviour in different ways: a) a *Step* implicitly uses a *SchedulableResource*; b) each primitive *Step* executes on a host processor; c) specialized steps, *AcquireStep* or *ReleaseStep*, explicitly acquire or release a *Resource*.

GQAM Workload Concepts. Each scenario is executed by a workload, which may be open or closed. A workload is represented by a stream of triggering events, *WorkloadEvent*, generated in one of the following ways: a) by a timed event; b) by a given arrival pattern (periodic, aperiodic, sporadic, burst, irregular, open, closed); c) by a generating mechanism named *WorkloadGenerator*; d) from a trace of events stored in a file.

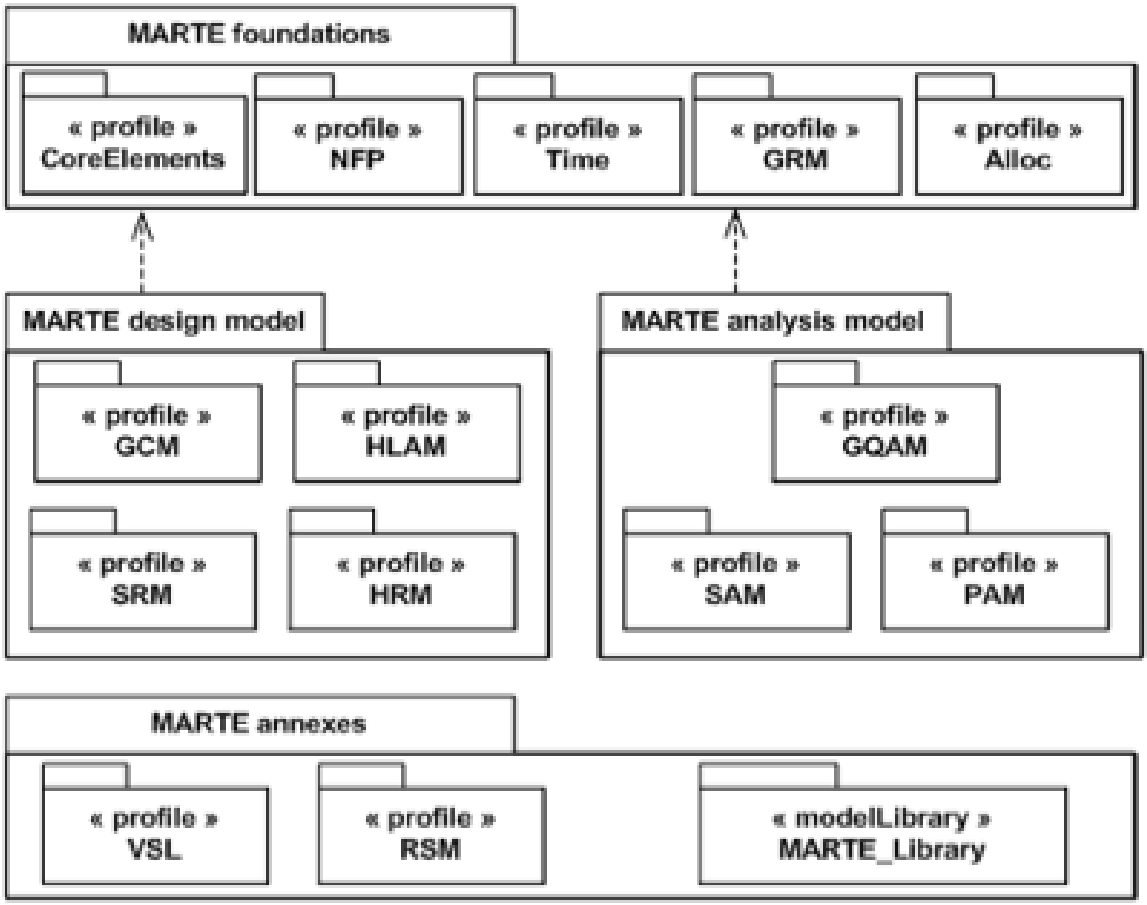


Figure 2.3: Architecture of the MARTE Profile

Performance Analysis Model (PAM) inherits from GQAM. Only a few new concepts are defined in PAM, while most of the concepts are reused from GQAM. In term of resources, PAM reuses *ExecutionHost* for processor, *Schedulable Resources* for processes and adds a *LogicalResource* for a software resource (such as semaphore, lock, buffer pool, critical section). A runtime object instance *PaRunTInstance* is an alias for a process or thread pool associated in behavior specifications to the role responsible for the actions on lifelines or swim-lanes. PAM specializes *Step* to include more kinds of operation demands during a step. Figure 2.4 shows the workload and scenarios part of the PAM sub-profile.

how system resources are utilized by the system. This includes properties of resources such as thread pools sizes or throughput.

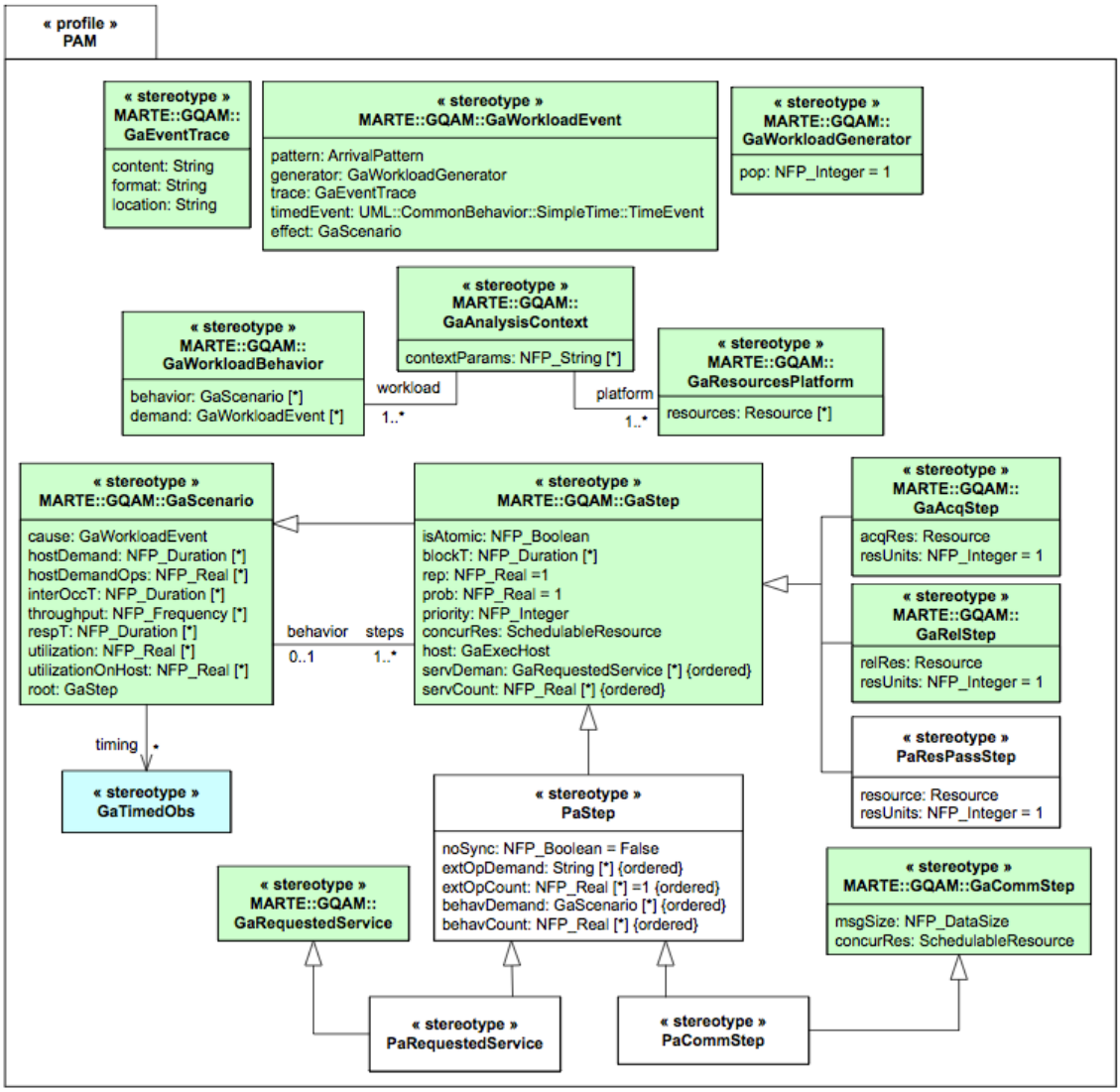


Figure 2.4: Profile diagram of performance extensions [MARTE]

Another important subprofile for this work is the Allocation Model Alloc (see Fig. 2.3). Allocating functional application elements onto their available resources is one of the main concerns of real-time embedded system designs. This combines both spatial distribution and temporal scheduling aspects, in order to map various algorithmic operations onto avail-

able computing and communication resources and services. [MARTE] As shown in Figure 2.6.

MARTE profile can be used to specify applications and execution platform models. A MARTE allocation stereotype is an association between a MARTE application and the target MARTE execution platform. Application elements could be any UML element which suitable for modeling an application, with structural and behavioral aspects. An execution platform could be a set of connected resources, which provides services to support the execution of the applications. [MARTE].

In our work, instead of using UML Deployment diagrams, we chose to use the stereotype 《Allocation》 from MARTE to show the deployment relationship from software component to hardware component. The main reason is that our modeling tool Papyrus does not support deployment diagrams so well. The most serious bug we found during our implementation process is that it does not allow users to model communication paths between nodes, so that we are not able to model a link (for example LAN or WAN) between hardware nodes.

There are some other indirect ways to deal with this problem and we chose to use the Alloc subprofile in which is available in Papyrus. Detailed use is described in section 4.4.2.

DAM PROFILE:

The Dependability Analysis and Modeling (DAM) profile enables to specify dependability requirements and properties in UML-based models [BERNARDI2011], [BERNARDI2013]. It is aimed at supporting automatic transformations of UML models to

formal models for dependability assessment purposes [DAM]. The DAM profile extends the OMG standard MARTE profiledescribed in the previous section.

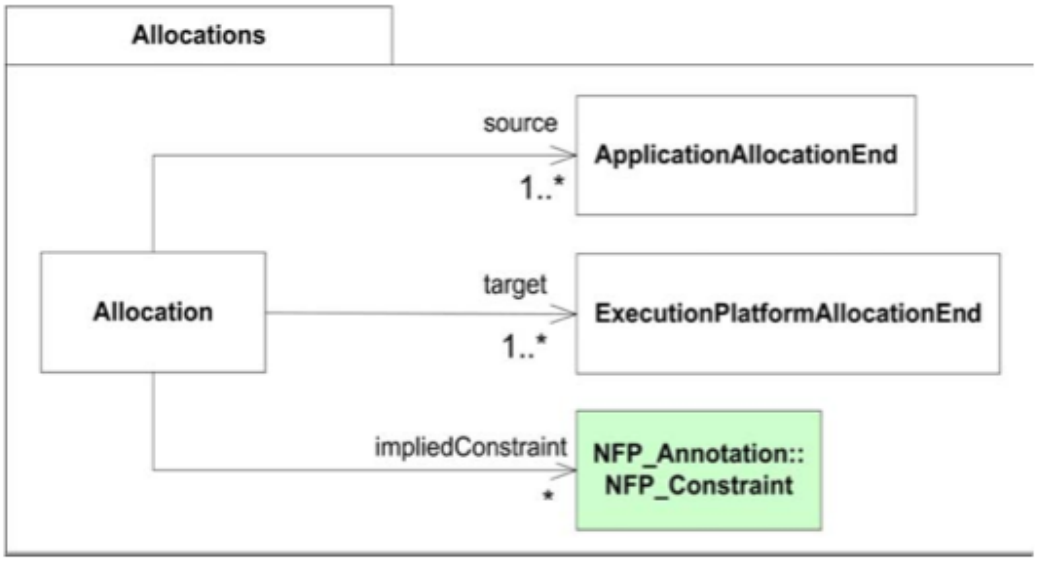


Figure 2.5: The Allocation Model of MARTE

The DAM domain model has been built for considering the main dependability concepts from the literature and the standard methods used for dependability assessment. It is being organized into several packages, as shown in Figure 2.6, in which the top level package includes:

a) The System model: provides the description of the system to be analyzed, based on the component view of the system. The model also includes the additional concepts for describing redundancy structures that may characterize a fault tolerant (FT) system.

b) The Threats model: introduces threats that may affect the system at different levels and the relationship between threats.

c) The Maintenance model: introduces repair/recovery actions assured in case of repairable system.

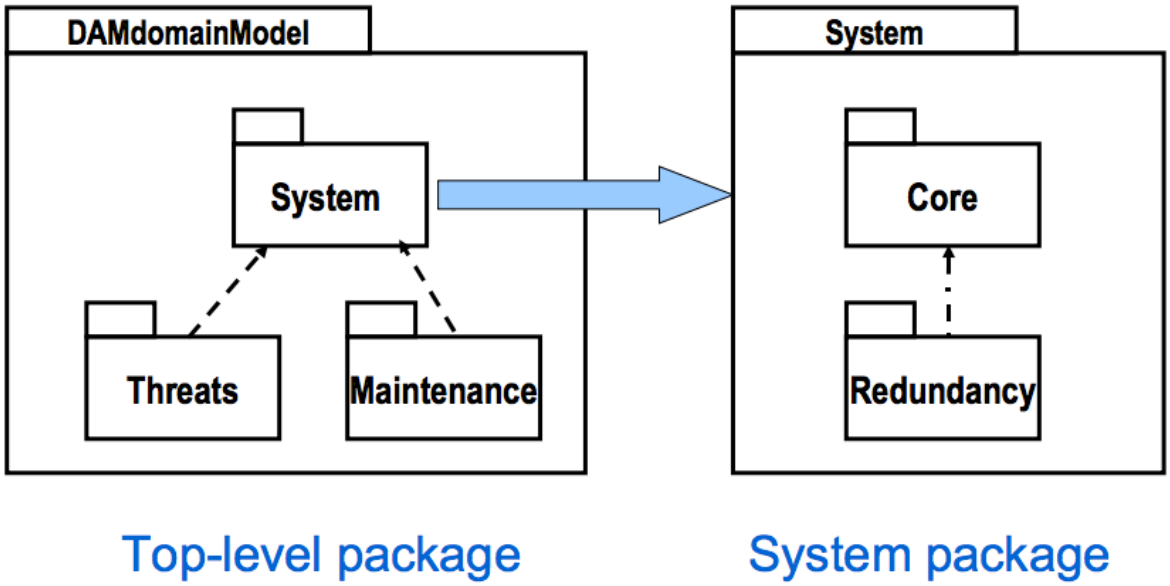


Figure 2.6: DAM Profile Domain Model Overview

The Core model (Figure 2.7) describes the context to conduct dependability analysis, which is a component-based description of the target system to be analyzed that may include both structural and behavioural aspects. From the structural view, the system contains several hardware and software components that are being connected through connectors, in order to interact with each other. The structure of the system enables it to generate the system behaviour.

The system could deliver a set of high level services, in order to respond to user service requests. Each high level service is a part of the system behaviour as noticed by its users and carried out by the interaction of components, which provide and request some basic ser-

vices to each other. A component, in this case, must either provide or request at least one basic service.

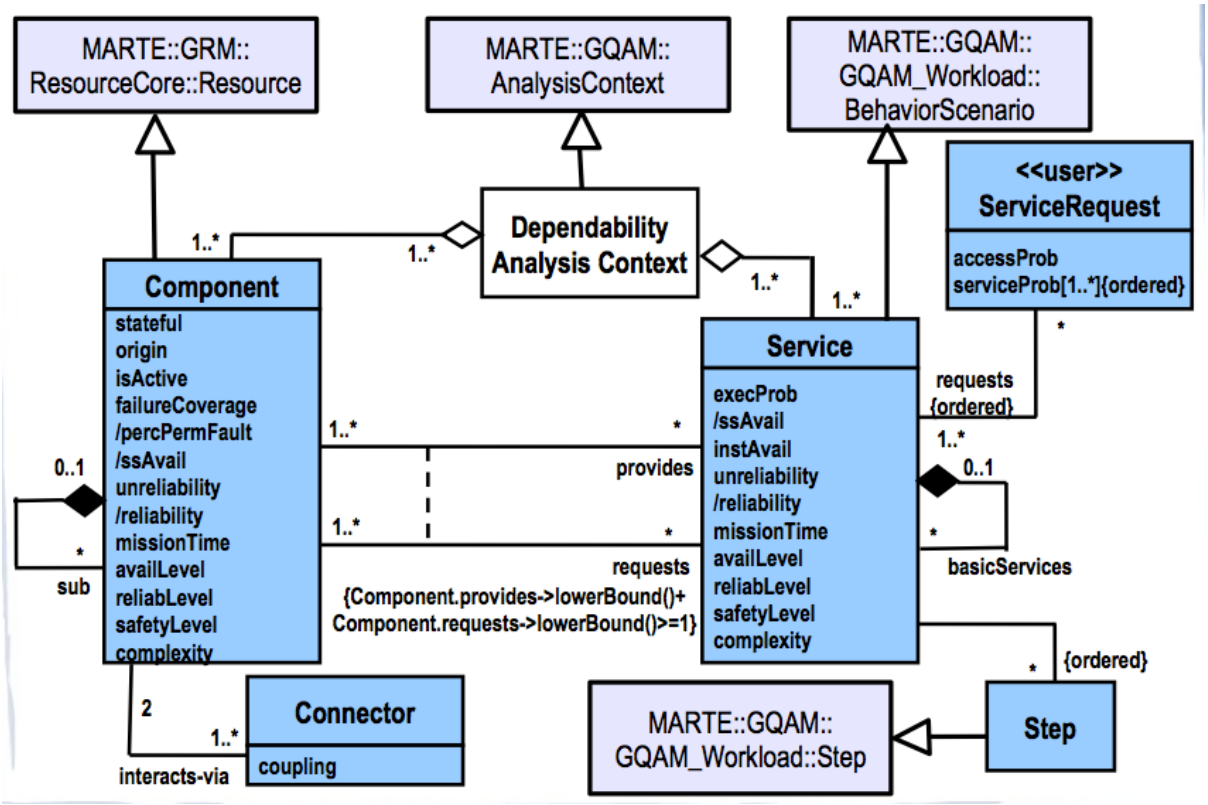


Figure 2.7: DAM Core Model

A service was implemented by a series of steps that may represent component actions and states.

The Core model could be considered as a bridge between the DAM concept and the concept introduced in MARTE profile for general quantitative analysis and modeling (GQAM). Indeed, several classes of the Core model will be mapped to stereotypes that specialize GQAM stereotypes.

In our work, the stereotype 《DaComponent》 is applied to the components in the composite structure diagram which represent lifelines in the sequence diagram), while 《Da-

Connector》 is applied to the hardware connectors between components. A more detailed description is in section 4.4.2.

A system could be characterized by its redundancy structure. Software and hardware redundancy are classic methods used to improve the fault tolerance (FT) of the system.

The Redundancy model (as shown in Figure 2.8) represents fault tolerance components [DAM], which may play different roles in redundant structures [DAM]. In particular, a redundant structure may include some variations, such as different modules may provide the same services, allocated to different spares.As we can see from Figure 2.8, a controller is in charge of coordinating the variations, an adjudicator seeks for an agreement of two or more outputs among the variants.

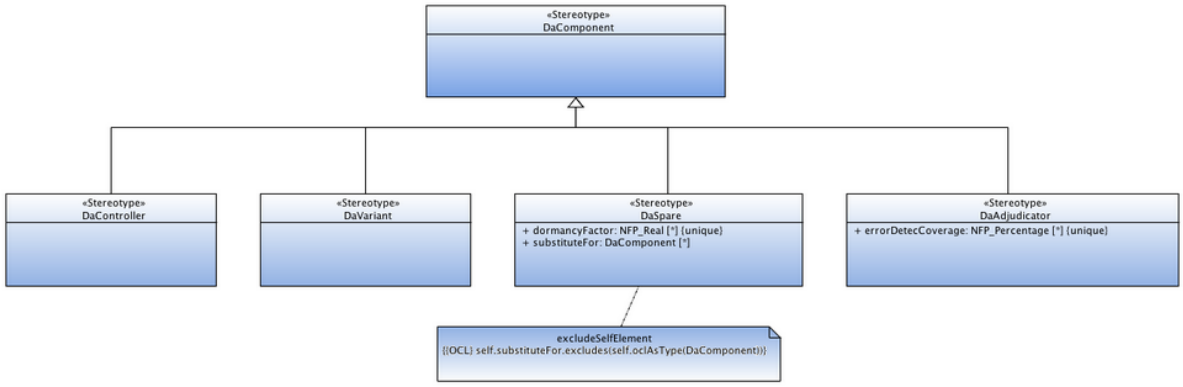


Figure 2.8 DAM Redundancy Model

In our work, the stereotype 《DaSpare》 is being applied to the spare components while transforming redundant elements. A more detailed description is in section 4.4.3.

2.1.2 Dependability Analysis

In system engineering, dependability is a composite measure of a system's availability, reliability, and its maintainability [DEPEND]. This may also enclose mechanisms for increasing and maintaining the dependability of a system.

Attributes are qualities of a system. They could be evaluated to define its general dependability using either qualitative or quantitative measurement. Avizienis et al. define the following Dependability Attributes:

a) Availability - readiness for correct service

b) Reliability - continuity of correct service

c) Safety - absence of catastrophic consequences on the user(s) and the environment

d) Integrity - absence of improper system alteration

e) Maintainability - ability for a process to undergo modifications and repairs

According to the above definition, Availability and Reliability are two attributes which could be quantitatively measured while the other three are more subjective. For example Safety , may not be directly measured via metrics, but it is more like a subjective evaluation that requires judgmental information to be applied in order to define a confidence level, while Reliability can be directly quantitatively measured as failures over time.

Threats are those things that can influence a system and may cause a decrease in Dependability. There are 3 main concepts that must be clearly defined [DEPEND]:

a) Fault: A fault is a defect in a system. The existence of a fault in the system may or may not lead to a failure. For example, although a system may have a fault, its state conditions or inputs may never lead this fault to be executed; and in this case that specific fault never shows as a failure.

b) Error: An error is a divergence between the intended behaviour of the system and its practical behaviour. Errors occur during the runtime when part of the system enters an unexpected state caused by the activation of a fault. Because errors are caused by invalid states, they are harder to detect without particular mechanisms, such as debuggers or monitors.

c) Failure: A failure is an instance in time when a system displays behaviour that is contrary to its specification. An error may not necessarily cause a failure, for instance an exception may be thrown by a system but this may be caught and handled using fault tolerance techniques, so the overall operation of the system will conform to the specification.

It is important to note that Failures are recorded at the system boundary. They are basically Errors that have propagated to the system boundary and have become observable. Faults, Errors and Failures operate according to a mechanism known as a Fault-Error-Failure chain. As a general rule a fault, when activated, can lead to an error (which is an invalid state) and the invalid state generated by an error may lead to another error or a failure (which is an observable deviation from the specified behaviour at the system boundary).

Once a fault is activated, an error is created. An error may act in the same way as a fault in that it can create further error conditions, therefore an error may propagate multiple times within a system boundary without causing an observable failure. If an error propagates

outside the system boundary a failure is said to occur. A failure is basically the point at which it can be said that a service is failing to meet its specification. Since the output data from one service may be fed into another, a failure in one service may propagate into another service as a fault so a chain can be formed of the form: Fault leading to Error leading to Failure leading to Error, etc.

2.1.3 Dependability Model

2.1.3.1. Overview

Dependability is a non-functional property (NFP) of a system [DEPEND], defined as the ability to avoid failures that are more frequent and severe than acceptable. Dependability encompasses a set of attributes: reliability, availability, maintainability, integrity and safety. The assessment of these attributes may imply quantitative and/or qualitative evaluation of the system, which has been a research topic since the early times of computing. The use of models for this purpose is extensively recognized. As already mentioned, a model is an abstraction of the system for the purpose of understanding it before building it. A software system model describes a specific system view; in a broad sense we can distinguish behavioural and structural software views, which together constitute the model of the system. A dependability model considers the abstractions needed to represent the failures of the system and their consequences. This implies that in some manner the dependability model needs to be related to the behavioural model of the system or at least to its abnormal behaviours. Models are developed using different kinds of languages and/or notations, some of them with an underlying mathematical formalism supporting some kind of analysis (e.g., fault trees, Markov chains, Petri nets or Bayesian networks). These are called formal models, analyzable models or mod-

els for analysis. The task of developing models is known as modeling, while the task of analyzing quantitative or qualitative properties is known as analysis.

2.1.3.2. Petri Net

Specifying systems at the state space level can be an error-prone, low level activity. Petri nets have been widely recognized in the literature as an effective way to specify systems using a reasonably high-level formalism, while at the same time having a precise operational semantics that allows the derivation of the associated state-space. In particular the class of Stochastic Petri Nets [PETRI](SPN) has a semantics defined through Markov chains, so they are considered a natural language to use when the stochastic process underlying the system is a Markov chain. Indeed SPN have been widely used not only for the study of generic performance indices, but also specifically for dependability studies, as testified by the available SPN tools that allow the computation of some pre-defined dependability quantities. Petri-net based models have been extensively used for performance and performability modeling to analyze computer and communication systems.

A Petri net [PETRI] is a directed bipartite graph with 2 disjoint sets of nodes: places and transitions. In a graphical representation of a Petri net, places are represented by circles, and transitions are represented by bars. In a Petri net, there are a finite number of places and transitions (see Fig. 2.9 for an example). Nodes are connected by directed edges. A place is an input to a transition if there is an edge from the place to the transition. That edge is an input arc. A place is an output of a transition if there is an edge from the transition to the place. That edge is an output arc. An integral multiplicity can be associated with each input and output arc.

A Petri net is marked if tokens are associated with the places. The dynamic behavior of the system is determined by the movement of tokens. The tokens move based upon t h e firing of transitions. A transition is enabled to fire if the number of tokens in each of its input places is at least equal to the multiplicity of the corresponding input arc from that place.

When a transition fires, tokens are removed from each of its input places and deposited in each of its output places. The number of tokens removed from each of the input places of a firing transition is equal to the multiplicity of the corresponding input arc; the number of tokens deposited in its output places is equal to the multiplicity of the corresponding output arc.

At any instant of time, more than one transition can be enabled but only one transition is allowed to fire. In a graphical representation of a Petri net, tokens are denoted by small dots or integers within a place. Multiplicity of arcs is denoted by putting a backslash on the arc and placing a positive integer with it. [MALHOTRA95]

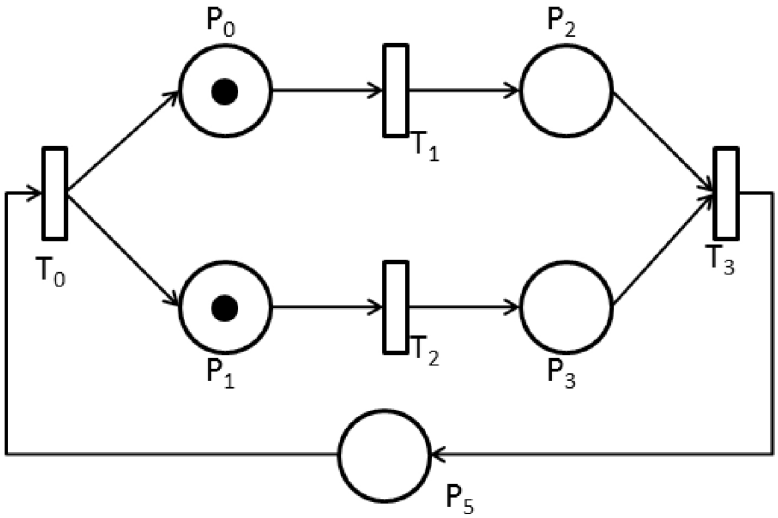


Figure 2.9: Example of Petri Net

2.1.3.3. Fault Tree

Fault Tree analysis (FTA) is a top down, deductive failure analysis method in which an undesired state of a system is analyzed using Boolean logic to combine a series of lower level events. This analysis method is mainly used in the fields of safety engineering and reliability engineering to analyze how a systems can fail, or to find out the best ways to reduce risk or to determine event rates of accidents or system level failures. It is especially used in the nuclear power, aerospace, and other high-hazard industries.

The methodological approach to dependability based on FTA consists of the following steps: a) definition of the Top Event, 2) construction of the Fault Tree, c) qualitative analysis and d) quantitative analysis.

a) *Definition of the Top Event (TE)*. Top Event candidates are events whose existence may lead to unsafe conditions, catastrophic failure or malfunction [FaultTree], unaccomplishment of the assigned mission, and so on. If more Top Events were needed to be investigated, a different tree for each one must be built and analyzed.

b) *Construction of the fault-tree*. Once the TE has been defined, the construction of the FT proceeds by identifying the immediate causes for the occurrence of the TE, and those logical relationships.

The immediate, necessary and sufficient causes for the TE constitute the first level of the tree. Each immediate cause is now treated as a sub-top event, and the analysis proceeds to determine their immediate causes. In this way, the construction evolves iteratively from

events to their causes, continuously approaching finer resolution, until a desired level of detail is reached.

Interactions between causes at each level of the iterative construction are represented by means of logic gates (usually OR and AND gates, but more complex gates can be defined, as, for example, k out of n gates), while the output of the logical gates represents the occurrence of the higher level of the tree. The events at which the construction of the tree is ended are called terminal events.

c) *Qualitative Analysis of a FT*. The qualitative analysis is aimed at identifying all the combinations of events that cause the top event to occur, as a function of the terminal events.

Combinations are ranked according to the number of events, since the smaller the number of events that cause the TE, the less resilient to failure the system is likely to be. The qualitative analysis of an FT consists in deriving a logical expression describing the TE, in such a way that all the combinations of events whose simultaneous occurrence provokes the TE are evidenced.

A combination of events whose simultaneous occurrence provokes the TE is called a *cut set* (CS) for the system. A CS that does not contain any subset which is again a CS, is minimal and is called a *minimal cut set* (MCS) or *mincut*.

d) *Quantitative analysis of a FT*. The computation of the probability of occurrence of the TE and of the MCS is, usually, the main concern of FTA. However, several other useful quantitative measures can be defined and evaluated, like the expected number of failed components, the main failure equivalence, and the Mean Time To Failure. FTA is a widespread

practice for the availability analysis of systems, and there are a number of tools that support it.

GRAPHIC SYMBOLS:

The basic symbols used in FTA are categorized as events, gates, and transfer symbols, as shown in figure 2.10. Minor variations may be used in different FTA tools.

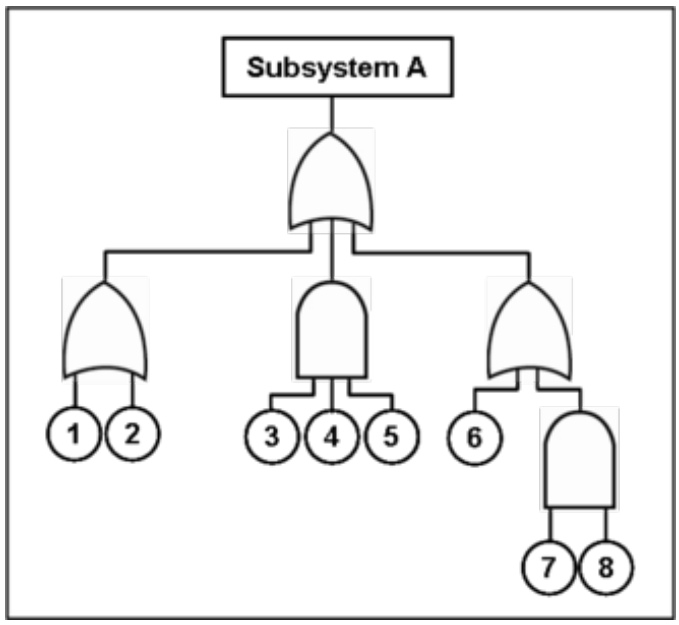


Figure 2.10: Example of Fault Tree Model

EVENT SYMBOLS:

Event symbols are used for primary events and intermediate events. Primary events are not further developed in the fault tree (they are leaves in the tree). Intermediate events are found at the output of a gate. The events symbols are shown in Fig. 2.11.

The types of events in Fig. 2.11 are as follows:

1) Basic event - failure or error in a system component or element.

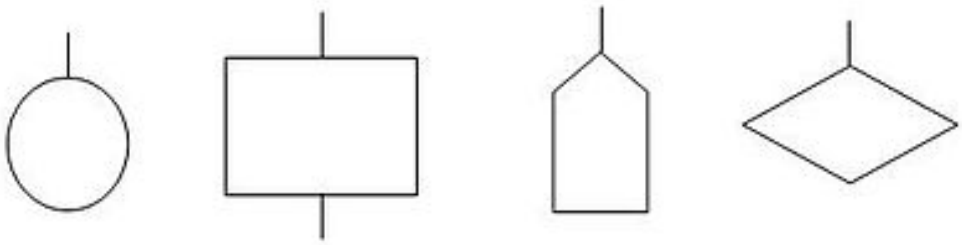


Figure 2.11: Event Symbols of Fault Tree

2) Intermediate event - A fault event that occurs as a result of the logical combination of other events.

3) External event - normally expected to occur outside the system (not a fault in itself).

4) Undeveloped event - an event about which insufficient information is available, or which is of no consequence.

GATE SYMBOLS

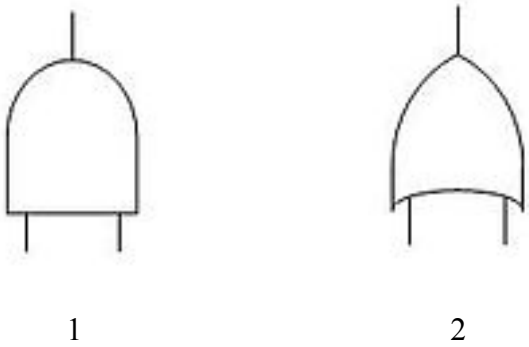


Figure 2.12: Gate Symbols of Fault Tree

Gate symbols describe the relationship between input and output events. The symbols are derived from Boolean logic symbols.

The gates shown in Fig. 2.12 are:

1) AND gate - the output occurs only if all inputs occur (inputs are independent).

2) OR gate - the output occurs if any input occurs.

ANALYSIS :

Many different approaches can be used to analyze a FT, but the most common and popular way can be summarized in a few steps.

The regular FTA analysis involves five steps [FAULTTREE2006]: 1). Define the undesired event to study. 2). Obtain an understanding of the system (system analysis can help with understanding the overall system. System designers have full knowledge of the system and this knowledge is very important for not missing any cause affecting the undesired events). 3). Construct the fault tree (Fault Tree is based on AND and OR gates which define the major characteristics of the fault tree). 4). Evaluate the fault tree (identify all possible hazards affecting in a direct or indirect way the system). 5). Control the hazards identified (after identifying the hazards all possible methods are pursued to decrease the probability of occurrence).

Since FTA is a deductive, top-down method aimed at analyzing the effects of initiating faults and events on a complex system. This contrasts with Failure Mode and Effects Analysis (FMEA), which is an inductive, bottom-up analysis method aimed at analyzing the effects of a single component or function failures on equipment or subsystems [FMEA1996]. FTA is very good at showing how resistant a system is to single or multiple initiating faults, bit it is not good at finding all possible initiating faults. FMEA is good at exhaustively cataloging ini-

tiating faults, and identifying their local effects, but it is not good at examining multiple failures or their effects at a system level. FTA considers external events, FMEA does not.

Alternatives to FTA include dependence diagram (DD), also known as reliability block diagram (RBD) and Markov analysis. A dependence diagram is equivalent to a success tree analysis (STA), the logical inverse of an FTA, and depicts the system using paths instead of gates. DD and STA produce probability of success (i.e., avoiding an undesired top event) rather than probability of an undesired top event.

FAULTCAT TOOL:

The Fault Tree Analysis tool we use in this work is FaultCAT. It is an open source fault tree creation and analysis application written in Java [FaultCAT]. The most important reason we chose to use FaultCAT as our analysis tool is that it supports XML format for both input and output, This is important because the model transformation generating FT in our work produces FT in an XML format that is very close to the input format for FaultCAT.

There are also other tools for doing fault tree analysis, such as Open FTA [OPENFTA] or FaultTree+ [FAULTTREE+], which provide more detailed analysis and more powerful features, but they do not support XML input. This means that an additional step is needed to translate the XML tree generated by our transformation into the input format accepted by the analysis tools. This is the main reason we decided to use FaultCAT as our analysis tool at this stage.

2.1.4 Model Transformations

The notion of model transformations plays a crucial role in the context of MDE [10]. Basically, a transformation is taking some input and subsequently generates the desired output. In the field of model-driven software engineering it is common to focus on model transformations. Based on the outcome of a model transformation, there are two fundamental types that may be distinguished. A transformation may either be a model-to-model (M2M) or a model-to-text (M2T) transformation. In the first case, both the input and the generated output are models. This type of transformation is especially useful for transforming a platform-independent-model (PIM) to a platform-specific-model (PSM) in the context of MDA, or an annotated software model to an analysis model. Another possibility is to define a model-to-text transformation that has a model as input and generates application code or other text artifacts as output. The type of transformation is chosen by the transformation designer and depends on the purpose of the particular transformation.

The Concept of Model Transformations

Model transformations aim at providing facilities and operations for converting an input model to some defined output model. To be more specific, by means of model transformations it is possible to generate some target model that conforms to a target metamodel from a given source model, which conforms to a source metamodel. Designing a model transformation is thus synonymous with specifying which source elements are converted to which target elements in which conditions. In our case we are only concerned with model-to-model transformations and thus, the term model transformation is always referring to a model-to-model transformation in the rest of the thesis.

Figure 2.13 illustrates the basic pattern and the involved artifacts of a typical model transformation. A given source model Model1 conforms to the source metamodel MM1. This meta-model again conforms to some higher-level meta-metamodel MMM. The goal of a model transformation is to generate a target model Model2 out of the source model Model1.

The necessary elements of the source model Model1 are converted into elements of the target model Model2. This target model conforms to the target metamodel MM2 which in turn conforms to the meta-metamodel MMM. The transformation itself, referred to as Mt, conforms to the transformation metamodel MMt. This transformation-specific metamodel is again an instance of the meta-metamodel.

Research on a classification of existing and proposed model transformation approaches identified some commonalities that all model transformation approaches share. For example, all approaches provide different variations of transformation rules, source-to-target relationships, rule organization or tracing. A model transformation language called QVT was specified by the OMG after issuing a Request for Proposal. In this work we are using the ATLAS Transformation Language (ATL), a widely-used transformation language that was released before QVT and has a good tool support. ATL is introduced in the following subsection.

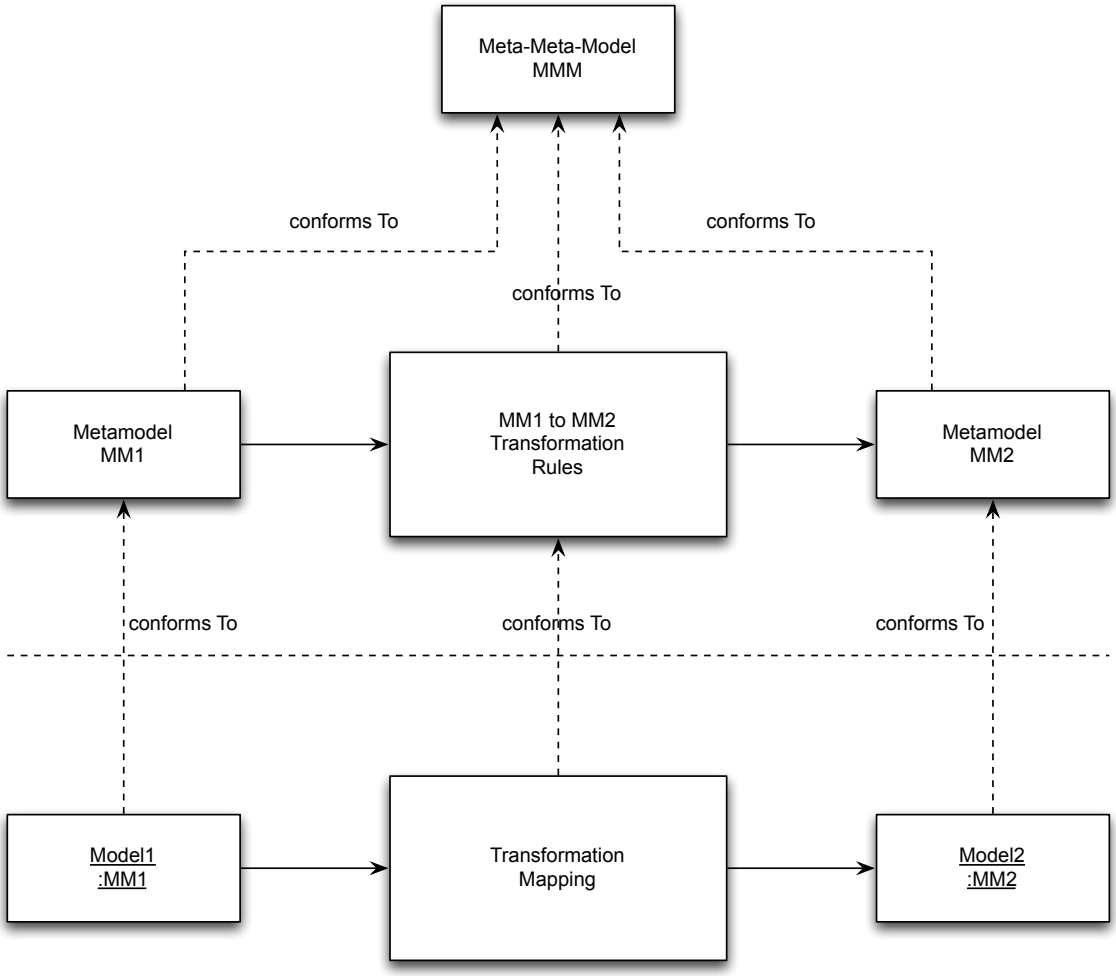


Figure 2.13: Basic Pattern of Model Transformation

2.1.5 Modeling Tools and Transformation Languages

2.1.5.1. Papyrus

Papyrus [PAPYRUS2011] is an official UML2 graphical modeller developed within the Eclipse platform [EMF2013] which aims at providing an integrated and user-friendly environment for editing any kind of EMF model and particularly supporting UML and related profiles such as SysML and MARTE. Papyrus provides diagram editors for EMF-based modeling languages amongst them UML 2 and SysML and the glue required for integrating these editors (GMF-based or not) with other MBD and MDSD tools. Papyrus also offers a very ad-

vanced support of UML profiles that enables users to define editors for domain-specific languages (DSLs) based on the UML 2 standard. The main feature of Papyrus regarding this latter point is a set of very powerful customization mechanisms which can be leveraged to create user-defined Papyrus perspectives and give it the same look and feel as a "pure" DSL editor.

In our case we use Papyrus as our UML modeling tool, so that we can directly apply the MARTE profile as well as apply the DAM profile as a plug-in application

---

2.1.5.2. QVT

QVT (Query/View/Transformation) is a standard set of languages for model transformation defined by the Object Management Group [QVT2011]. As the name QVT indicates, the OMG standard covers transformations, views and queries together. Model queries and model views can be seen as special kinds of model transformations, provided that we use a suitably broad definition of model transformation, such as “a model transformation is a program which operates on models”.

The QVT standard defines three model transformation languages: Operational, Relations and Core. All of them operate on models which conform to MOF 2.0 metamodels; the transformation states which metamodels are used. A transformation in any of the three QVT languages can itself be regarded as a model, conforming to one of the metamodels specified in the standard. The QVT standard integrates the OCL 2.0 standard and also extends it with imperative features.

QVT-Operational is an imperative language designed for writing unidirectional transformations. QVT-Relations is a declarative language designed to permit both unidirectional and bidirectional model transformations to be written. A transformation embodies a consistency relation on sets of models. Consistency can be checked by executing the transformation in check-only mode; the transformation then returns True if the set of models is consistent according to the transformation and False otherwise. The same transformation can be used in enforce mode to attempt to modify one of the models so that the set of models will be consistent. The QVT-Relations language has both a textual and a graphical concrete syntax.

QVT-Core is a declarative language designed to be simple and to act as the target of translation from QVT-Relations. However, QVT-Core has never had a full implementation and in fact it is not as expressive as QVT-Relations. The transformation from QVT-Relations to QVT-Core given in the QVT Standard is not semantics-preserving. Finally there is a mechanism called QVT-BlackBox for invoking transformation facilities expressed in other languages (for example XSLT or XQuery). Although QVT has a broad scope, it does not cover everything that has been considered as a model transformation, view or query. For example, the QVT languages do not permit transformations to or from textual models, since each model must conform to some MOF 2.0 metamodel. Model-to-text transformations are being standardized separately by OMG.

2.1.5.3. ATL

INTRODUCTION

ATL (ATL Transformation Language) is a model transformation language and toolkit [ATL2012]. In the field of Model-Driven Engineering (MDE), ATL provides ways to produce

a set of target models from a set of source models. Developed on top of the Eclipse platform, the ATL Integrated Environment (IDE) provides a number of standard development tools (syntax highlighting, debugger, etc.) that aims to ease the development of ATL transformations. ATL provides a way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. The ATLAS Transformation Language (ATL) is currently the state-of-the-art transformation language in the Eclipse Modeling Framework (EMF). It was developed by the ATLAS group [ATLAS]. ATL was invented as an answer to the Query/View/Transformation Request for Proposal issued by the OMG.

A transformation engineer is using ATL for developing rule-based model transformations between a source metamodel representing the source modeling language and a target metamodel representing the target modeling language. An ATL transformation is then executed on a source model conforming to the source metamodel in order to produce a target model which conforms to the target metamodel. The source model may also be referred to as input model while the target model is also called output model.

DEFINING AN ATL TRANSFORMATION

The definition of a new ATL transformation starts with the definition of a new *ATL module*. An ATL module is the highest-level unit within one ATL file and corresponds to a single model-to-model transformation. The ATL module comprises all parts of the transformation, namely the compulsory header section, the optional import section, the helper section and the rule section.

HEADER SECTION

The header section is used for specifying the name of the ATL module as well as the source and target models. The target models are specified after the keyword *create* while the source models are specified after the keyword *from*. The metamodels of the respective models are defined after the name of the model, separated by a colon. The names of the metamodels are subsequently used in the transformation to address specific metamodel elements. An example of an ATL header is given in CodeFragment 2.1. The name of the module is set to UML2FT.

The target model is named OUT and conforms to the FT metamodel. The source model is called IN and conforms to the UML metamodel.

IMPORT SECTION

The import section is used to specify ATL libraries that may be used to out- source certain code parts, e.g., helpers. Libraries are reusable and may be imported into different ATL modules. They help in keeping the transformation code short and maintainable. Code Fragment 2.1 shows the import of an ATL library called *profileLibrary*.

```
module UML2FT;
create OUT : FT from IN : UML;

uses profilelibrary;
```

Code Fragment 2.1: Import Section

HELPER SECTION

The helper section usually follows after the header section and the optional import statements and provides the possibility to declare attribute helpers or operation helpers. The

term helper actually corresponds to the operation helpers that are equivalent to methods in some programming language like Java. Operation (or functional) helpers always have a name and may take parameters. These helpers may be called by the rules of a transformation or from some other helper function.

```
-- Attribute helper
helper def : stereoName : String = 'DaComponent'

-- Operation helper
helper context UML!Element def: setValue (a: OclAny, b:
String, c:OclAny) : OclAny =
if (not c.OclIsUndefined())
then
    self.setValue (a,b,c)
else
    OclUndefined
endif;
```

Code Fragment 2.2: Attribute helper and operation helper

Operation helpers are used to calculate some return value of a specified data type. Attribute helpers on the other hand may be used to store a specific constant value that may be needed at different points of the transformation. Unlike functional helpers, attribute helpers do not accept any parameter. Code Fragment 2.2 shows the two types of helpers in ATL.

All helpers start with the keyword *helper*, followed by some optional context definition. The context defines for which type of metamodel element the declared helper is applicable. For example, the *setVal* helper in CodeFragment 2.2 may only be called by an *Element* of the *UML* metamodel. The keyword *def* is written after the context definition and in front of the name of the helper. Optional parameters in brackets, the return type and the actual helper definition (introduced by an equality sign) complete the helper declaration.

The attribute helper of CodeFragment 2.2 is named *stereoName* and returns a String value set to the constant *DaComponent*. The operation helper *setVal* is defined for the context of a *UML Element* and takes three parameters. The return type of this helper is set to *OclAny*, the most abstract ATL data type from which all other data types inherit. The body of the helper consists of a simple if-then-else statement which returns some value based on the evaluation of the if-part.

RULE SECTION

The rule section is the most important part of an ATL model transformation. This section contains all transformation rules that are executed in order to generate the target model. The meaning of the term rule and the three different rule types are discussed later on.

ATL is a rule-based transformation language. Therefore, the core of an ATL transformation consists of several rules that are defined by the transformation engineer. Basically, each rule is indicated by the keyword *rule*, followed by its name and the body of the rule. ATL provides three types of rules that satisfy different requirements. Each type is briefly introduced:

Matched rule: A matched rule is the most important rule type. Its purpose is to match a distinct type of source model element and to generate the corresponding target model element. The source model elements that are to be matched are introduced after the keyword *from* ,whereas the target model elements are introduced after the keyword *to*. The initialization details of the resulting target model elements are also determined in the rule. Please note that matched rules are automatically executed by the ATL engine once for each match. This means that the given source model element is matched exactly once and the corresponding

target model element is automatically created during the matching phase. CodeFragment 2.3 gives an example of a simple matched rule. Source model elements of the type Model are matched and the desired target model elements of the type *IntermediateEvent* are generated. Model elements conform to the specified *uml* metamodel while I*ntermediateEvent* elements conform to the *FT* metamodel. The details of a target model element are specified using a set of bindings. A binding determines how features and references of the target element are initialized.

An example can be found of CodeFragment 2.3. Here, the name of the *IntermediateEvent* is initialized with the value of the name attribute of the Model source element.

```
rule Model{
from
    uml: MMa!Model
to
    FT: MMb!IntermediateEvent(
    name <- uml.name
    )
}
```

Code Fragment 2.3: Matched rule example

Lazy Rule: Lazy rules have the same structure as matched rules, with the difference that the keyword lazy is written in front of the keyword rule. A second difference is that lazy rules are not automatically executed but they have to be called explicitly by another rule. As a result, lazy (and called) rules are not executed during the matching phase but are evaluated during the last phase (target model elements initialization phase) of an ATL transformation execution. An example of a lazy rule is given in CodeFragment 2.4.

```
Lazy rule  getLifeline{
from
    uml: MMa!Lifeline
to
    FT: MMb!BasicEvent(
    name <- uml.name
    )
}
```

Code Fragment 2.4: Lazy rule example

Called rule: This type of rule must also be invoked explicitly. As opposed to a matched rule, a called rule does only include the definition of the target model element. Thus, no source model element for matching purposes is required. An example of a called rule can be seen in CodeFragment 2.5.

```
rule NewModel (na: String) {
to
    FT: MMb!Model
    do {
     name <- na
    }
}
```

Code Fragment 2.5: Called rule example

Rule inheritance: Since the release of the ATL 2006 version, rule inheritance is also provided by the concept of abstract rules. The ATL developer can define an abstract super rule and arbitrary many sub rules that extend the super rule. By this, all the bindings that are defined in the abstract rule are automatically copied and combined via a union operator with the bindings of the sub rule. For the sake of completeness, however, it should be mentioned that rule inheritance is also possible between concrete rules. The concept of rule inheritance is especially useful when having class inheritance in the source and the target metamodel.

ActionBlock: Every ATL rule may contain one optional ActionBlock, also referred to as *do*-block. This block, introduced by the keyword *do*, contains an arbitrary number of imperative code statements that may be used for setting features and/or references of the generated target model element. The code statements defined within an ActionBlock are executed in a sequential order after the initialization of the corresponding target model element is completed. An example is illustrated in CodeFragment 2.6. Every time the called rule *newModel* is executed, the target model element *Model* gets initialized and afterwards, the name of the *Model* is set.

```
rule newModel (na: String){
to
    t: UML!Model(
      name <- na
    )
}
```

Code Fragment 2.6: Called rule with ActionBlock

ATL LANGUAGE CHARACTERISTICS:

ATL is a hybrid transformation language as it provides both declarative and imperative language features.

Imperative code*:* Imperative programming means that the programmer specifies the exact flow of the program instructions, i.e., the successive program steps. Imperative code includes programming constructs like for-loops and if-then-else-statements. The imperative transformation code, as for example defined inside an ActionBlock, is executed in a strict sequence.

Declarative code*:* Declarative programming on the other hand is not concerned with the flow of operations but rather focuses on the desired outcome.

The declarative part of ATL comprises all matched rules that match certain types of source model elements. Nevertheless, also imperative code is needed in the form of called rules, ActionBlocks, attribute helpers and operation helpers. Imperative code parts are used when complex computations are not feasible by means of declarative constructs.

The reason we choose to use ATL as our transformation language rather than QVT in this work is that, so far as a mature model transformation language, ATL has a larger community of users, and it has better tools support, with better documentation as well as an active discussion forum, which will make our implementation more convenient and efficient.

2.2 Related Work

There exists several works focusing on software dependability analysis by transforming UML model to different analysis models, as surveyed in [BERNARDI2012]. Some of them chose fault tree as their target analysis model, but only a few of them applie model transformation languages to achieve automatic generation. The most related works are briefly discussed below:.

[DAMBROGIO02] focuses on the software architecture of the system, and introduced a method that translates UML-based specification into a fault tree reliability model. The path-based approach is considered, and the UML system specification is mapped onto a fault tree (FT) model to predict the system failure rate. The method is implemented into a tool that gives the reliability prediction of software architectures described by UML sequence dia-

grams and deployment diagrams. Although no UML extension standard mechanisms are used, several UML model elements whose failure (basic events in Fault tree models) can lead to the system failure (top-event in Fault tree models) are identified, such as failure of nodes and communication paths, call/return actions and operations.

[DUGAN02] describes a framework for modeling computer-based systems, based on UML, that facilitates automated dependability analysis during design, using Dynamic Fault Trees as target formalism to evaluate the system unreliability of fault-tolerant software systems at design stage. The authors define a set of stereotypes to enrich UML system models with information needed for the reliability analysis and to define fault propagation and software-to-hardware allocation. In particular, tags are used to define input parameters, such as failure rate of system components and error propagation probability. Special purpose gates are defined to capture sequence dependencies which frequently arise when modeling fault tolerant computer systems. Such special gates are: functional dependency gate, for modeling situations where one component's correct operation is dependent upon the correct operation of some other component; spare gate, for modeling cold, warm and hot pooled spares; and priority-AND gate, for modeling ordered AND-ing of events. Thus the approach supports the modeling and analysis of sequence error propagations that lead to dependent failures, redundancies and reconfiguration activities. An algorithm to automatically synthesize dynamic fault trees (DFTs) from the UML system model is also developed.

[GRUNSKE05A] proposed a component- based dependability analysis technique that annotates components with failure mode assumptions. The probabilities and dependencies of these failure modes are specified by Component Fault Trees (CFT's). Based on these CFT's

and the architectural model the propagation of failures throughout the system can be automatically determined and a quantitative analysis is possible.

[GRUNSKE05B] outlined a technique which annotates components with modular failure mode assumptions, described in the Failure Propagation Transformation Notation (FPTN) and generates an analyzable failure propagation model for the complete system. Based on this technique, a model-based safety evaluation is possible, which enables the automatic generation of safety cases based on system models.

[HARPER07] indicated that failure analysis is only complete with respect to a particular failure model, which must be a prior assumption at the start of any analysis. So that they introduced six basic software failure types, which corresponded to HAZOP guide words. The map between UML sequence diagrams and fault tree model is provided along with their failure model based on a particular case study.

[LAUER11] proposed a modeling approach that focuses on reusability and automatic fault tree synthesis of the models. They identified capturing fault propagation and fault containment information as a major challenge in automatic fault tree synthesis and offer an application dependent and an application independent approach to modeling this kind of information.

The authors also introduced an algorithm that transforms the UML model into a fault tree representation of the respective system and validate their approach using an example from the automotive domain.

The comparison of closely related works is shown in the table below.

	Source Model	Target Model	Limitation
[XIANG11] "Automatic Synthesis of Static Fault Trees from System Models"	SysML configuration diagrams	Static Fault Tree	No transformation language are being used
[JOSHI07] "Automatic Generation of Static Fault Trees from AADL Models"	AADL Models	Static Fault Trees	Failure modes still need to be verified by hand
[LAUER11] "Fault Tree Synthesis from UML Models for Reliability Analysis at Early Design Stages"	UML Component Diagram, State Machine	Static Fault Trees	No transformation language are being used
[D'AMBROGIO02] "A Method For The Prediction Of Software Reliability"	UML Sequence diagrams and Deployment diagrams	Static Fault Trees	allocation and failure propagation not considered
[DUGAN02] "Automatic Synthesis of Dynamic Fault Trees from UML System Models"	UML Class diagram and Deployment diagram	Dynamic Fault Tree	Mostly focuses on architecture, system behavior not considered
[HU11] "A Method of FTA Base On UML Use Case Diagram"	UML Use case diagram and Activity diagram	Static Fault Trees	only use cases are being transformed
[HARPER07] Fault Tree Analysis of UML Designs	UML sequence diagrams	Static Fault Trees	semi-automatic generation, unnecessary redundancy
[GRUNSKE05A] "Automatic generation of analyzable failure propagation models from component-level failure annotations"	Failure Propagation Transformation Notation (FPTN)	Component Fault Trees	Expert knowledge and manual work required
[GRUNSKE05B] "An Automated Dependability Analysis Method for COTS-Based Systems"	System Structure Diagram	Component Fault Trees	Predictions depend strongly on expert knowledge and premature models
[DOMIS08] "Integrating Safety Analyses and Component-Based Design"	Functional Specification	Component Fault Trees	Manual work required, failure modes can not be determined automatically

Table 2.2: Summary of related works

	Source Model	Target Model	Limitation
[LAUER09] "Fault Tree Generation from EMF Models"	Customized EMF Models	System Fault Tree and Behaviour Fault Tree	No transformation language are used, profiles could be added in
[HASSAN05] UML Based Severity Analysis Methodology	UML Use Case Diagram and Sequence Diagram	FFA (Functional Failure Analysis)	Heavy manual work required

Table 2.2: Summary of related works continued

As we can see from the discussion of related works, some of the existing approaches (e.g., those considering different failure modes and those using component fault trees) do contain manual steps throughout the derivation of the fault trees, so it is difficult to completely automate the entire generation of the fault trees. Since our focus was on complete automation of the transformation from UML to fault tree models, we have considered the generation of traditional fault trees, which do not require manual intervention. [HARPER07] proposed to consider every possible failure mode for each different element of the UML sequence diagrams, but this may bring too many unnecessary redundant events that may not really make sense. Therefore, we choose not to take different kinds of failure modes into account. The works in [D'AMBROGIO02] and [DUGAN02] have presented a clearer logic for automating the model transformation by providing high-level algorithms (implemented in general purpose languages rather than model transformation languages). The former has taken system behavior into account, while the latter focused mainly on the architecture. One of the objectives of our work was to take into account the interaction between software components like in [D'AMBROGIO02] and the redundancies in system architecture like in [DUGAN02]. Another main objective was to implement the transformation in a specialized

Model Transformation Language (in our case ATL) instead of a general purpose languages, like in the existing works.

CHAPTER 3. TRANSFORMATIONS

3.1 Source Model

The source model of the transformation proposed in the thesis includes UML2 sequence diagrams, use case diagrams, and composite structure diagrams with MARTE/DAM performance annotations. A sequence diagram is a scenario notation used to describe the behaviours of distributed systems at an abstract level or to represent a component-based system behavioural requirements [SUPER]. They describe a finite sequence of operations between different objects. A use case diagram is a representation of a user's interaction with the system and depicting the specifications of a use case [SUPER]. In our case each use case is associated with an interaction. A composite structure diagram shows the internal structure of a classifier, including its interaction points to other parts of the system [SUPER]. It shows the configuration and relationship of parts, which together perform the behavior of the containing classifier. We chose to use composite structures in order to model the represented components, allocated hardware devices and connectors.. Also, as mentioned in section 2.1.1, instead of using deployment diagrams to show the actual software to hardware allocation of the system, we chose to use the stereotype 《allocate》 from the MARTE profile. The main reason of this is that Papyrus has a bug in the implementation of deployment diagrams, in the sense that does not allow users to model communication paths between processing nodes. Thus we are unable to model a link (for example LAN or WAN) between hardware nodes, which is important for a dependability model. Furthermore, Papyrus does not allow users to connect two ports using a connector between two classes, but only properties within a class could be directly connected. In order to show such connections between components in com-

posite structure diagrams, we modeled the whole system as a big interconnected class and all its components as properties, so that they could be linked by using the Papyrus tool. We have already reported this bug to the Papyrus team; when they will solve this problem, we will refine our work to separate these components and use ports as connector ends.

As the ATL transformation rules are defined based on the source and target metamodels, we will present below the UML metamodel fragments relevant to our work. All the UML metamodel diagrams presented in this section were selected from the official OMG document UML2 Superstructure [SUPER]. Figure 3.1 gives a snapshot of the UML2 metamodel for use case diagrams. We can see the class UseCase has a property of subject of type Classifier, which gives a classifier the capability to own use cases. In our work Interactions are such kind of classifiers who own use cases, and each interaction only owns one use case.

Figure 3.2 shows the metamodel for Composite Structure Diagram. As we can see, the class Property specializes ConnectableElement, which is an abstract metaclass representing a set of instances that play roles of a classifier. This is useful when in our case we consider the whole model as one big interconnected element, then the components are shown as properties which may be connected within the model.

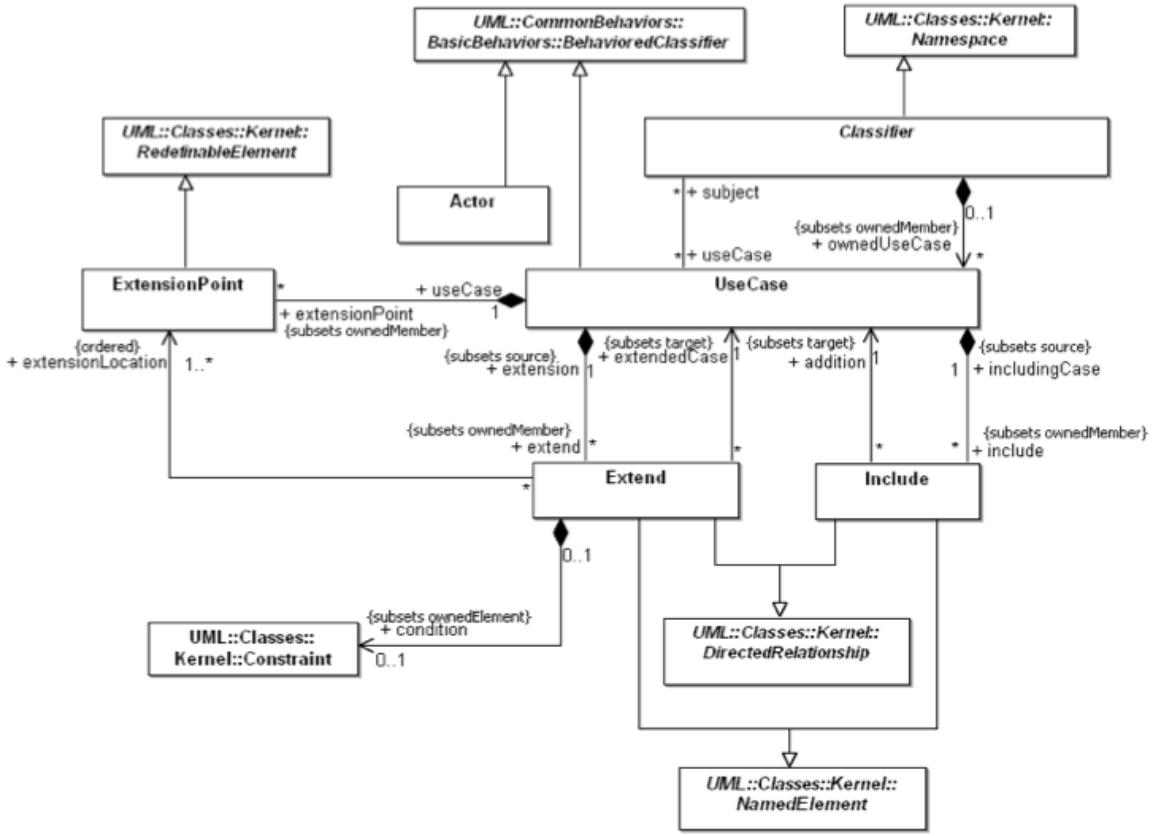


Figure 3.1: UML2 metamodel- The concepts used for modeling use cases

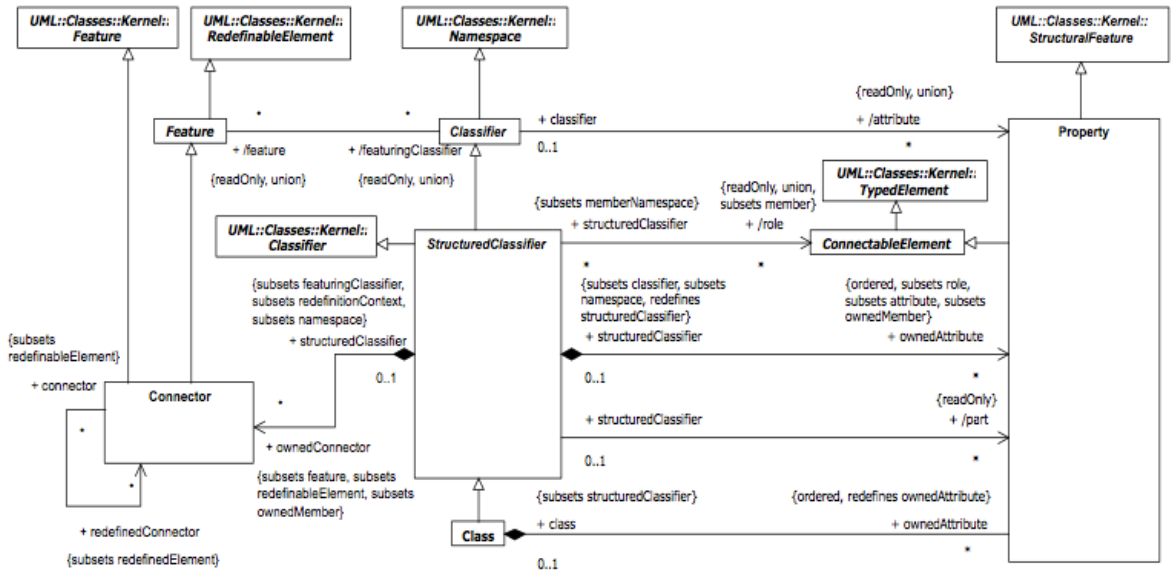


Figure 3.2: UML2 metamodel- Structured classifier

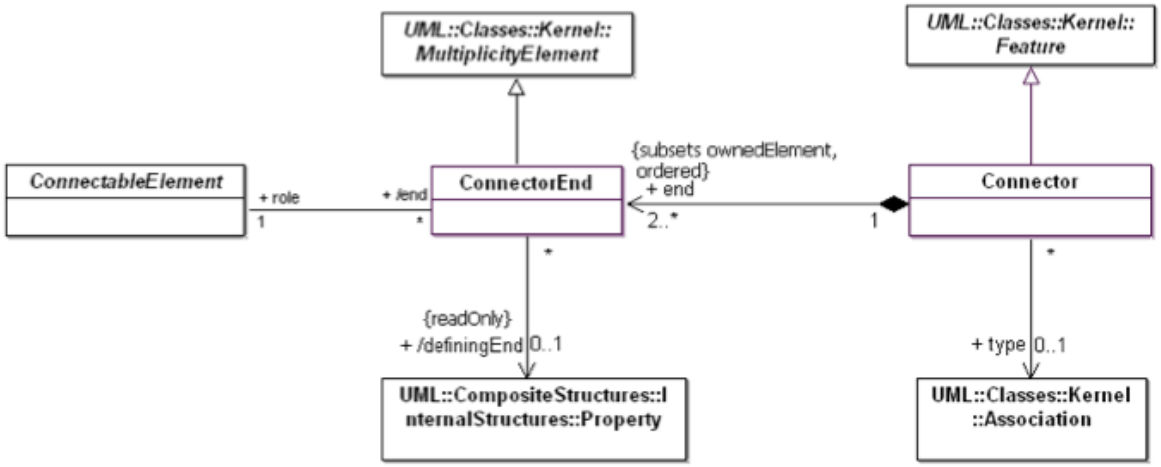


Figure 3.3: UML2 metamodel- Connectors

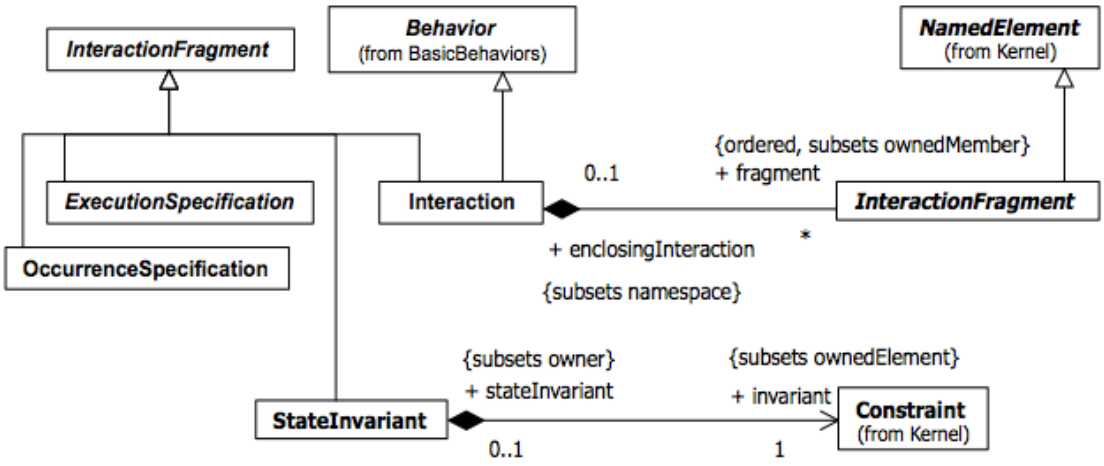


Figure 3.4: UML2 metamodel- Interactions

Figure 3.3 shows the metamodel for connectors of UML composite structure diagram, The class Connector specifies a link that enables communication between instances. Each connector may be attached to two or more connectable elements, each representing a set of instances. Each connector end is distinct in the sense that it plays a distinct role in the communication realized over a connector. The communications realized over a connector may be

constrained by various constraints (including type constraints) that apply to the attached connectable elements [SUPER].

As we are using sequence diagrams to show the behavior of the system. Figure 3.4 describes the metamodel for Interaction, and Figure 3.5, 3.6, 3.7 and 3.8 shows the metamodel for lifeline, message, occurrence, and combined fragment respectively. All these are elements we use in our transformation. We can see from Figure 3.5 that lifeline can represents connectable elements in composite structure diagrams; in our work these elements are properties. Lifelines are covered by OccurrenceSpecification, as we can see from Figure 3.7. An OccurrenceSpecification can be either ExecutionOccurrenceSpecification or MessageOccurrence- Specification, which are two of the elements we used in our transformation. For message, as shown in Figure3.6, each of them has two MessageEnds, which is an abstract NamedElement that represents what can occur at the end of a Message [SUPER]. In our work, we considered all the MessageEnds as OccurrenceSpecifications. The detailed transformation will be discussed in the next section. Figure 3.8 shows the metamodel for a combined fragment of a sequence diagram, which defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. The semantics of a CombinedFragment is dependent upon the interactionOperator. A CombinedFragment has one or more InteractionOperands, which may contain fragments of other InteractionFragments.

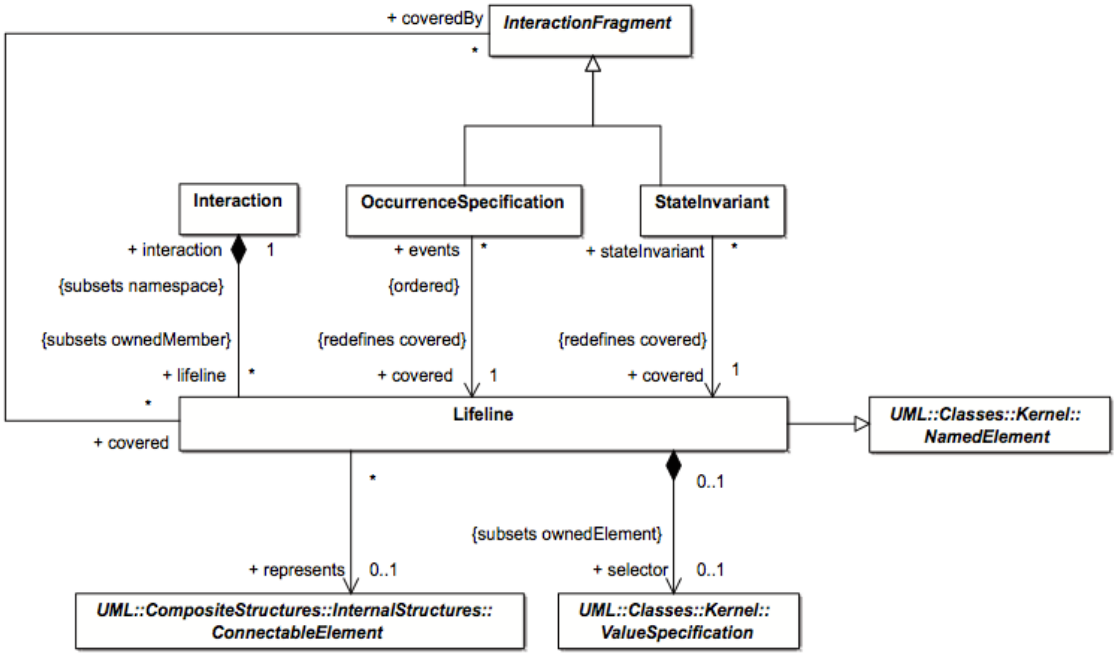


Figure 3.5: UML2 metamodel- Lifeline

Our UML model makes use of the MARTE Performance Analysis Modeling (PAM) package via the Papyrus UML editor. Each use case use is stereotyped with 《PaStep》, specialized from 《GaStep》, and defines a number of non-functional properties. We use the property 'prob' to get the execution probability of each use case. The stereotype 《Allocation》 in MARTE is being used to associate individual application elements to individual execution platform elements. In our case, we use it to associate a software component with its allocated hardware processor.Each allocation link has a source which is an ApplicationAllocationEnd and a target which is an ExecutionPlatformAllocationEnd.

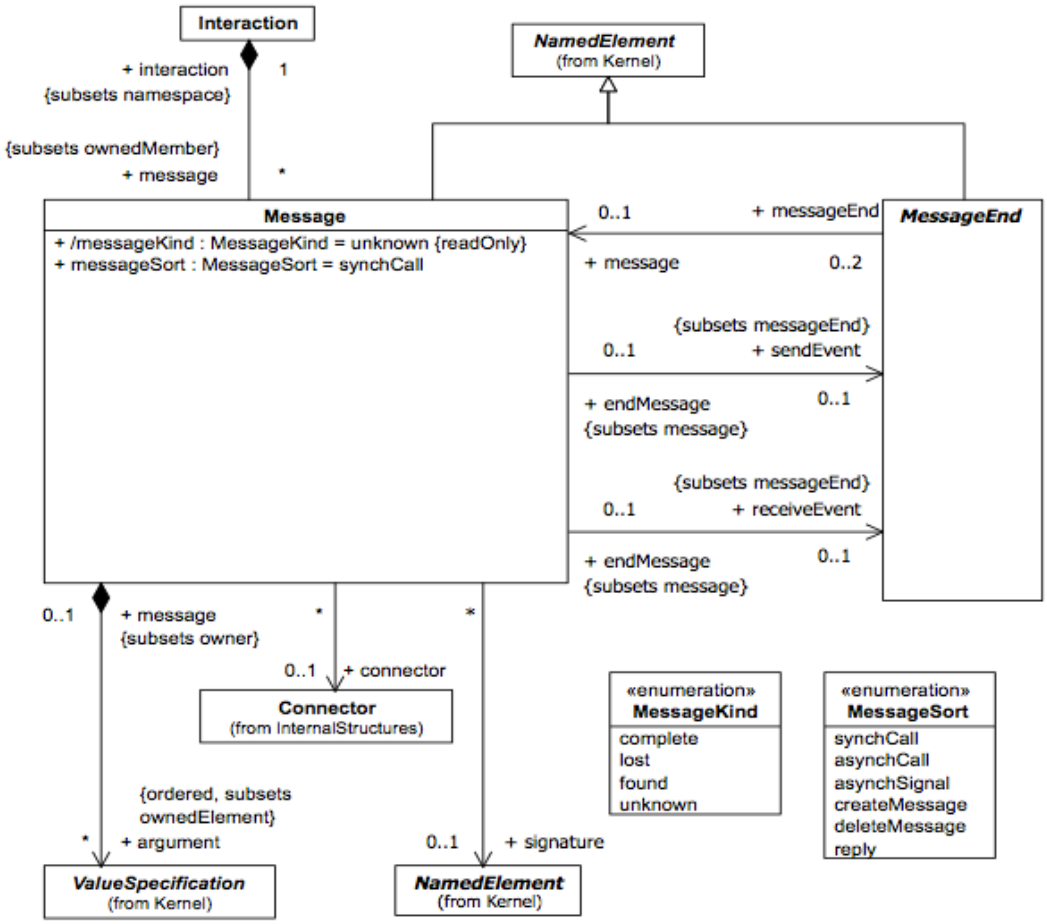


Figure 3.6: UML2 metamodel- Messages

We also apply the DAM profile as a profile application plugin to our source model to add dependability properties. For instance, we apply the stereotype 《DaComponent》 to each component of the system, where the 'Component' class is associated with the 'Failure' (a complex data type). The latter is used to define the attribute 'failure' of 'DaComponent' (with complex type 'DaFailure', in which we can define the failure rate of the component). The stereotype 《DaConnector》 also has an attribute 'failure' so that we applied it to the hardware connections in our system in order to express the associated failure occurrence probability. The stereotype 《DaRedundantStructure》 is used to characterize the impairments affecting simultaneously the set of fault tolerant components belonging to a redun-

dant structure, of which we used the stereotype 《DaSpare》 to model the redundant elements of the system. for the stereotype 《DaSpare》 has an attribute of 'substitute For', of the type of 'DaComponent'. The detailed use of stereotypes in our model transformation is explained in the next sections.

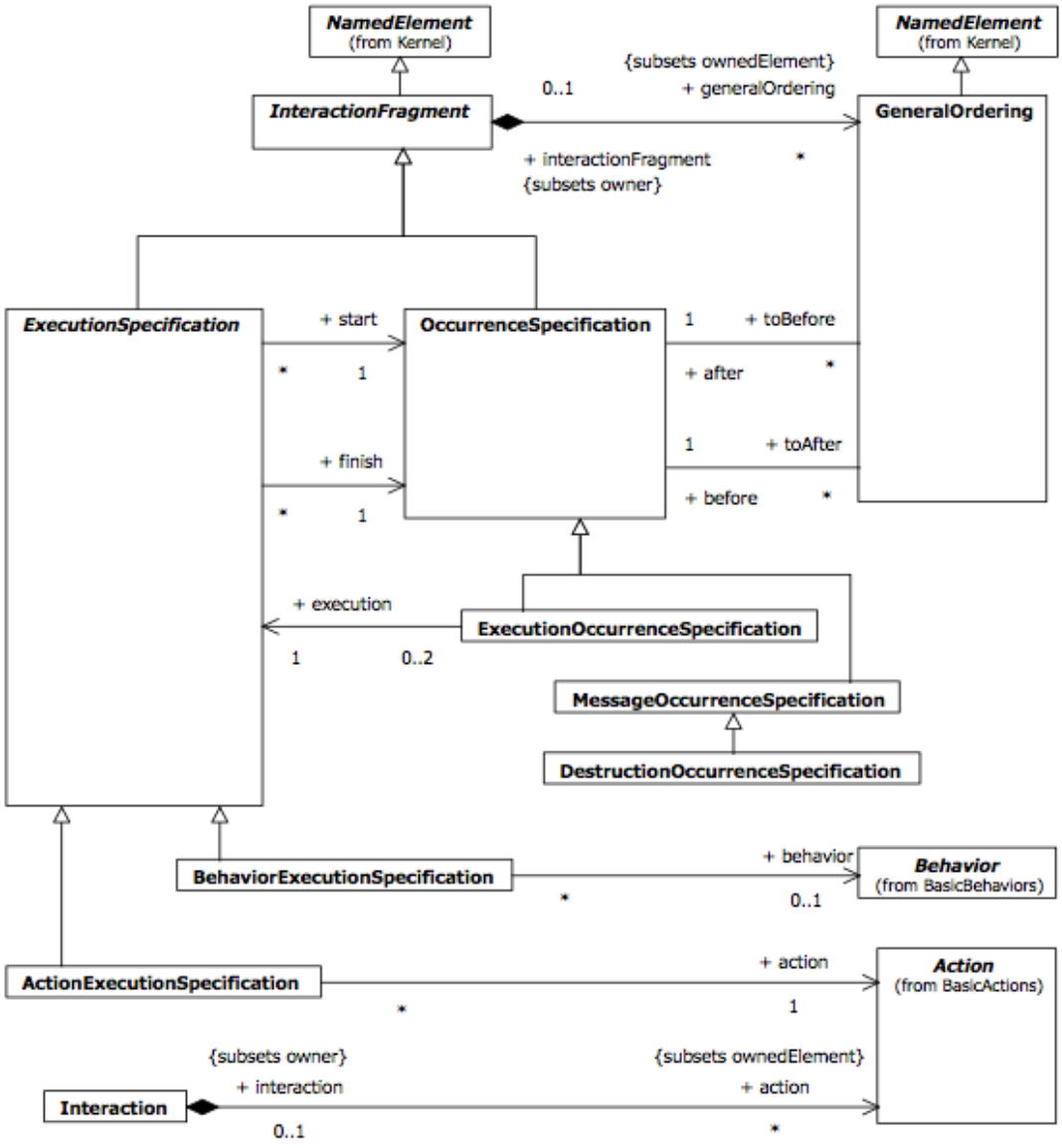


Figure 3.7: UML2 metamodel- Occurrences

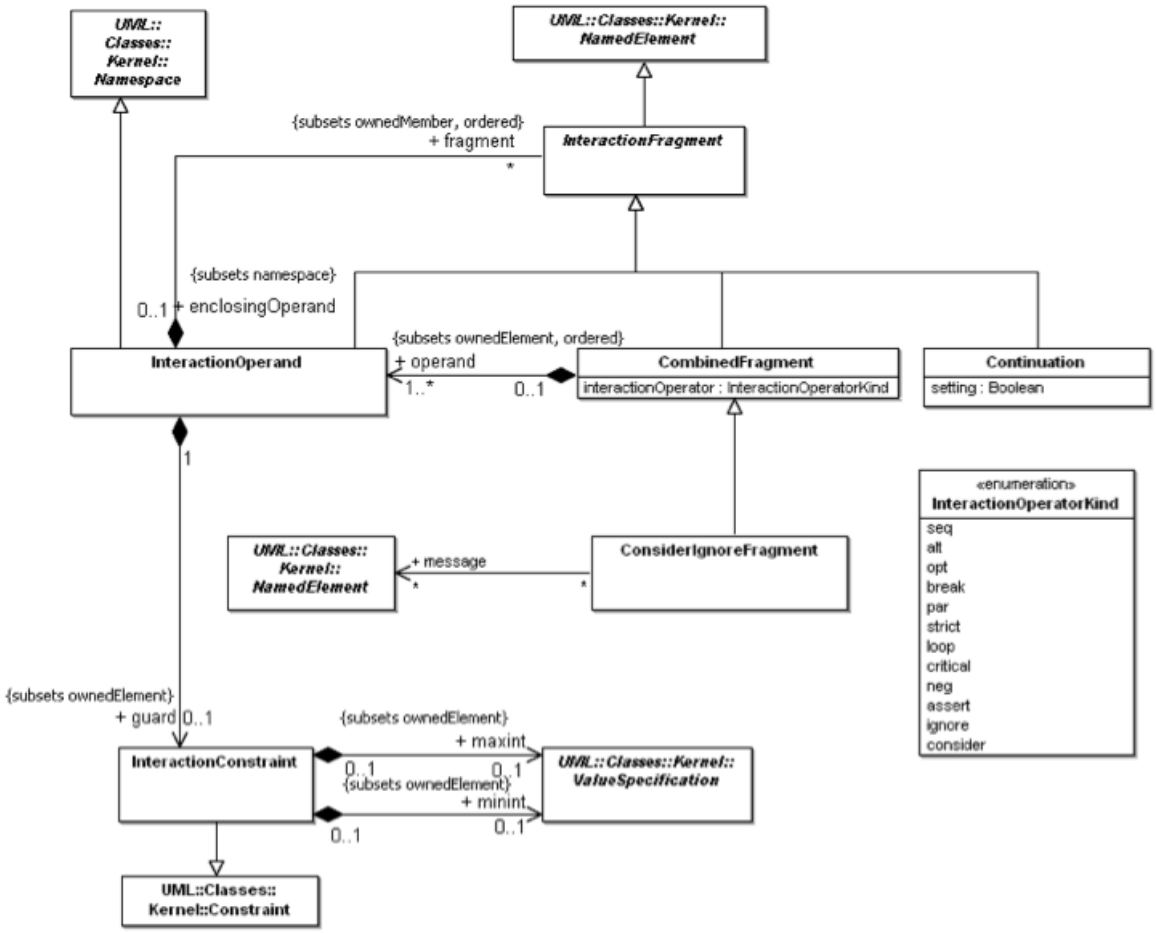


Figure 3.8: UML2 metamodel- CombinedFragment

An annotated UML source model example (Composite structure diagram, Use case Diagram and sequence diagram) is shown as Figure 3.9 below, which uses all the stereotypes mentioned above. The use case is stereotyped with 《PaStep》 from MARTE_Profile and it has an attribute of probability (prob) which is a real number typed with the NFP Types defined in the *Basic_NFP_Types* model library [MARTE], and it requires the input from the user.

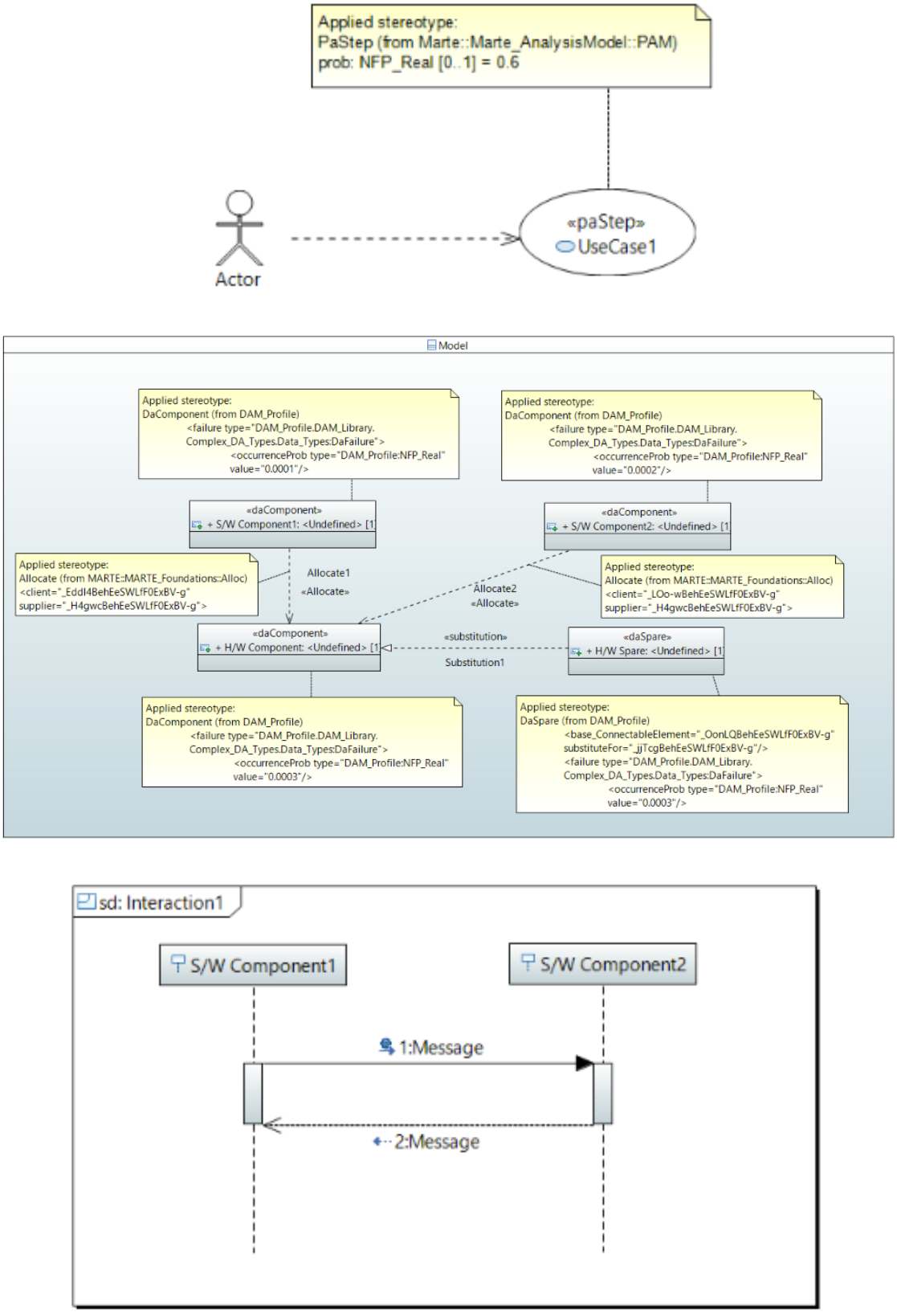


Figure 3.9: Annotated UML source model example

In the Composite Structure diagram, :S/W Component 1 and :S/W Component2 and H/W Component are stereotyped with 《DaComponent》 from DAM_Profile, which has an attribute of “Failure”, with the type of “DaFailure” and it allow users to input the “occurrenceProb” to each of the component. The hardware spare is stereotyped with 《DaSpare》 and shows its substituted component (in this example is “H/W Component”), it also has the complex property type of “Failure”, same as stereotyping other components. The stereotype 《Allocate》 from MARTE shows both the client and supplier of allocation, For instance in the example model, both of the software components are being allocated to the same H/W Component, so that the two client xmi ids are the same. There is no more stereotypes being used in sequence diagram, each lifeline represents a component in the Composite Structure diagram.

3.2 Target Model

As there is no standard metamodel for Fault Tree defined in advance, the metamodel we referred to in our transformation is built according to the analysis tool we use (FaultCAT). As shown in Figure 3.10, the Fault Tree model contains elements named FTelements, which can be either Events or Gates. A FTelement can have children elements, for example an event can have a child gate, and a gate can have several children events. The constraint here is that a gate cannot have another child gate, and an event cannot have more than one child gate.

The metamodel from Fig. 3.9 is a simplified Fault Tree metamodel, which is fully supported by the FaultCAT tool used in this work. (As already explained in the previous chapter, we selected FaultCAT because it is the only open-source tool which directly support XML input/output.) In the literature there are extended fault trees models, such as dynamic

fault trees which contain other types of elements (such as spare gate, functional dependency gate or conditional events). Such elements are not being considered in our work yet, but we may refine our transformation in future work.

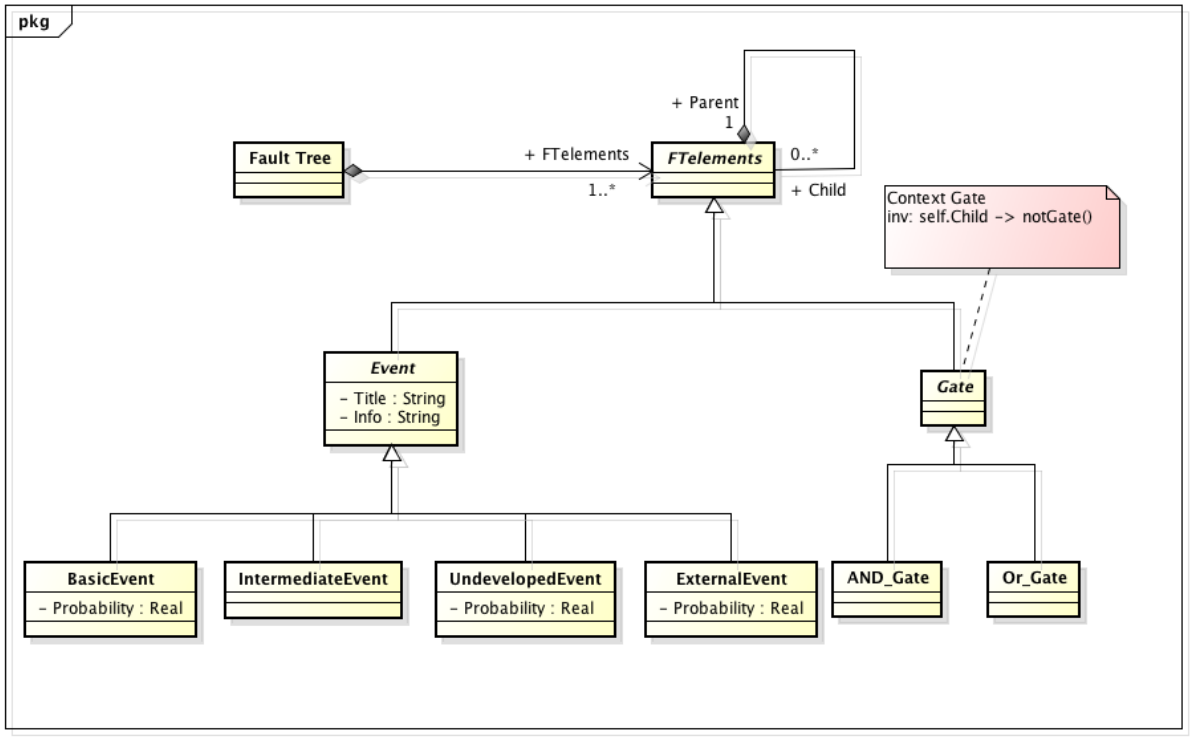


Figure 3.10: Fault Tree Metamodel

As we can see from Figure 3.10, the Fault Tree model contains elements named FTelements, which can be either Events or Gates. Every event has two attributes: Title and Info (which can be used to store extra useful informations). Some specialized events types (BasicEvent, Undeveloped Event and ExternalEvent) have also a Probability attribute, which is the occurrence probability of the event. The comment shows that the gate element can not have another gate as a child element. Figure 3.11 shows an example fault tree. Note that UndevelopedEvent and ExternalEvent are not used in our model transformation, and the content of Info is not shown in the figure.

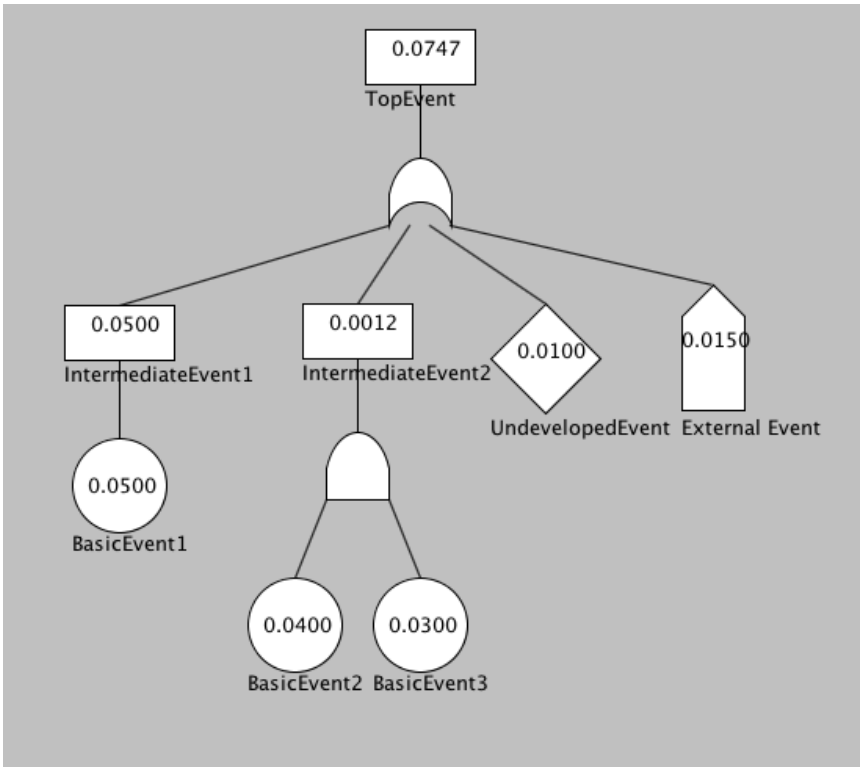


Figure 3.11: Fault Tree Example

3.3 Source to Target Mapping

Based on the metamodels for both UML and Fault Trees, , the mapping from source to target can be briefly described as in the table 3.1 below.

UML+MARTE/DAM diagrams	Fault Tree Model
Model	IntermediateEvent(TopEvent)
	OR_Gate
《PaStep》 UseCase	IntermediateEvent
Interaction	OR_Gate

Table 3.1: Mapping from source to target model

UML+MARTE/DAM diagrams	Fault Tree Model
CombinedFragment	IntermediateEvent
	OR_Gate
InteractionOperand	IntermediateEvent
	OR_Gate
BehaviorExecutionSpecification	IntermediateEvent
	OR_Gate
《DaComponent》 S/W Component	BasicEvent
《DaComponent》 H/W Component (without spare)	BasicEvent
《DaComponent》 H/W Component (with spare)	IntermediateEvent
	AND_Gate
	BasicEvent
《DaSpare》 Component	BasicEvent
《DaConnector》 Connector	BasicEvent

Table 3.1: Mapping from source to target model continued

This table shows the schematic mapping from a UML model to a Fault Tree model. The fault tree is generated from the top (which represents the undesired event that the system fails) to the bottom (each leaf is a basic events). In other words, the tree is constructed backwards: for every gate, first is generated the output event and then the input events. The top level of the fault tree (top event) corresponds to the root of the source model (a Model element), which is the output of an OR gate; The inputs of this OR gate are intermediate events corresponding to the failure of each use case. Each use case should have an execution probability, but for an intermediate event, we are unable to set any value to it because the interme-

diate event value is a result calculated by input events, and the FaultCAT tool does not support conditional event. Our solution to this problem is to transform the occurrence probability value to the attribute of info of the intermediate event in the fault tree model. Then, when assigning the value to any of the children basic event of this intermediate event, we multiply the original failure occurrence probability value by the one in the intermediate event's information.

A more detailed description of the transformation method is given in the implementation section. Since we consider that each use case is associated to a single interaction, the associated interaction is transformed into an OR gate, whose inputs are all the interaction fragments inside the interaction. The level generated next includes either a combined fragment, or the messages and their execution occurrences that do not belong to any combined fragment. Each of the combined fragments is transformed into an intermediate event proceeded by an OR gate. Then the next level is InteractionOperand, also transformed into an intermediate event and an OR gate, followed by the next inner fragments. Each BehaviorExecutionSpecification is transformed into an intermediate event which is the output of an OR gate. Each of its covered lifelines represents a software component, which its allocated on a hardware component; all components are transformed into basic events. The failure probability of a component is assigned to the value of the corresponding basic event. The connectors associated with messages are being transformed into basic events as well (the detailed mapping is discussed in next section). The components that are stereotyped by 《DaSpare》 are redundancy components of other components, which are transformed into basic events. For the H/W components that have spares, the transformation is slightly different and will be explained later.

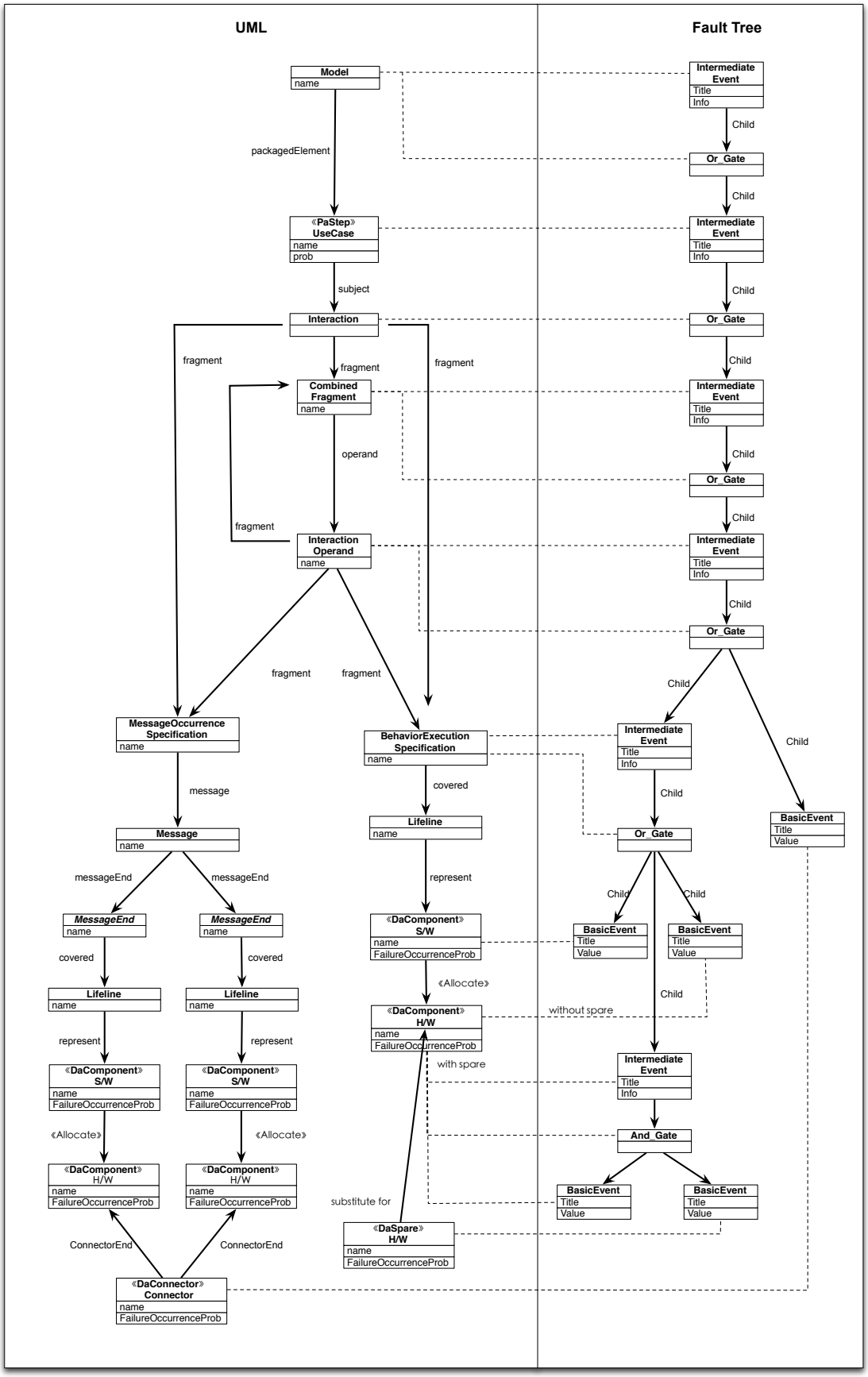


Figure 3.12 Mapping method from UML to Fault Tree model

Figure 3.12 shows the mapping from UML model to Fault Tree model, where the dotted line shows the mapping relationship while the solid lines show actual associations.

As we can see from Figure 3.12, most of the important mappings were presented in table 3.2 above. It is worth noting that for Messages in the UML model, each of them has two message ends located on different lifelines; each lifeline represents a single software component, and each software component is allocated to a hardware component.

We consider that a message may fail due to the failure of the hardware link conveying the message. A message can then be mapped to a connector between the hardware components on which the two software components exchanging the message are allocated. For a hardware component, we have two different mapping methods. For those that do not have spare components, we directly map the hardware component to a basic event representing the failure of the component, and whose failure occurrence probability is an attribute of the the applied stereotype 《DaComponent》. For the hardware components with spares, we generate an intermediate event representing the failure of all hardware components, along with an And Gate (only if all hardware components fail, then the system would fail). The input events to the AND gate are basic events representing the failure of the original hardware component along with its spares as basic events.

3.4 Transformation Rules

3.4.1 Overview

The transformation rules we propose here include: a) basic rules representing the transformation of the basic structures from UML to Fault Tree model as well as quantitative mapping; b) redundancy rules describing the mapping of hardware spares; and c) customized

rules used to provide choices for users to decide whether to include the message failure into the transformation or only to focus on the execution of components. The detailed mapping are explained in the next few sections.

3.4.2 Basic Rules

3.4.2.1. Model

The top level of our target fault tree model is transformed from the UML Model element, which is the root container of the source model. As we can see from CodeFragment 3.1, the Model element of the source model indicated in section "from" is transformed into the top intermediate event of a Fault Tree, which is the root container of the target model, and an Or_gate. The top intermediate event in a fault tree shows the undesired failure that need to be analyzed; in our case the top system failure covers all possible failures that may occur during the execution of the software system.

```
rule Model{
from
    uml: MMa!Model
to
    Event: MMb!IntermediateEvent(
           Title <- uml.name,
           Child <- Gate
           ),
    Gate: MMb!Or_Gate(
           Chid <- uml.packagedElement (type = “uml:Use-
Case”)
           )
}
```

Code Fragment 3.1: Transformation rules - Model

The rule shown in Code Fragment 3.1 contains two target patterns in section "to": the first,labeled with the variable "Event" represents a generated target node of type IntermediateEvent and the second, labeled with the variable Gate represents a generated target node of type Or_gate. The bindings of the first target node Event: (Title <- uml.name, Child <- Gate) indicate how its features (attributes and/or references)are initialized. More specifically, the attribute Event.Title is initialized with the name of the source Model element, and the reference Event.Child points to the target element generated by the target pattern Gate. The binding of the second target pattern Gate: Chid <- uml.packagedElement (type = “uml:Use-Case”) indicates that the reference Gate.Child is initialized to point to the target model element generated by a different transformation rule corresponding to the UseCase source model element.

3.4.2.2. Use Case

Each use case within the model is mapped to an intermediate event that represents the failure of the use case, as shown in Figure 3.13 and CodeFragment 3.2 below

We would like to take into account in the computation of the probability of occurrence of the top event the occurrence probability of different use cases. Unfortunately, the tool does not allow us to set such an attribute for an intermediate event. As an work around this problem, we map the probability of each use cases to the intermediate event’s Info value. In the analysis step, we can multiple every of a use case’s children basic event’s probability by this value, in order to get a reasonable result, but because FaultCAT tool does not support conditional event yet, this part of calculation still needs to be done by hand at current stage.

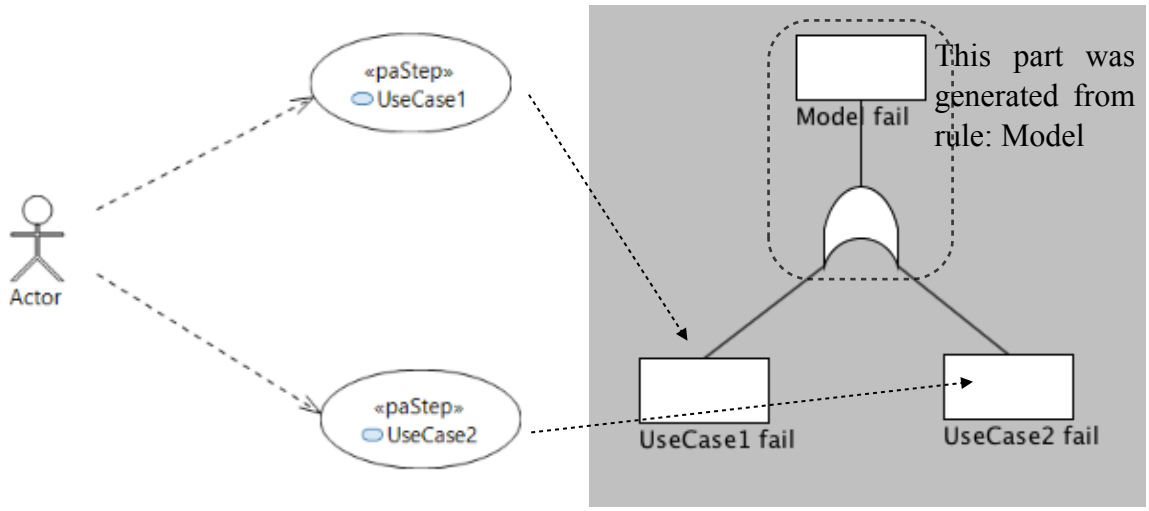


Figure 3.13: Transformation rule example - UseCase

```
rule UseCase{
from
    uml: MMa!UseCase
to
    FT: MMb!IntermediateEvent(
          Title <- uml.name + 'fail',
          Child <- uml.subject,
          Info <- uml.getAppliedStereotype.prob
          )
do  {
    uml.applyProfile (MMa!Profile.allInstances() -> se-
lect (p|p.name = 'Marte' ) -> first());
    }
}
```

Code Fragment 3.2: Transformation rule - UseCase

3.4.2.3. Interaction

An Interaction contains all the visual elements that make up a UML2 Sequence Diagram. Since we consider each use case is associated with one interaction, as shown in Figure 3.14 and CodeFragment 3.3 below, each Interaction can then be mapped to an Or_Gate in the target model below its associated UseCase.

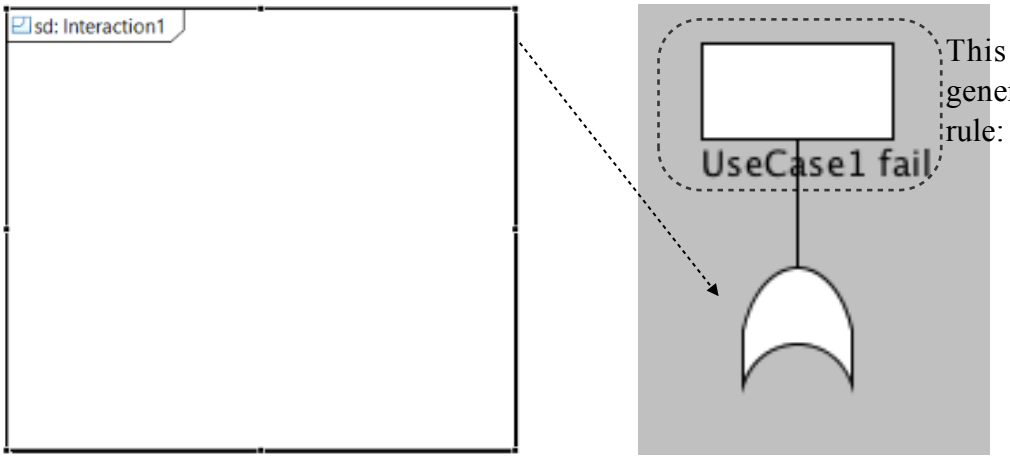


Figure 3.14: Transformation rule example - Interaction

```
rule Interaction{
from
    uml: MMa!Interaction
to
    FT: MMb!Or_Gate(
           Child <- uml.fragment
    )
}
```

Code Fragment 3.3 Transformation rule - Interaction

3.4.2.4. CombinedFragment

As we can see from Figure 3.15 and CodeFragment 3.4, each combined fragment is transformed to an intermediate event followed by an Or_Gate. The name attribute of the intermediate event contains the name of the combined fragment itself and its operator kind.

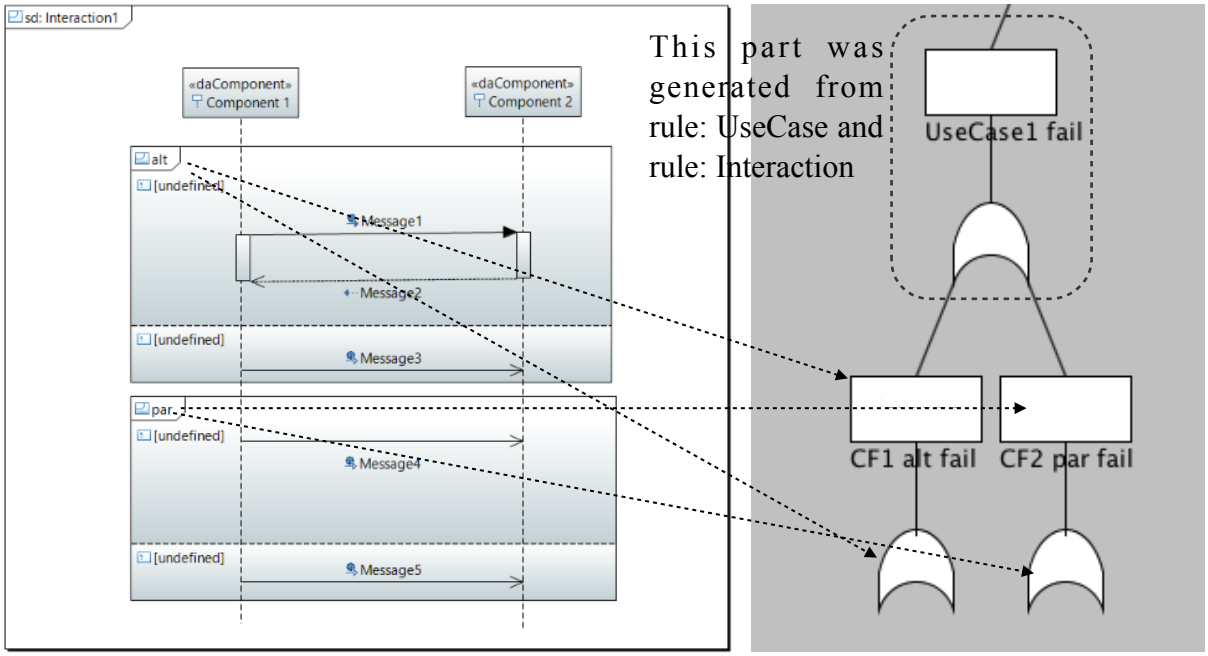


Figure 3.15: Transformation rule example - CombinedFragment

```
rule CombinedFragment{
from
    uml: MMa!CombinedFragment
to
    Event: MMb!IntermediateEvent(
           Title <- uml.name + uml.interactionOperator + 'fail',
           Child <- Gate
           ),
    Gate: MMb!Or_Gate(
           Chid <- uml.operand
           )
}
```

Code Fragment 3.4: Transformation rule - CombinedFragment

3.4.2.5. InteractionOperand

Within each CombinedFragment, as we can see from Figure 3.16 and CodeFragment 3.5, each InteractionOperand is transformed into an intermediate event followed by an Or_-Gate, for every of the operand, the guard name is set to the intermediate event's Info area.

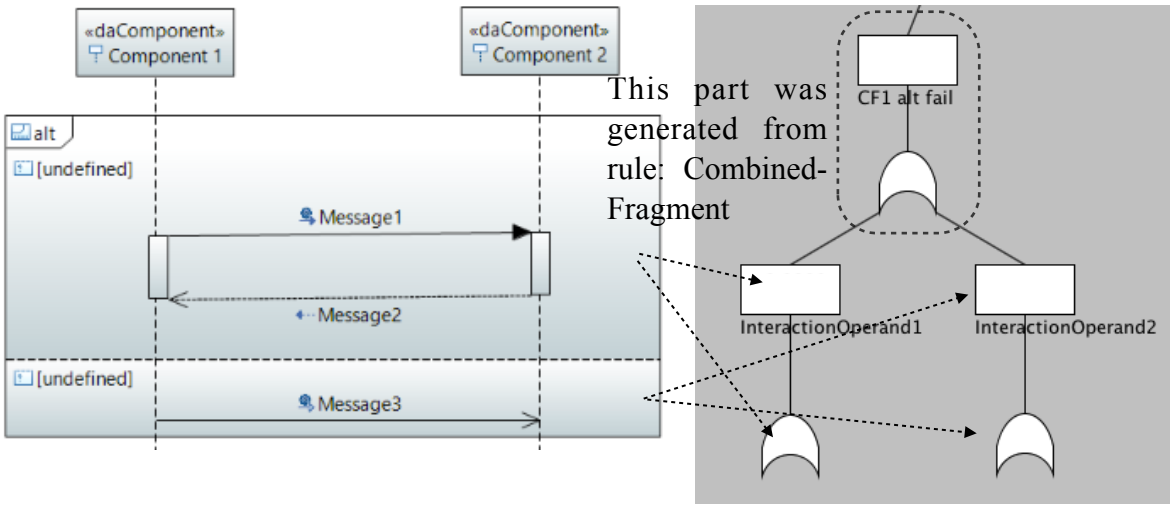


Figure 3.16: Transformation rule example - InteractionOperand

```
rule InteractionOperand{
from
    uml: MMa!InteractionOperand
to
    Event: MMb!IntermediateEvent(
           Title <- uml.name,
           Info <- uml.guard.name,
           Child <- Gate
           ),
    Gate: MMb!Or_Gate(
           Child <- uml.fragment
           )
}
```

Code Fragment 3.5: Transformation rule - InteractionOperand

3.4.2.6. BehaviorExecutionSpecification

BehaviorExecutionSpecification models the execution of components. As shown in Figure 3.17 and CodeFragment 3.6, each of the BehaviorExecutionSpecification is mapped to an intermediate event followed by an Or_Gate. This event in the fault tree model shows only the possibility of software failure during each execution period, without considering the possibility of hardware failure that will be taken into account in next level of the fault tree.

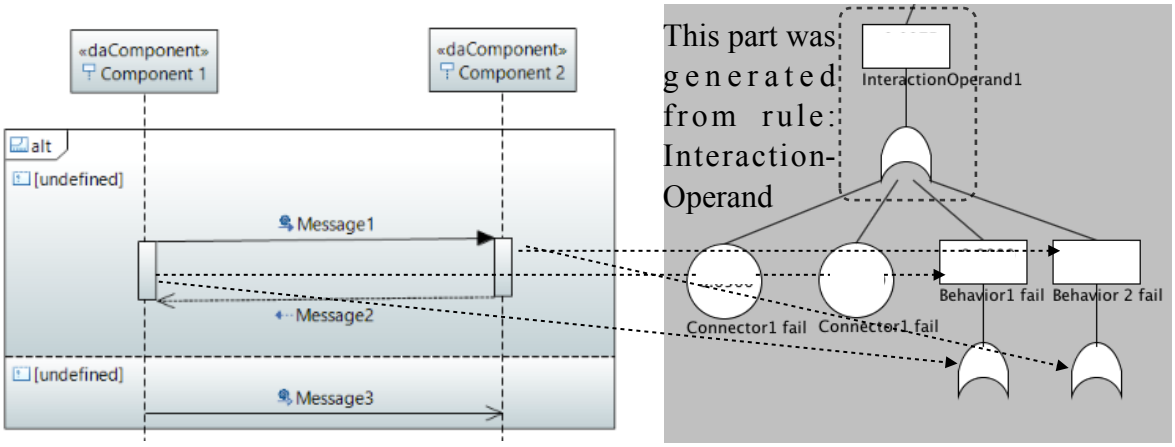


Figure 3.17: Transformation rule example - BehaviorExecutionSpecification

```
rule BehaviorExecutionSpecification{
from
    uml: MMa!BehaviorExecutionSpecification
to
    Event: MMb!IntermediateEvent(
           Title <- uml.name + 'fail',
           Child <- Gate
           ),
    Gate: MMb!Or_Gate(
           Child <- uml.getSoftwareComponent(),
           Child <- uml.getHardwareComponent(),
           Child <- uml.getRedundantComponent()
           )
}
```

Code Fragment 3.6: Transformation rule - BehaviorExecutionSpecification

---

3.4.2.7. SoftwareComponent

As we can see from Figure 3.18 and CodeFragment 3.7, we use a lazy rule to get the target software component for each BehaviorExecutionSpecification. Because each component contains many execution occurences, here we cannot use matched rules to get these components.

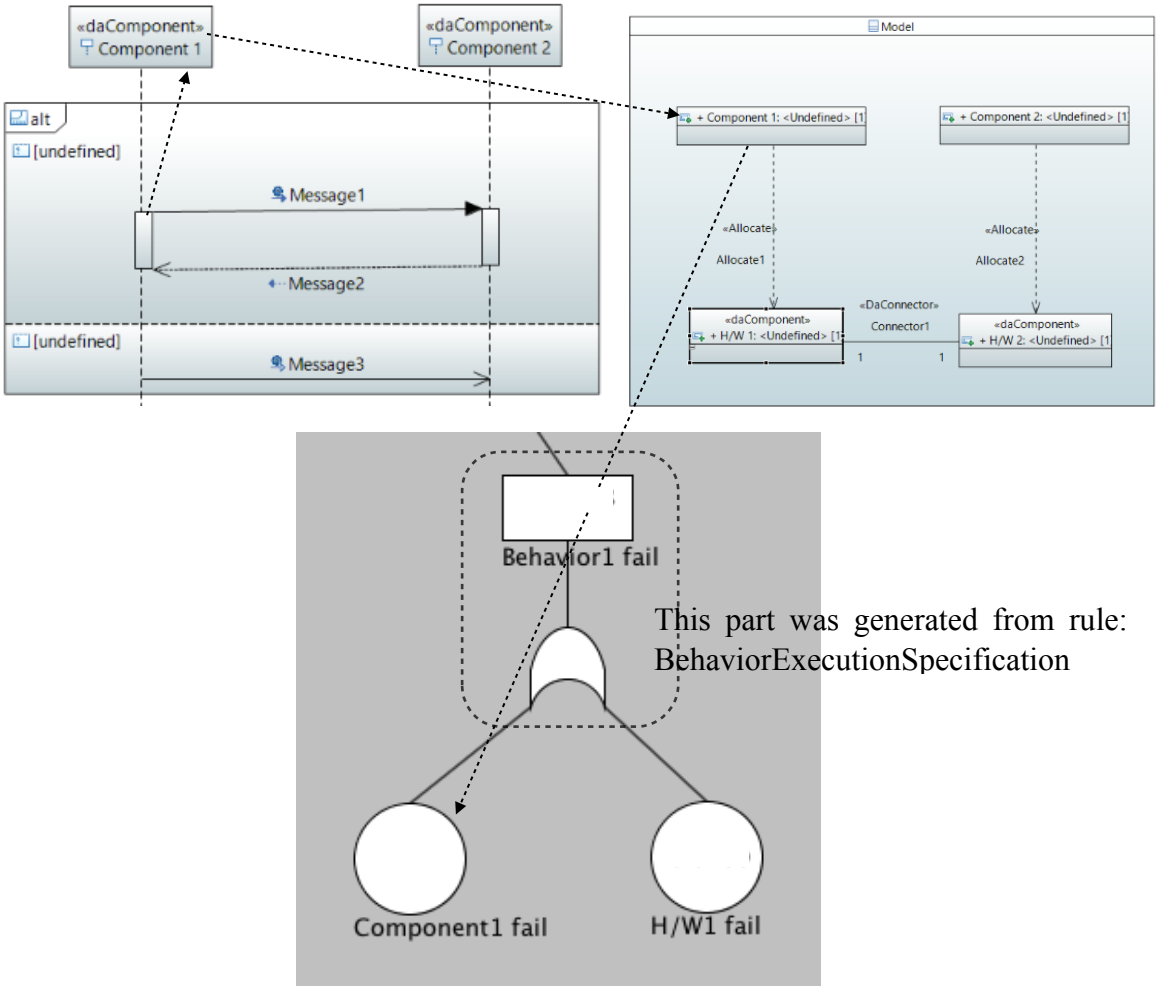


Figure 3.18: Transformation rule example - SoftwareComponent

```
lazy rule  getSoftwareComponent{
from
    uml: MMa!BehaviorExecutionSpecification
to
    FT: MMb!BasicEvent(
           Title <- uml.covered.represent.name + 'fail',
           Probability <- uml.getSoftwareProb()
    )
}
```

Code Fragment 3.7: Transformation rule - SoftwareComponent

A lazy rule is a declarative rule which is explicitly called. We use it here to get each BehaviorExecutionSpecification of a target software component.

For each BehaviorExecutionSpecification, we first get its covered lifeline in the sequence diagram, then get the lifeline's represented software component and transform it to a basic event in the fault tree model. The probability of the software component to fail is mapped to the Value of the basic event by calling a helper 'getSoftwareProb()'. All the helpers used in our transformation are described in detail in the next chapter.

3.4.2.8. HardwareComponent

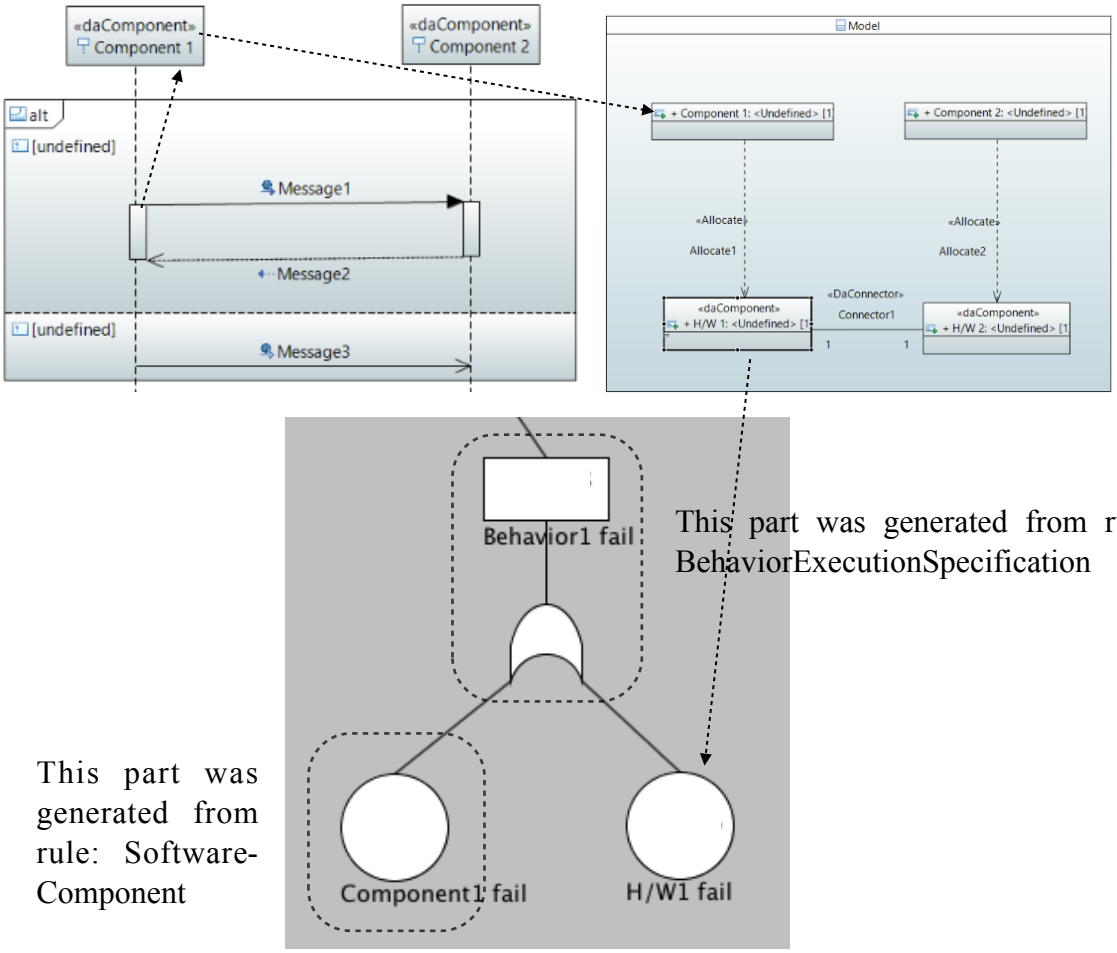


Figure 3.19: Transformation rule example - HardwareComponent

For the hardware components that do not have spares, we use a simple matched rule to transform them to basic events. As shown in Figure 3.19 and CodeFragment 3.8, in the se-

quence diagram, each BehaviorExecutionSpecification covers a lifeline, which represents a software component in the composite structure diagram. Each of the software components is allocated to a hardware component through the MARTE《allocation》stereotype. The getAllocate() helper is used for returning the allocated hardware component of a software component. Then the respective hardware component is transformed into a basic event with its failure probability mapped to the Value of the basic event by calling getHardwareComponent() helper.

```
lazy rule  getHardwareComponent{
from
    uml: MMa!BehaviorExecutionSpecification
           (uml.covered.represent.noSpare())
to
    FT: MMb!BasicEvent(
           Title <- uml.covered.represent.getAllocate().name +
'fail',
           Probability <- uml.getHardwareProb()
    )
}
```

Code Fragment 3.8: Transformation rule - HardwareComponent

3.4.3 Redundancy Rules

For the hardware components with spares, as we can see from Figure 3.20, CodeFragment 3.9 and CodeFragment 3.10, an intermediate event is generated from the allocated hardware component, followed by an And_Gate, because a hardware failure will happen only if both the allocated hardware and its spare would fail. The lazy rule getRedundantComponent shows that the generated And_Gate has two kinds of child elements. One of basic event which is input to the And_Gate is obtained by calling lazy rule getHardwareComponent. The other basic event is generated from the lazy rule getSpare, in which the helper getSubstitution() is used for getting the spare component of the target hardware component.

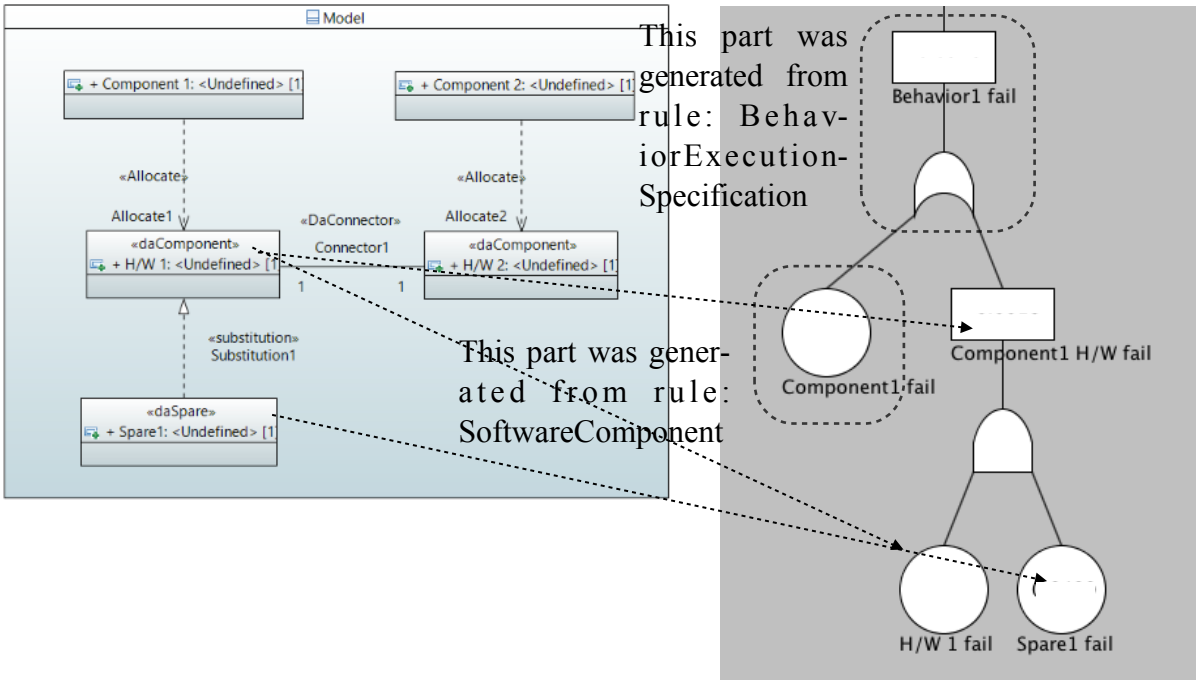


Figure 3.20: Transformation rule example - Redundancy

```
lazy rule  getRedundantComponent{
from
    uml: MMa!BehaviorExecutionSpecification
           (uml.covered.represent.getAllocate().hasSpare())
to
    Event: MMb!IntermediateEvent(
           Title <- uml.covered.represent.getAllocate().name  +
'fail',
           Child <- Gate,
    Gate: MMb!And_Gate(
           Child <- uml.getHardwareComponent(),
           Child <- uml.represent.getAllocate().getSpare()
    )
}
```

Code Fragment 3.9: Transformation rule - getRedundantComponent

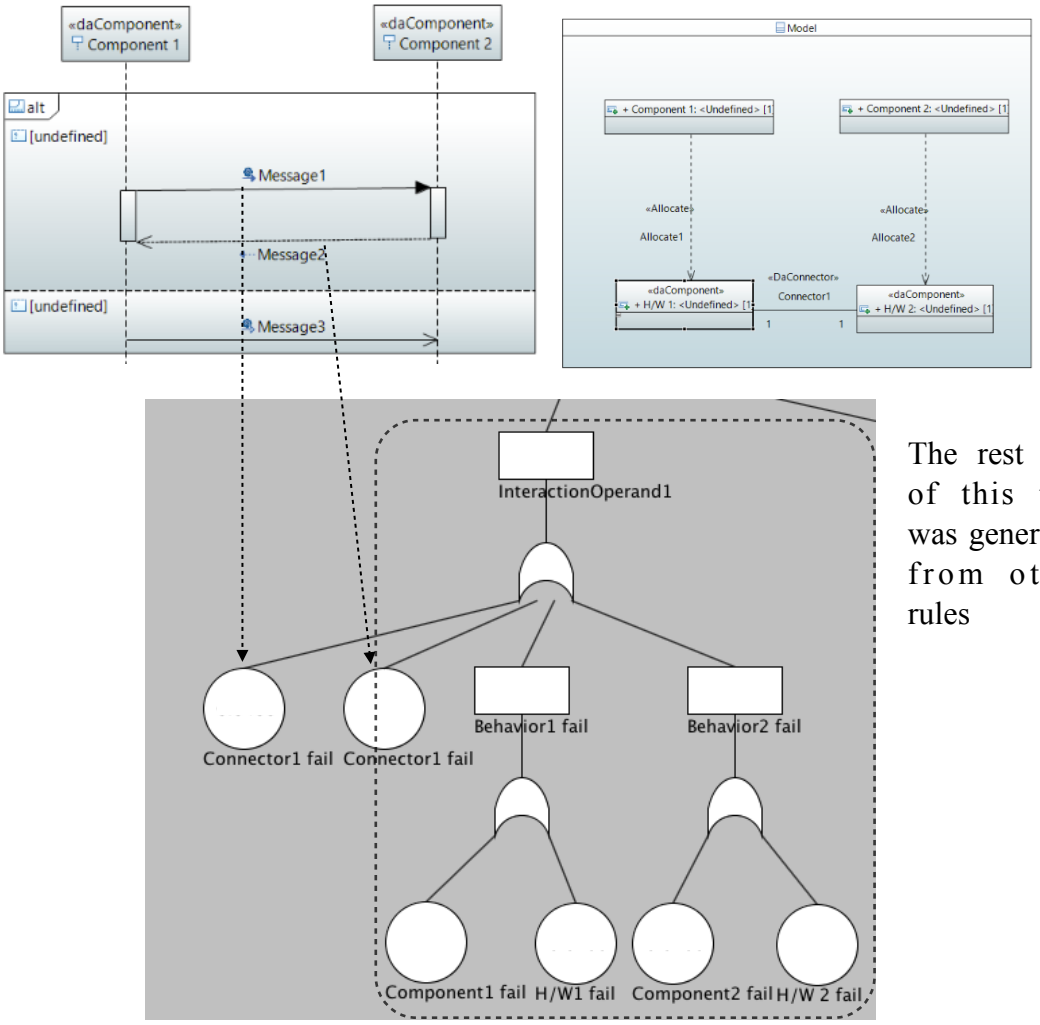
```
lazy rule  getSpare{
from
    uml: MMa!BehaviorExecutionSpecification

to
    FT: MMb!BasicEvent(
      Title <- uml.covered.represent.getAllocate().getSubstitution()

    )
}
```

Code Fragment 3.10: Transformation rule - getSpare

3.4.4 Customized Rules

For the users who may only want to consider components as the source of failure, the intermediate events generated from message may be omitted.. We proposed the choice for users to decide for which parts of the source model they would like to generate events in the fault tree model for further analysis. If the user does not assign a value to the failure probability of a connector, then we ignore the possibility of message failure due to that connector in our transformation. Consequently, the fault tree would not contain useless basic events from messages.

Figure 3.21 shows the mapping of messages, as we can see each message is mapped to a basic event of the fault tree. From CodeFragment 3.11 we can see that only those messages conveyed on connectors with a failure probability assigned by the user were generated. The helper getConnector() is used for getting the hardware connector associated with each message, the helper getConnectorProb() is to get the failure probability of that connector. Figure 3.22 shows the simplified version of our fault tree when the user did not assign any failure probability to connectors.



The rest part of this tree was generated from other rules

Figure 3.21: Transformation rule example- Message

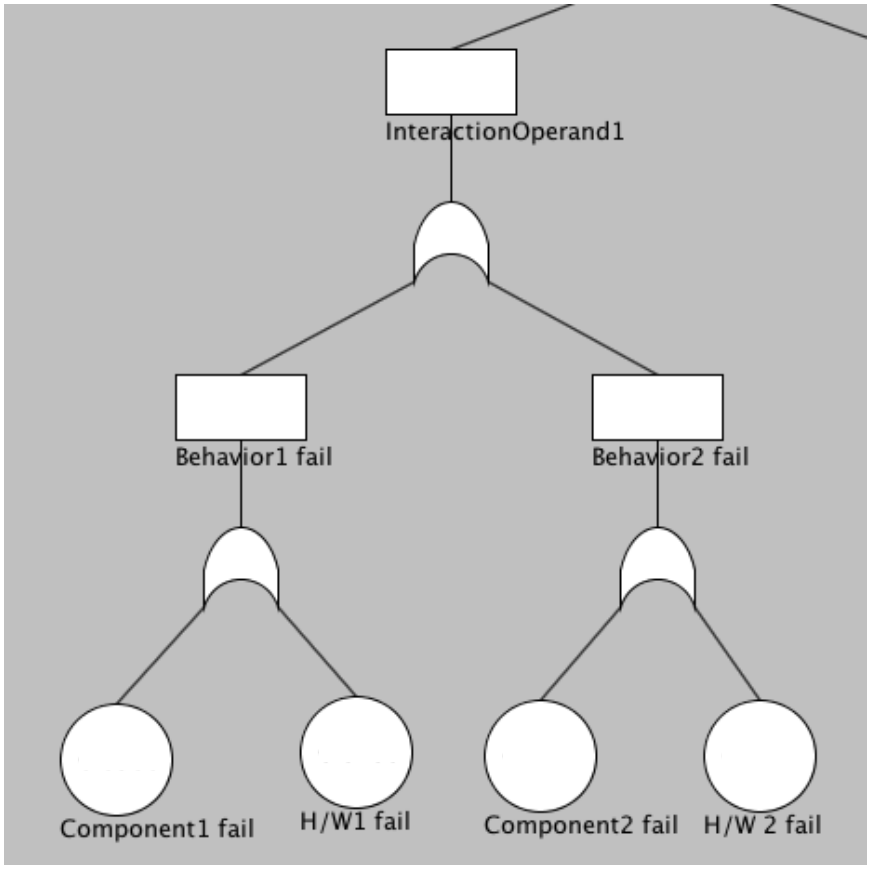


Figure 3.22: Transformation rule example - Message omitted

```
rule Message{
from
    uml: MMa!Message
    (uml.hasConnectorProb())

to
    FT: MMb!BasicEvent(
           Title <- uml.getConnector(),
           Probability <- uml.getConnectorProb()
    )
}
```

Code Fragment 3.11: Transformation rule - Message

CHAPTER 4. IMPLEMENTATION

4.1 Transformation Model

4.1.1. Architecture

The ATL implementation of the transformation in this study follows the regular structure of ATL transformations. The source, transformation, and target models each have their own separate metamodels, which are each based on a common meta-metamodel (ECORE).

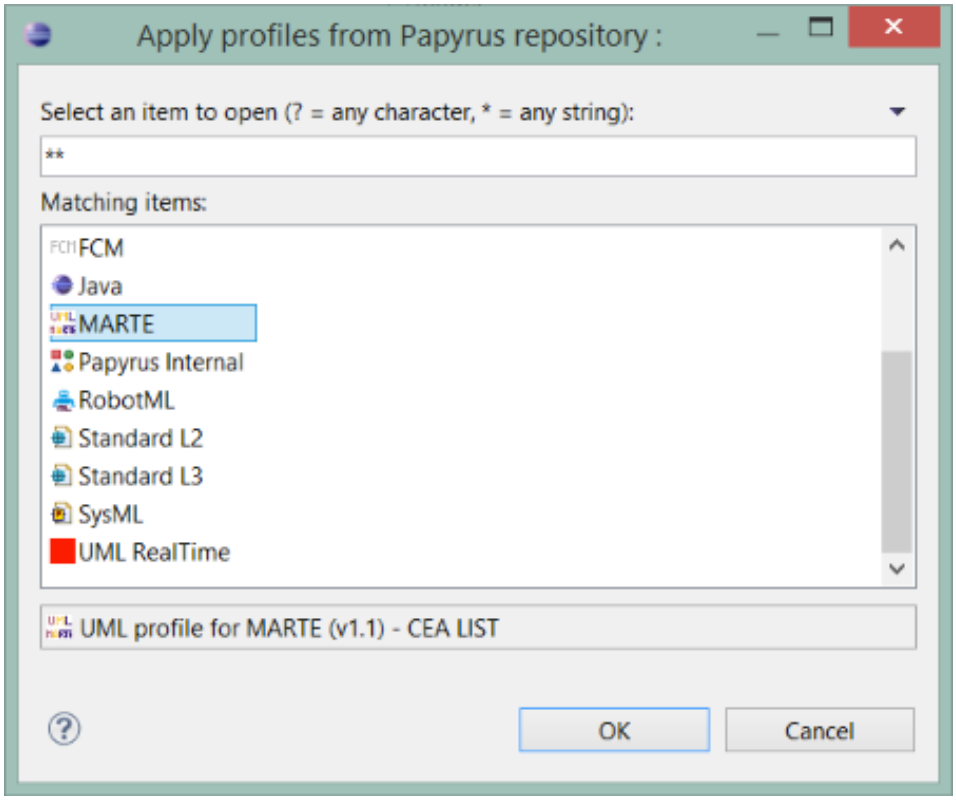


Figure 4.1 Apply MARTE Profile

The source metamodel we use is the UML2.4 EMF registered package which is included in the Eclipse environment, while the target Fault Tree metamodel is custom written using the ECORE tool. The transformation module, UML2FT, conforms to ATL and it accepts UML and profiles as source model, directly built with the Papyrus UML editor, and

produces the target Fault Tree. Papyrus originally supports the standard MARTE profile, so we can apply its stereotypes directly (as shown in Figure 4.1). The second profile used, DAM, is not standardized, so as a we use it as a plugin and could apply it during modeling (as shown in Figure 4.2).

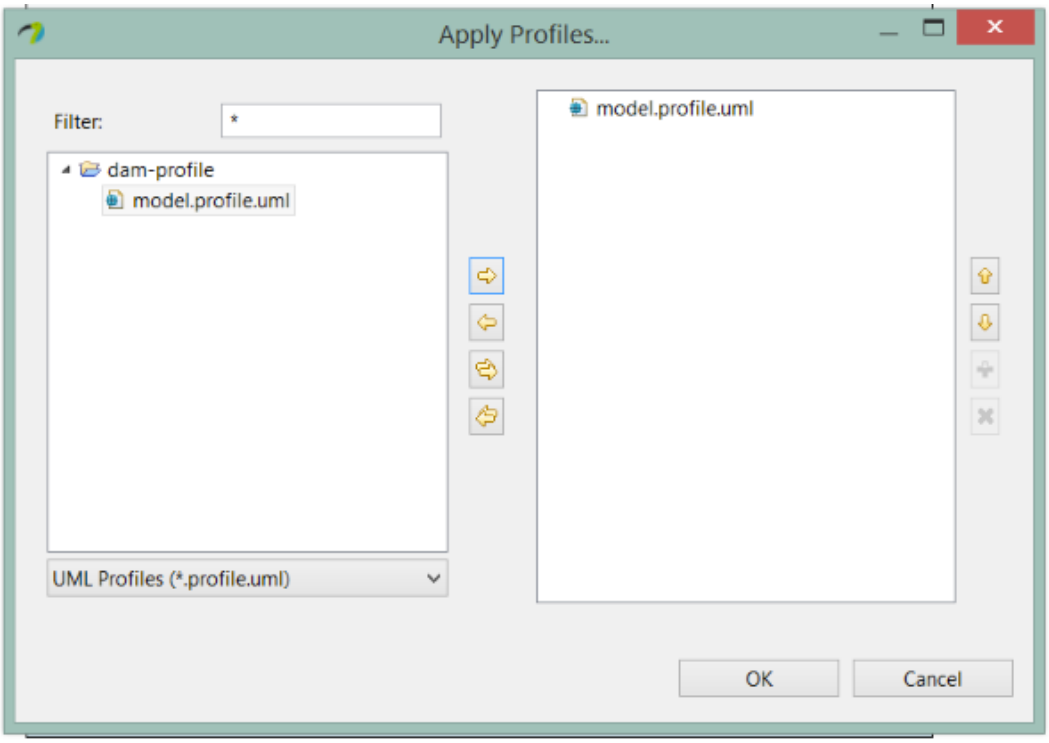


Figure 4.2 Apply DAM Profile

4.1.2. ATL implementation

Within the ATL module containing all the transformation rules, UML2FT, are 7 matched rules for all conditional mappings, and 4 lazy rules for all unconditional mappings. We also used 32 helpers that may be invoked by the transformation rules, 8 of which were particularly important. The description of specific ATL blocks is shown in the table 4.1 below.

	Name	Description
Matched Rules	Model	Transformed uml Model to IntermediateEvent and Or_Gate
	UseCase	Transformed UseCase to IntermediateEvent
	Interaction	Transformed Interaction to Or_Gate
	CombinedFragment	Transformed CombinedFragment to IntermediateEvent and Or_Gate
	InteractionOperand	Transformed InteractionOperand to Intermediate Event and Or_Gate
	BehaviorExecutionSpecification	Transformed BehaviorExecutionSpecification to IntermediateEvent and Or_Gate
	Message	Transformed Message to BasicEvent
Lazy Rules	getSoftwareComponent	For each BehaviorExecutionSpecification, find its covered lifeline, then find its represented SoftwareComponent, transformed it into BasicEvent
	getHardwareComponent	For those BehaviorExecutionSpecification running on HardwareComponent without spares, find its covered lifeline, then find its represented SoftwareComponent, then find the allocated HardwareComponent, transformed it into BasicEvent
	getRedundantComponent	For those BehaviorExecutionSpecification running on HardwareComponent with spares, transformed it to IntermediateEvent and And_Gate

Table 4.1: Main rules and helpers in UML2FT transformation

	Name	Description
Lazy Rules	getSpare	For those BehaviorExecutionSpecification running on HardwareComponent with spares, find its covered lifeline, then find its represented SoftwareComponent, then find its allocated HardwareComponent, then get its substitution. Transformed it into BasicEvent
Helpers	getSoftwareProb	For BehaviorExecutionSpecification, get its covered lifeline's represented SoftwareComponent's failure occurrence probability, returns the value of it
	getHardwareProb	For BehaviorExecutionSpecification, get its covered lifeline's represented SoftwareComponents allocated HardwareComponent's failure occurrence probability, returns the value of it
	getConnectorProb	For Message, get its associated Connector's failure occurrence probability, returns the value of it
	hasSpare	To determine if a BehaviorExecutionSpecification running on a HardwareComponent has spares. Returns true if it has substitution
	noSpare	To determine if a BehaviorExecutionSpecification running on a HardwareComponent do not have spare. Returns true if it does not have substitution

Table 4.1: Main rules and helpers in UML2FT transformation continued

	Name	Description
Helpers	getConnector	For Messages, get its associated connector's name by finding the two associated MessageEnd, then find their covered lifelines as well as represented SoftwareComponent, then get the two allocated HardwareComponents, then find the connector which its two connectorEnd matches the two HardwareComponents. Returns the name of the stereotyped Connector
	getSubstitution	For BehaviorExecutionSpecification running on HardwareComponents with spare, get its substitution. Each substitution element in UML has a client and a supplier. When the client is determined, which is the original allocated HardwareComponent, we can then trace its target supplier, returns the name of the spare component.
	hasConnectorProb	To determine if a message associated connector has a failure occurrence probability value, returns true if it has an input, otherwise the message would be omitted in our transformation process.

Table 4.1: Main rules and helpers in UML2FT transformation continued

```
helper context MMa!BehaviorExecutionSpecification
       def: getSoftwareProb(stereotype: String): OclAny =
      if self.covered.represent.getAppliedStereotype('DAM::'+
stereotype).oclIsUndefined()
      then OclUndefined
      else self.covered.represent.getValue('DaComponent','DaFail-
ure.occurrenceProb')
      endif
;
```

Code Fragment 4.1: Helper example_getSoftwareProb()

As an example of helpers, CodeFragment 4.1 shows the helper getSoftwareProb() which returns the represented SoftwareComponent's failure rate of the targeted BehaviorExecutionSpecification.

As we can see from CodeFragment 4.1, the getAppliedStereotype() call retrieves the stereotype with the specified qualified name that is applied to this element, or null if no such stereotype is applied. The getValue() call retrieves the value of the property with the specified name in the specified stereotype for this element. So that this helper returns the occurrenceProb value of the complex data type 'DaFailure' from stereotype 'DaComponent'.

In this implementation, matched rules are used for source elements such as model, use case, interactions, fragments, and certain applied MARTE/DAM stereotypes, whereas lazy rules are used for source elements that satisfy specific conditions generating target elements such as represented software, allocated hardware or spare components. The logic linking of each target elements with one another, based on the ordering of the source model's elements, is embedded within each rule, where the relationship is determined by the containment 'Child' association. This ensures that target elements are properly linked with each other when generated.

4.2 Transformation Process

The entire transformation process of a UML+MARTE/DAM Sequence Diagram, UseCase Diagram and Composite Structure Diagram to a Fault Tree Model contains 5 basic steps:

1) Using Ecore in Eclipse, define Fault Tree metamodel in UML class diagrams. This should be done only once, and then the file containing the target metamodel can be reused.

2) Using Papyrus in Eclipse, a source model containing UML Sequence Diagram, UseCase Diagram and Composite Structure Diagram with applied MARTE/DAM stereotypes is created.

3) In Eclipse, a transformation run configuration, which specifies both the UML file as the source model and the destination of the target model, is created.

4) The ATL transformation configuration is executed.

5) The generated target XML model should then be exported into a certain format which can be analyzed by the selected Fault Tree analysis tools. In our case, we use Fault-CAT which accepts its input in an XML file.

4.3 ATL Run Configuration

In order to run ATL transformation in Eclipse, a run configuration is required. In the dialog below, Figure 4.3, we can see that the configuration contains a name, ATL Module, which is the source of our ATL file, source and target metamodels. We directly used the UML2.4 metamodel from EMF registered packages, and the target Fault Tree metamodel are defined in Ecore. The source models are built with Papyrus in the Eclipse environment as well, and we use the .uml file as input to the transformation. The example of the run configuration of our test case 1 is shown as Figure 4.3 below.

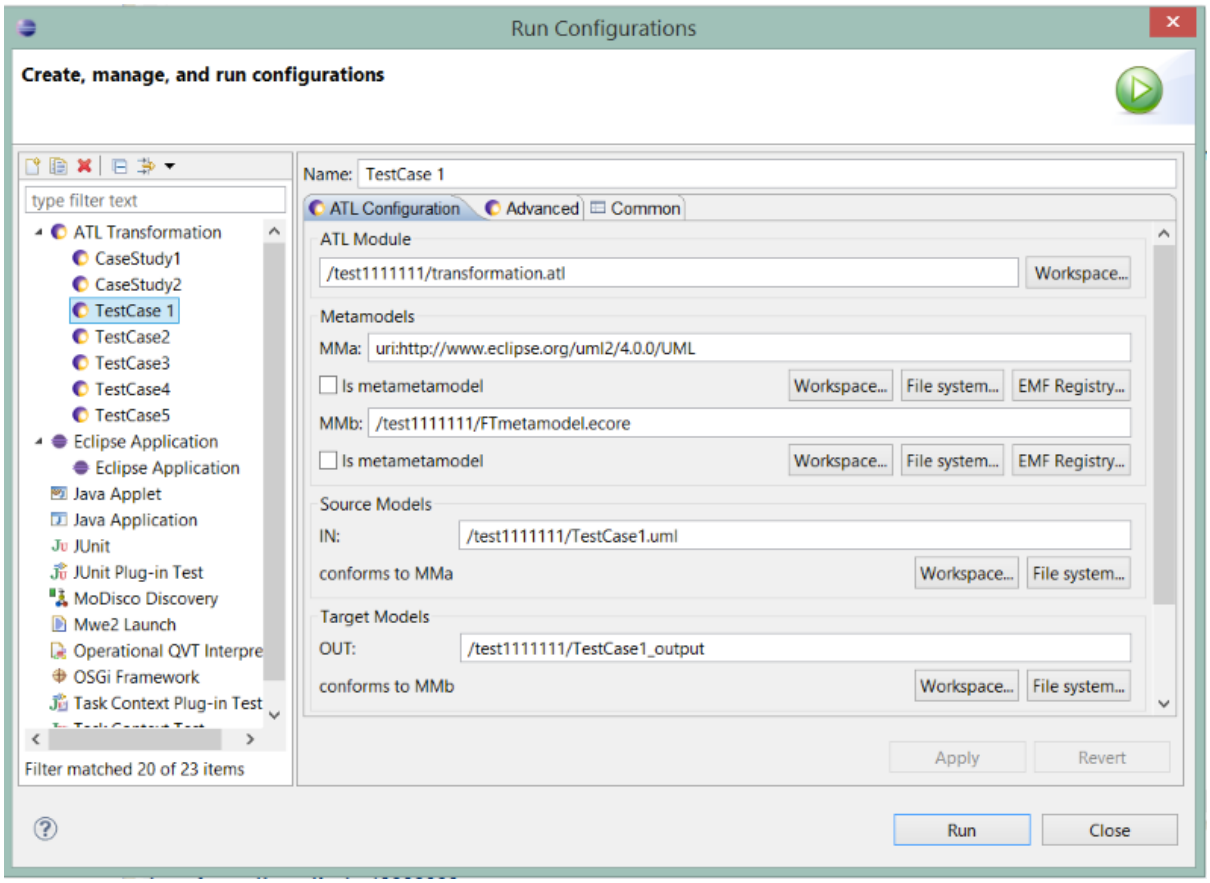


Figure 4.3 ATL Run Configurations

4.4 Output Processing

The target fault tree model generated by the ATL transformation is in XMI form, but the original output of our transformation is slightly different from the XML schema supported by the tool we use (FaultCAT). The difference is ….Since the differences are small, so a simple processing of the XML file is needed in order to achieve this transformation automatically. We did this by an additional Java program, which took the output of the ATL transformation as input, and did some text editing to remove some of the metamodel names, etc., so that the output of the Java program is totally legal for the analysis tool FaultCAT.

An example of the original output of our transformation is given in Code Fragment 4.2 below and the desired format for the input to FaultCAT is given in CodeFragment 4.3.. The

elements marked in grey in Code fragment 4.3 need to be removed.. An example piece of code to remove a specific string from the original output is shown as Code Fragment 4.4.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ftmetamodel="http://ftmetamodel/1.0">
  <ftmetamodel:IntermediateEvent Title="model"/>
  <ftmetamodel:Or_Gate>
    <Child xsi:type="ftmetamodel:IntermediateEvent" Title="UseCase1 fail"
Info="0.6">
      <Child xsi:type="ftmetamodel:Or_Gate">
        <Child xsi:type="ftmetamodel:IntermediateEvent" Title="Combined-
Fragment1 alt fail">
          <Child xsi:type="ftmetamodel:Or_Gate">
            <Child xsi:type="ftmetamodel:IntermediateEvent" Title="Inter-
actionOperand1fail">
              <Child xsi:type="ftmetamodel:Or_Gate">
                <Child xsi:type="ftmetamodel:IntermediateEvent"
Title="BehaviorExecSpec2fail">
                  <Child xsi:type="ftmetamodel:Or_Gate">
                    <Child xsi:type="ftmetamodel:BasicEvent" Title="Com-
ponent 2 fail" Value="0.04"/>
                    <Child xsi:type="ftmetamodel:BasicEvent" Title="H/W 2
fail" Value="0.03"/>
                  </Child>
                </Child>
```

Code Fragment 4.2: Example of original output of our transformation

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ftmetamodel="http://ftmetamodel/1.0">
<Fault-Tree>
      <Intermediate-Event><Title>Model fail</Title>
            <Or-Gate>
                  <Intermediate-Event><Title>UseCase1 fail</Title>
                  <Info/>0.6<Info/>
                  <Or-Gate>
                        <Intermediate-Event><Title>CombinedFragment1 alt
                              fail</Title>
                              <Or-Gate>
                              <Intermediate-Event><Title>Interaction-
Operand1                                    fail</Title>
                              <Or-Gate>
                              <Intermediate-Event><Title>BehaviorExecu-
tion                                 Spec2 fail</Title>
                              <Or-Gate>
                                    <Basic-Event>
                                    <Title>Component2 fail</Title>
                                    <Value>0.04</Value>
                                    </Basic-Event>
                                    <Basic-Event>
                                    <Title>H/W2 fail</Title>
                                    <Value>0.03</Value>
                                    </Basic-Event>
                              </Or-Gate>
                              </Intermediate-Event>
```

Code Fragment 4.3: Example of desired input of FaultCAT

CHAPTER 5. VERIFICATION AND CASE STUDY

5.1 Verification

5.1.1 Overview

This section describes the verification of our ATL transformation from UML model to Fault Tree model with several test cases. Each test case was designed to cover one or more specific conditions. These tests will cover all the matched rules and lazy rules that we have introduced in chapter 3. During the execution, all the helpers have also been called at least ones. Table 5.1 below shows the coverage of each test case.

Test Case 1	simple message and execution
Test Case 2	use case
Test Case 3	combined fragment
Test Case 4	nested combined fragment
Test Case 5	redundant components
Test Case 6	omitted message

Table 5.1: Test Cases

5.1.2 Test Case 1: Message and Execution

In this test case we meant to test simple transformations of messages and execution occurrence, to see if these most basic elements have been correctly transformed. Figure 5.1 shows the sequence diagram of our test model, which contains two messages and two execution occurrences, and the lifelines represent components in Figure 5.2 CompositeStructureDiagram. Both software and hardware components are applied with stereotype 《DaComponent》 in order to set their failure occurrence probability. Software components are allocated

to hardware components by using the stereotype 《Allocate》 of MARTE profile. The connector between hardware components is stereotyped with 《DaConnector》 with given failure occurrence probability. In Figure 5.3 the UseCase diagrams shows there is only one use cases in this model, in which UseCase1 is mapped to the Interaction in this test case. The Fault Tree model generated from the above UML model is Figure 5.4.

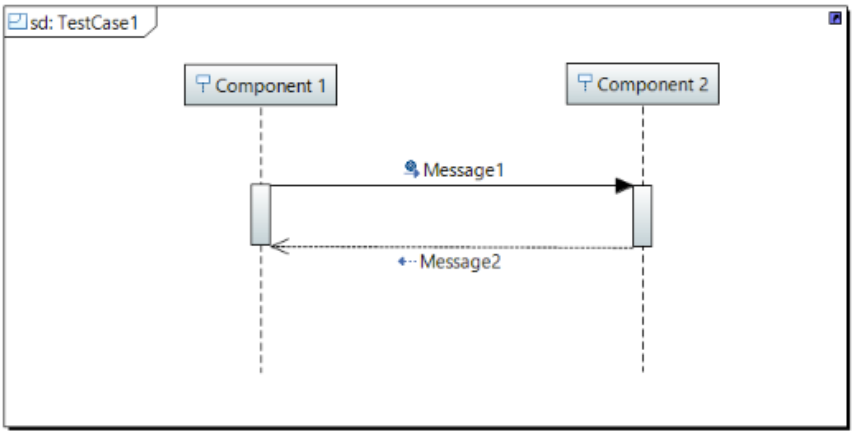


Figure 5.1 Sequence Diagram of TestCase1

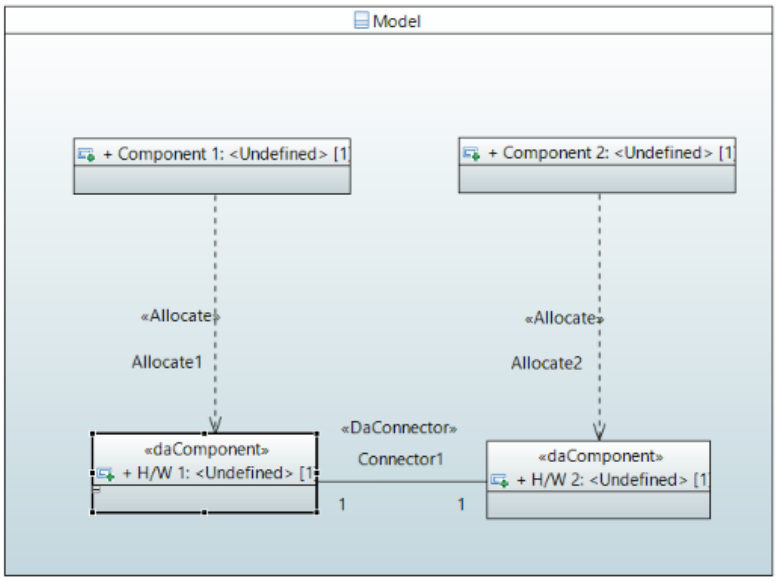


Figure 5.2 Composite Structure Diagram of TestCase1

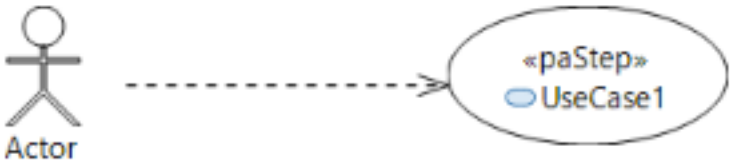


Figure 5.3 Use Case Diagram of TestCase1

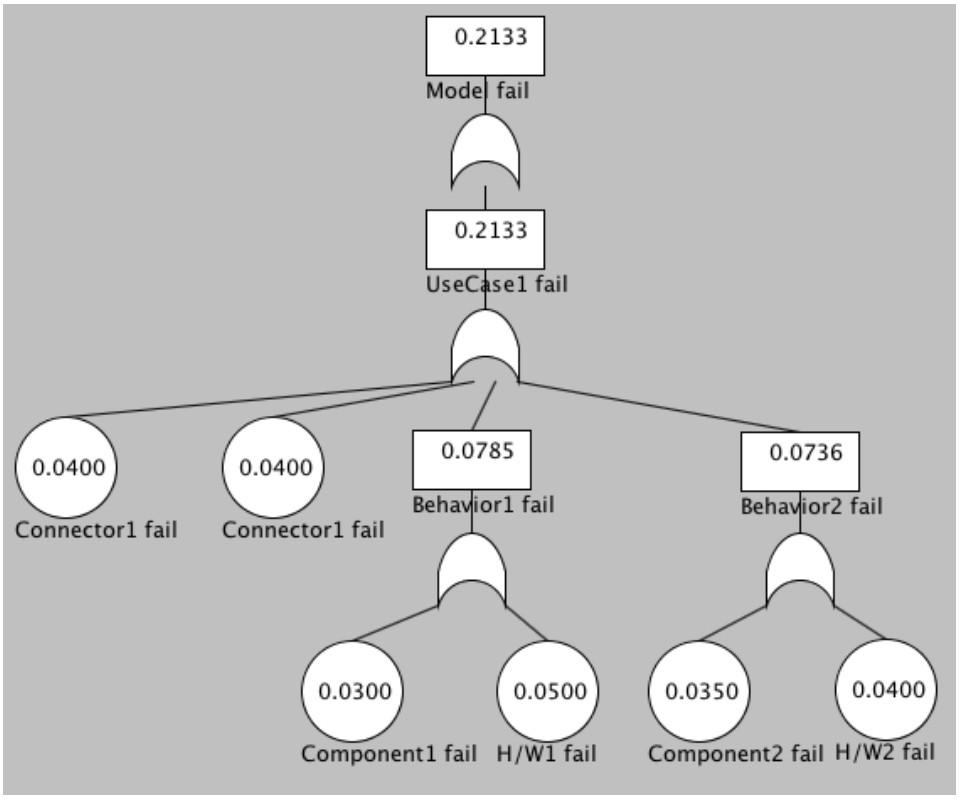


Figure 5.4 Generated fault tree model of test case 1

5.1.3 Test Case 2: Use Case

In this test case we added one more use case to our model, thus, two uses cases are mapped to two interactions, as in Figure 5.5 below. To simplify the test case, the two interactions (sequence diagram) are the same as Figure 5.1, but with different execution probabilities (0.6 and 0.4). The Composite Structure is the same as in Figure 5.2.

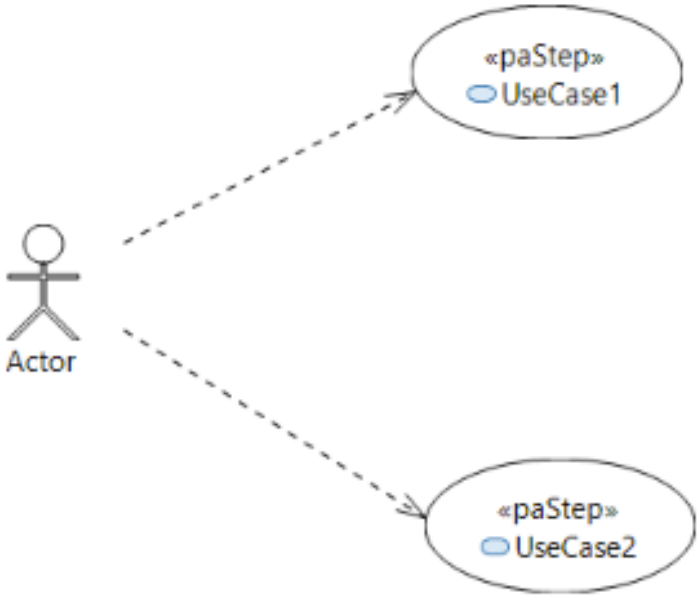


Figure 5.5 Use case diagram for test case 2

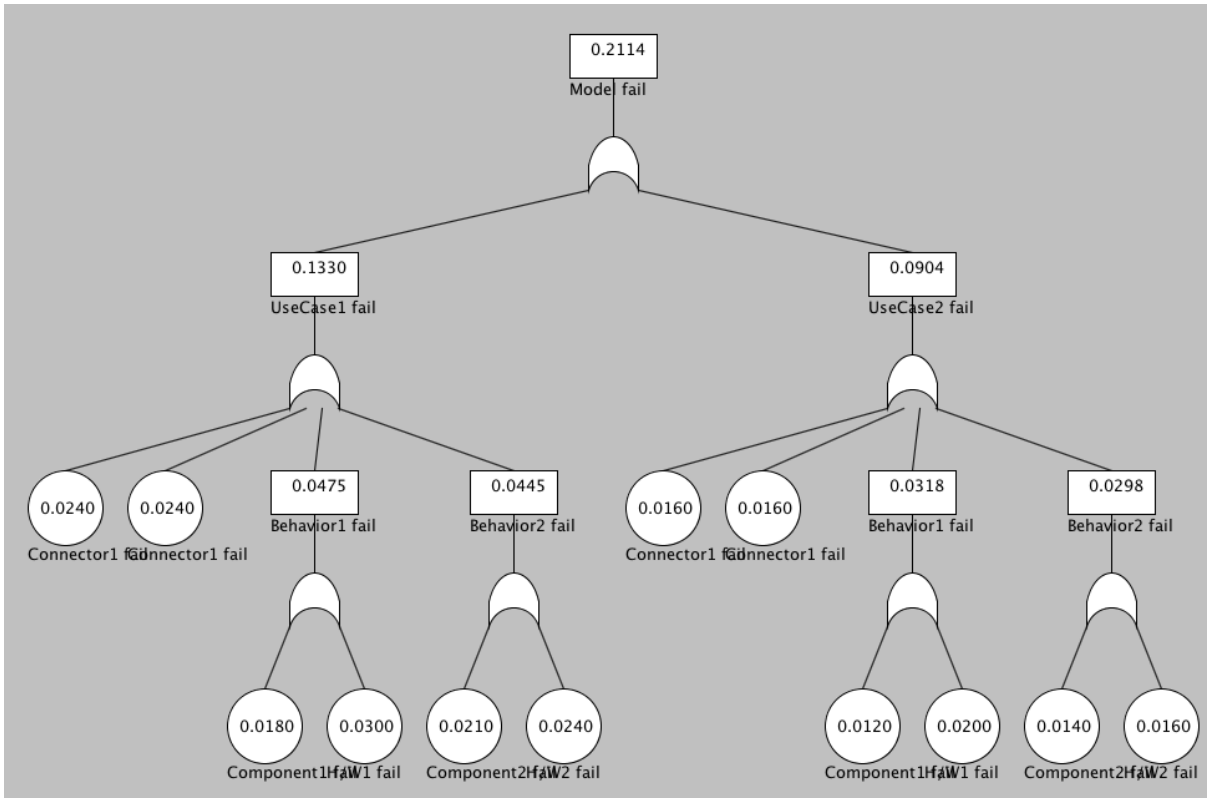


Figure 5.6 Generated fault tree for test case 2

The generated fault tree model of test case 2 is shown in Figure 5.6. As two use cases have their own execution probability, in the basic event under each use case, we multiplied the probability from each use case's Info block by the original failure occurrence probability. Since the tool FaultCAT does not support conditional events yet, this part of the computation is done manually at the current stage.

5.1.4 Test Case 3: Combined Fragment

This test case is designed to test combined fragments in our transformation. The use case and composite structure are the same as in test case 1. Figure 5.7 and Figure 5.8 shows the sequence diagram we used in this test case and the generated fault tree model, respectively.

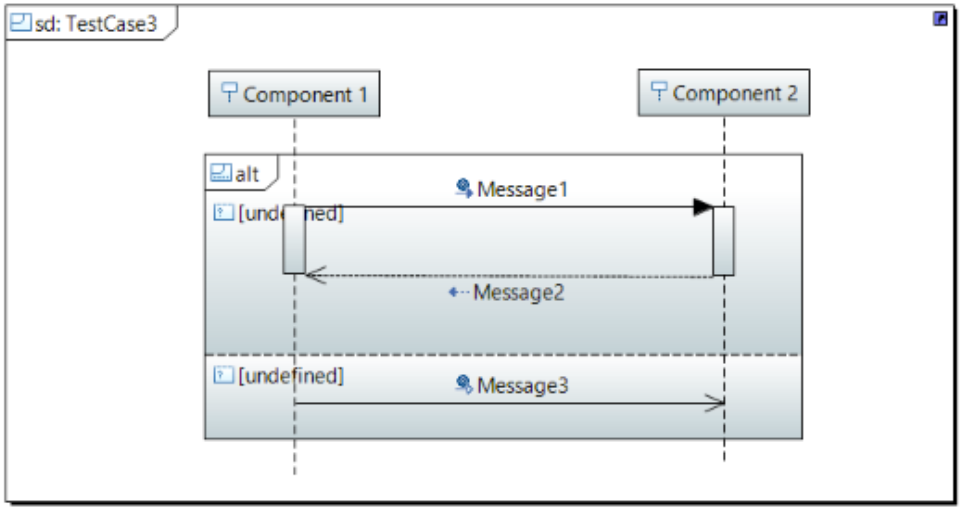


Figure 5.7 Sequence diagram of test case 3

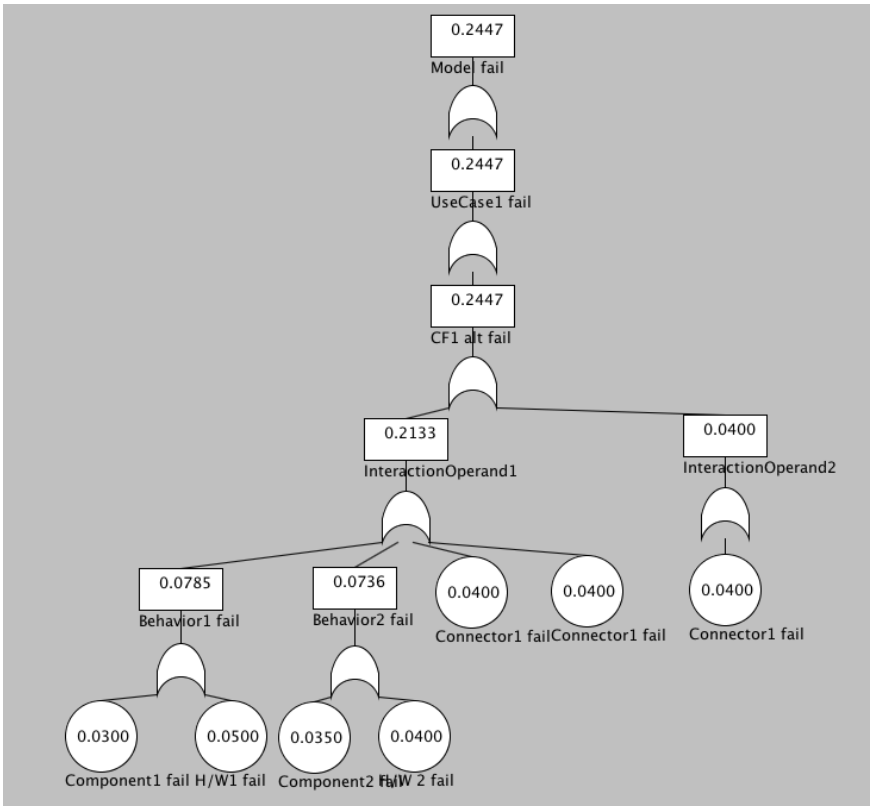


Figure 5.8 Generated fault tree model of test case 3

5.1.5 Test Case 4: Nested Combined Fragment

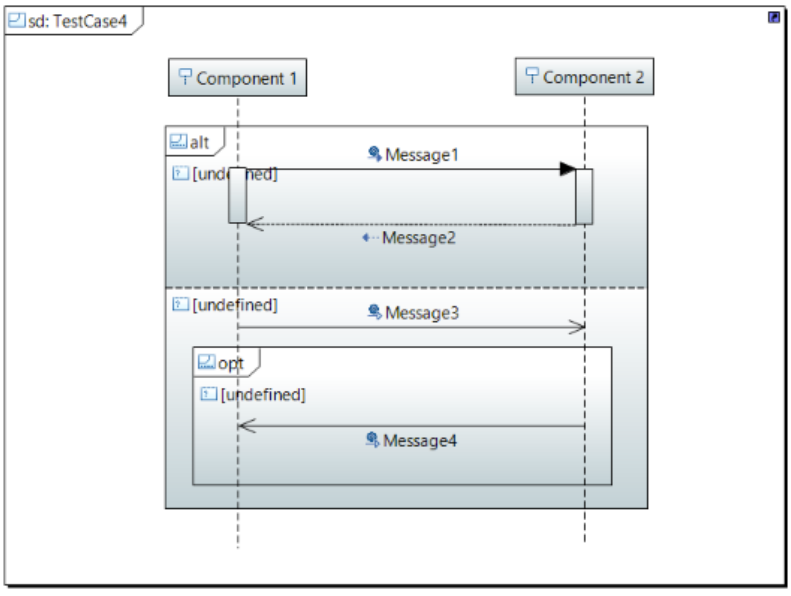


Figure 5.9 Sequence diagram of test case 4

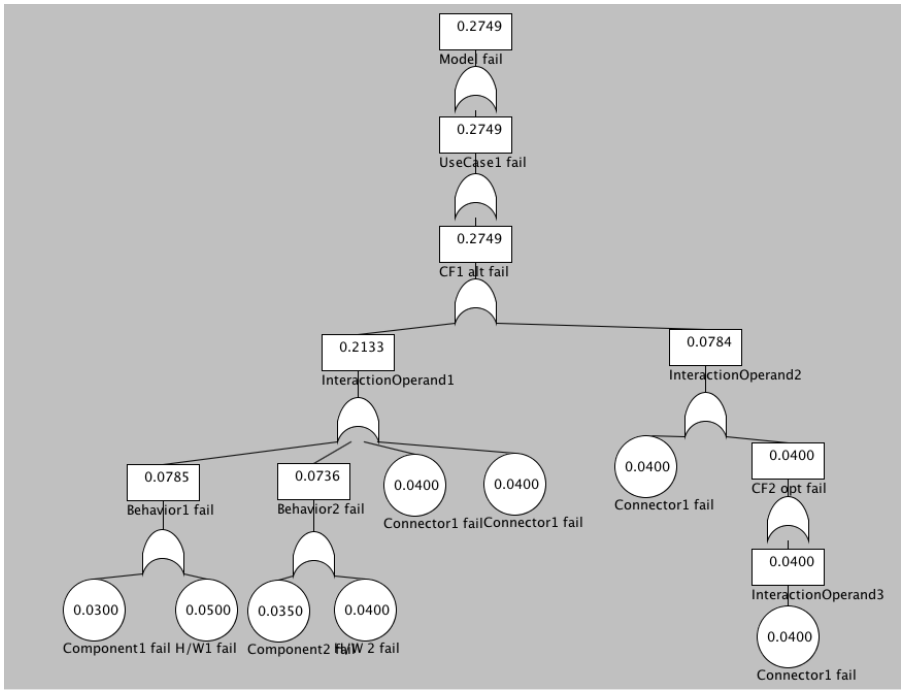


Figure 5.10 Generated fault tree model of test case 4

As for nested combined fragments, the use case and composite structure are the same as in test case 1, while the sequence diagram is shown as Figure 5.9. The generated fault tree models is shown in Figure 5.10.

5.1.6 Test Case 5: Redundant Component

To test the mapping of redundant hardware components, we reuse the use case and sequence diagram from test case 1, and changed the composite structure diagram to Figure 5.11 below.

As we can see from Figure 5.11, we have added a spare component 'Spare1' to the model, which is the substitution of HW1, so that the generated fault tree model turned out as in Figure 5.12. As we can see ,an And_Gate is generated, showing that the above output event occurs if all of the input lower level events occur (both the H/W and Spare1 fail).

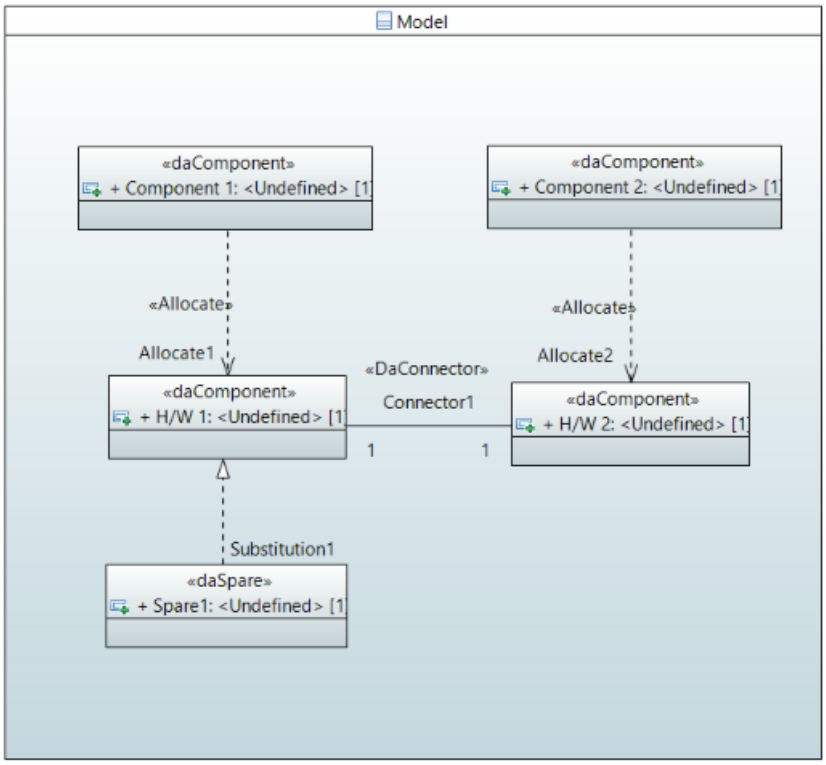


Figure 5.11 Composite structure diagram of test case 5

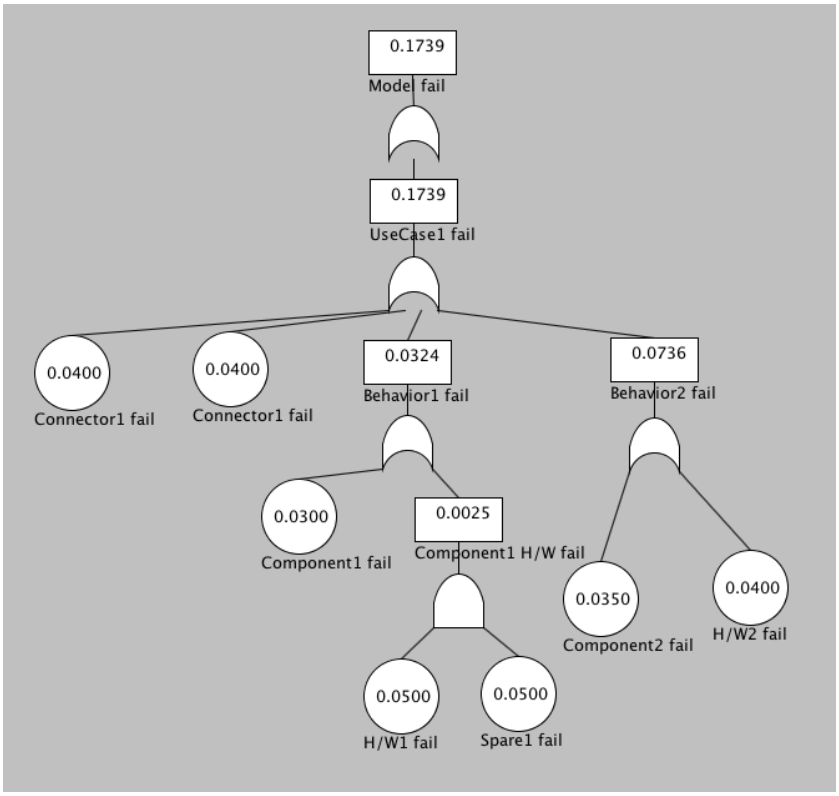


Figure 5.12 Generated fault tree model of test case 5

5.1.7 Test Case 6: Omitted Message

If the user choses not to give a failure probability for the connectors, it is assumed that only component may fail. In this case the generation of the basic events corresponding to message failure can be omitted,so that there the fault tree would be simpler. The model we use in this test case is the same as for test case 5, but without inputing the occurrenceProb value under the stereotype of DaConnector. Figure 5.13 shows the generated fault tree of this test case.

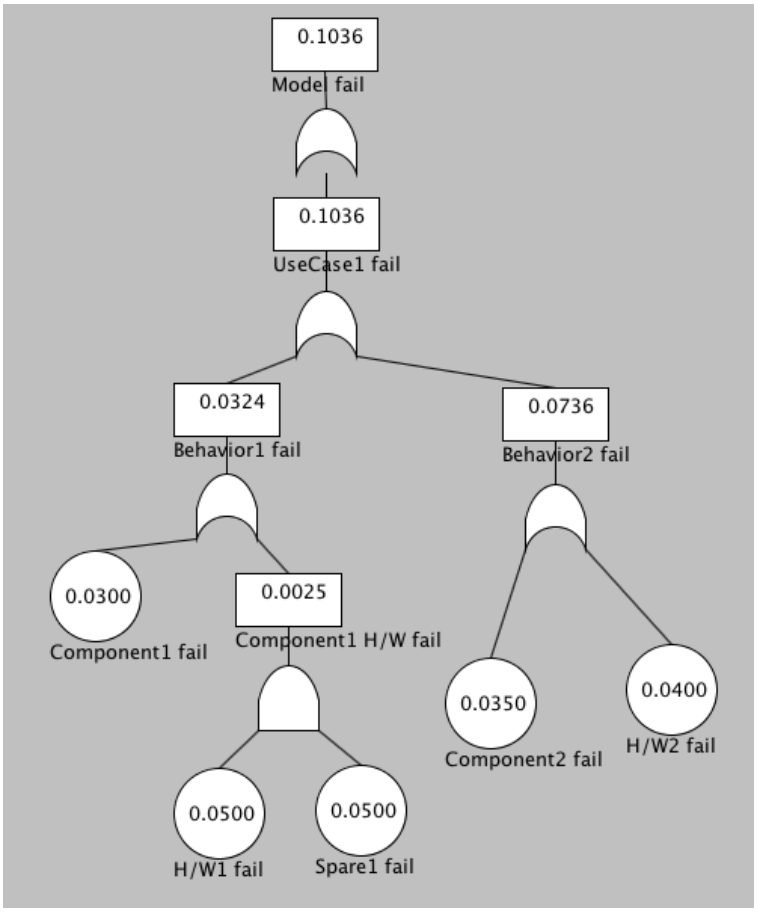


Figure 5.13 Generated fault tree model of test case 6

5.1.8 Test Case 7: Variable number of children

This test case is to show that our transformation is able to handle the generation of a variable number of children nodes to the fault tree since most of the test cases shown before

generate only one or two children of the same type. Figure 5.14 shows the UseCase diagram of this test case, which contains three use cases, each of them is mapped to an interaction (sequence diagram), as shown in Figure 5.16, 5.17 and 5.18 respectively. Figure 5.15 is the Composite Structure diagram of this test case.

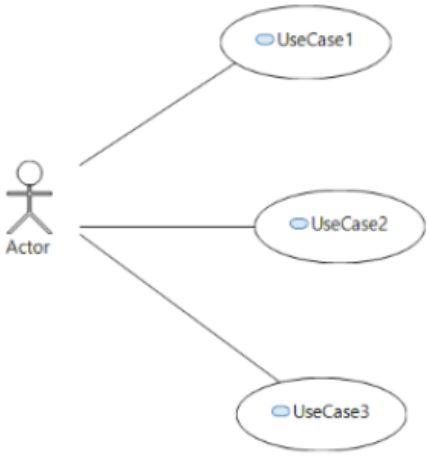


Figure 5.14 UseCase digram of Test Case 7

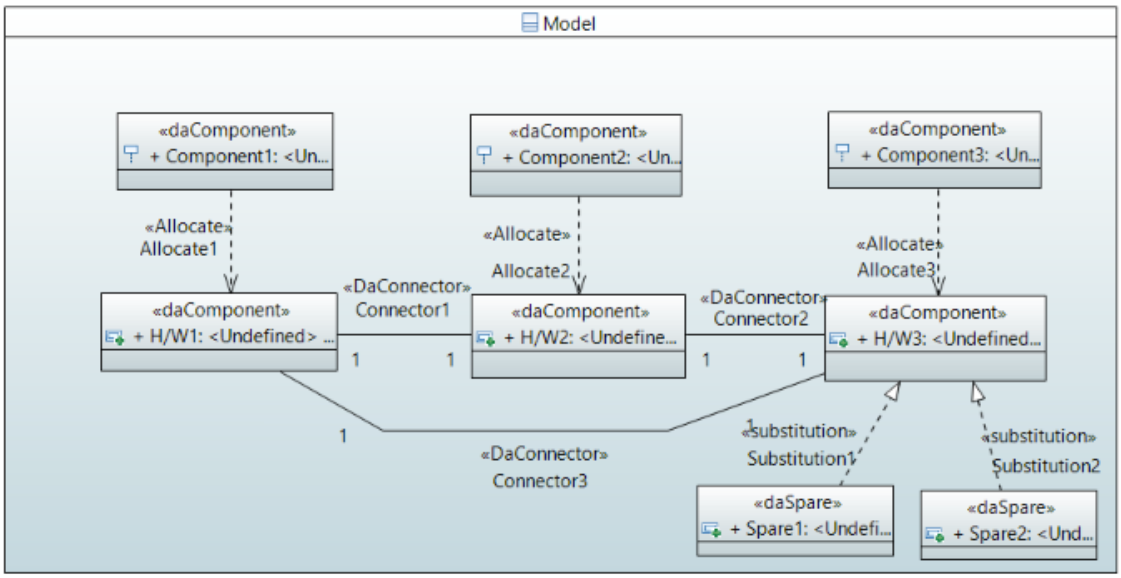


Figure 5.15 Composite Structure digram of Test Case 7

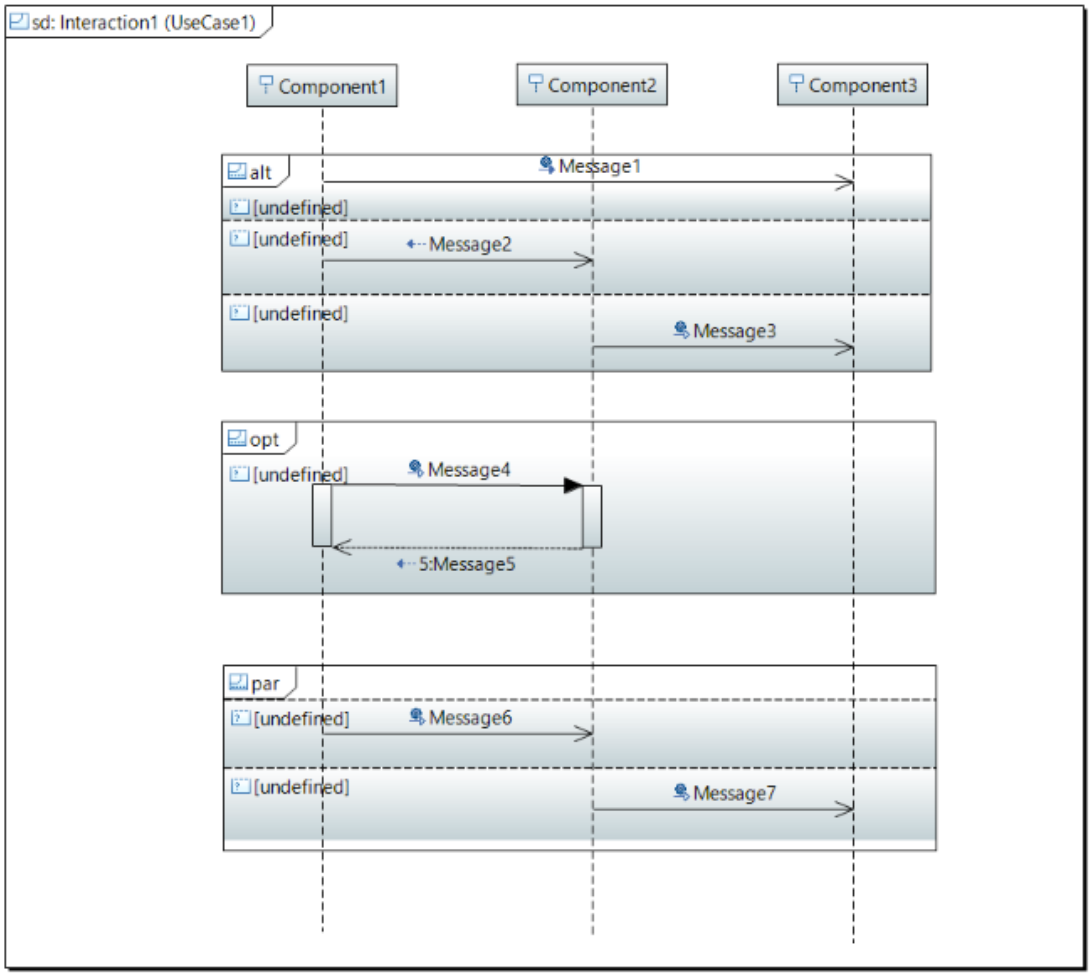


Figure 5.16 Sequence digram of Test Case 7, UseCase1

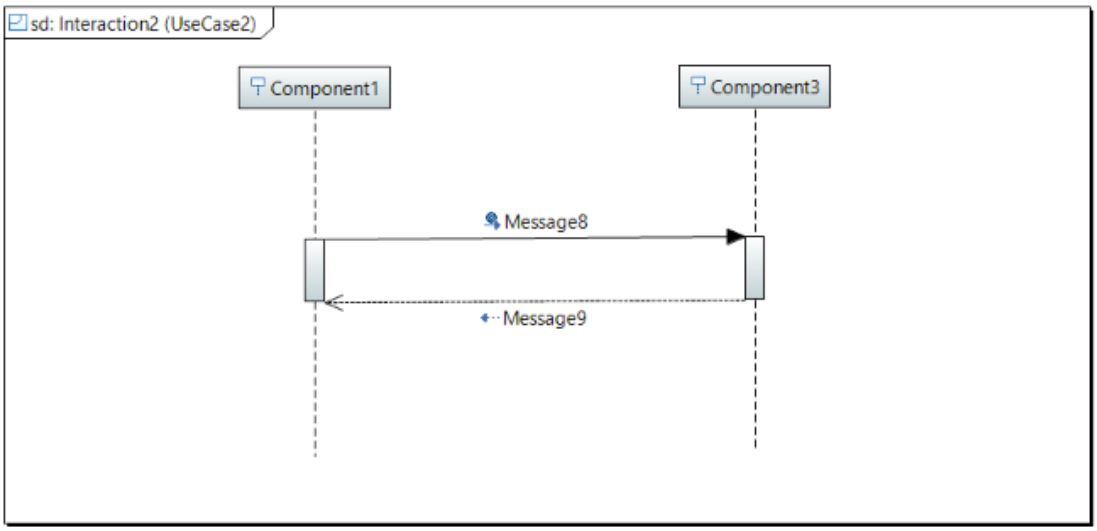


Figure 5.17 Sequence digram of Test Case 7, UseCase2

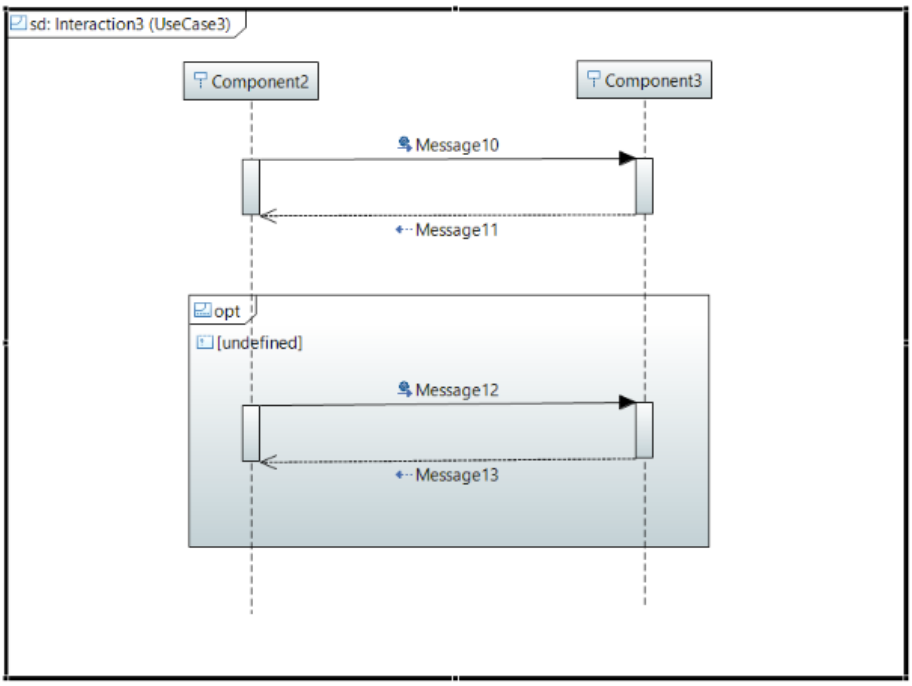


Figure 5.18 Sequence digram of Test Case 7, UseCase3

As we can see from these diagrams above, as in Figure 5.15, H/W3 has two spares. Figure 5.14 shows that the model has three use cases. Figure 5.16 shows that UseCase 1 has three combined fragments, the first of the combined fragment has three interaction operands. From Figure 5.17 we can see that Component3, which is allocated to H/W3 (with more than one spare), has an execution occurrence, so that the multiple redundant elements are generated under this situation. Figure 5.18 was to test that the elements(messages and execution occurrences), either within or outside a combined fragment will appear in their proper layer. We can see in Figure 5.18, a synchronous call and its return message with two execution occurrences appears twice in the sequence digram, one outside while another was inside a combined fragment. Figure 5.19 is the fault tree generated from the diagrams described above, in which we can see that, compared to the previous six test cases which shows the generation of only one or two children of the same type, our transformation method is also able to deal with

a variable number of children elements (UseCase, CombinedFragment, InteractionOperand, Messages, ExecutionOccurrence, and Spares), and it ensures every element appears in their correct layer of the tree.

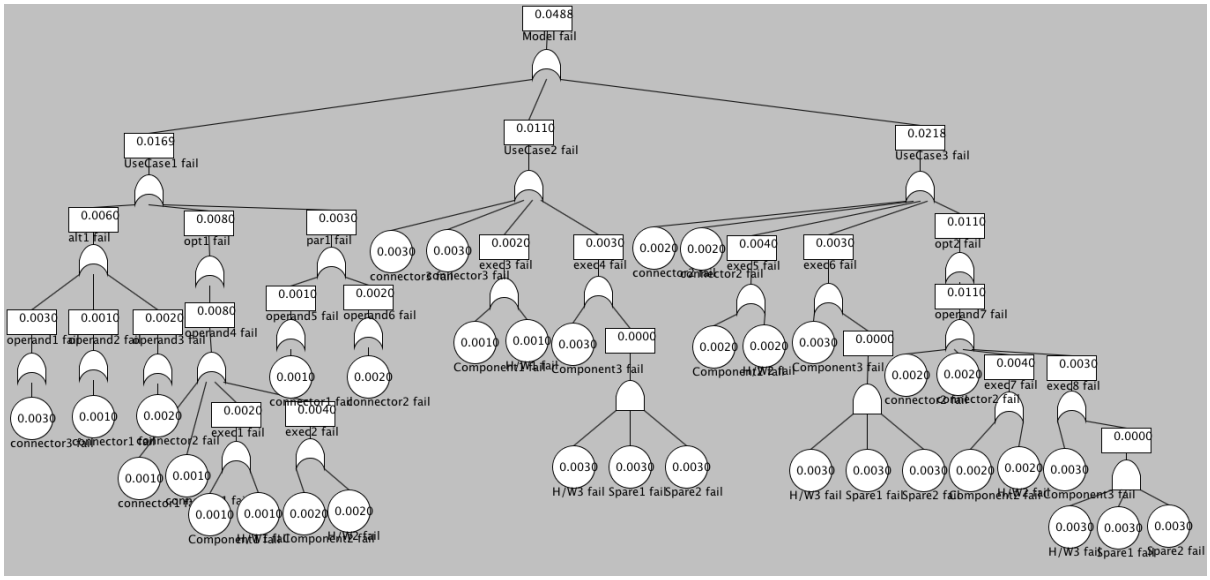


Figure 5.19 Generated Fault Tree of test case 7

To indicate, this test case, as the most complicated one in our work, takes about 6 to 7 seconds to execute by ATL. The test was done by a 2.4GHz Intel Core i7 Processor on a Macbook Pro.

5.2 Case Study 1

In this case study we took an example from the work of D'Ambrogio et al.[D'AMBROGIO02], in which we reused a simplified version of their sequence diagram (Figure 5.20) and rebuilt their deployment diagram as a composite structure diagram (Figure 5.21) to show the system architecture. This example contains only one use case so that the use case diagram is the same as in Figure 5.3 described above. Because their paper did not give the quantitative information for each component's failure occurrence probability, we input these values based on our own assumption, as shown in Table 5.2 below. Figure 5.22 shows the

fault tree generated based on D'Ambrogio et al. original approach, while Figure 5.23 shows the generated fault tree based on the example system by our work.

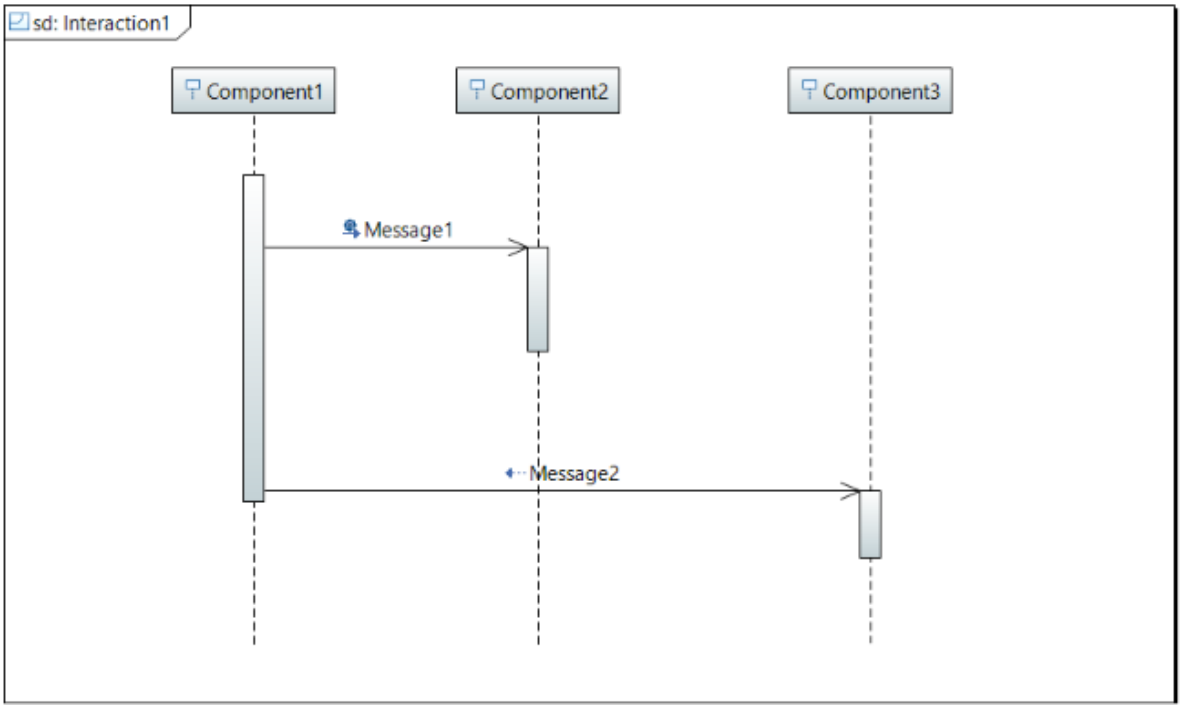


Figure 5.20 Simplified sequence diagram of Case study 1

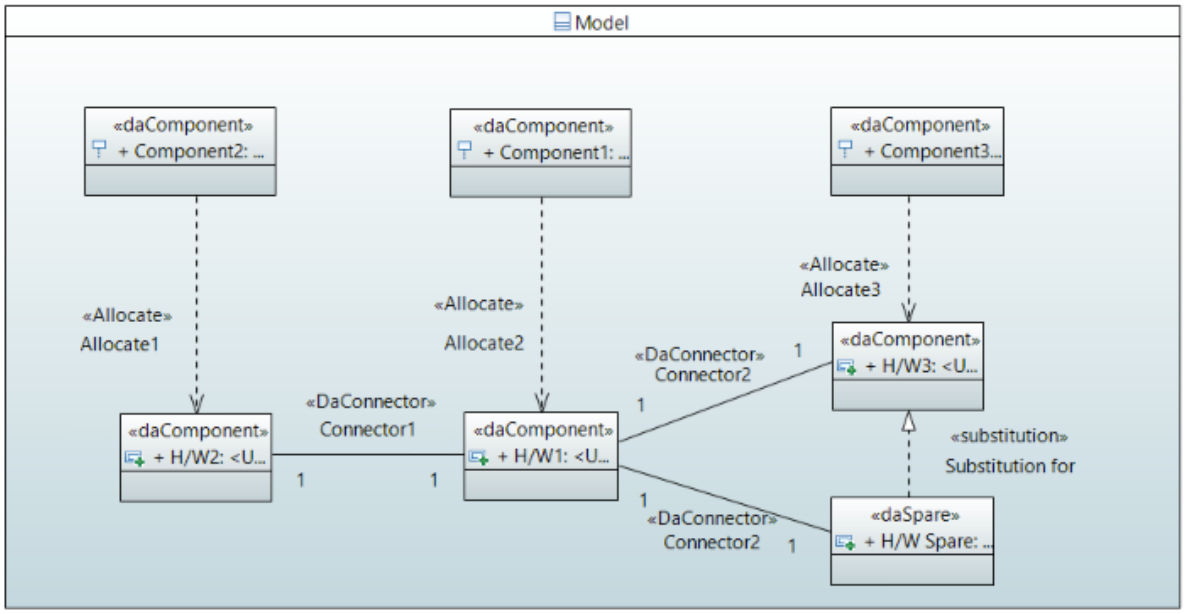


Figure 5.21 Composite Structure diagram of Case study 1

Component Name	Failure Occurrence Probability
《DaComponent》 :Component1	0.003
《DaComponent》 :Component2	0.004
《DaComponent》 :Component3	0.0035
《DaComponent》 :H/W2	0.0045
《DaComponent》 :H/W3	0.005
《DaSpare》 :H/W Spare	0.005
《DaComponent》 :H/W1	0.0033
《DaConnector》 :Connector1	0.002
《DaConnector》 :Connector2	0.003

Table 5.2: Input Failure Occurrence Probability of Case Study 1

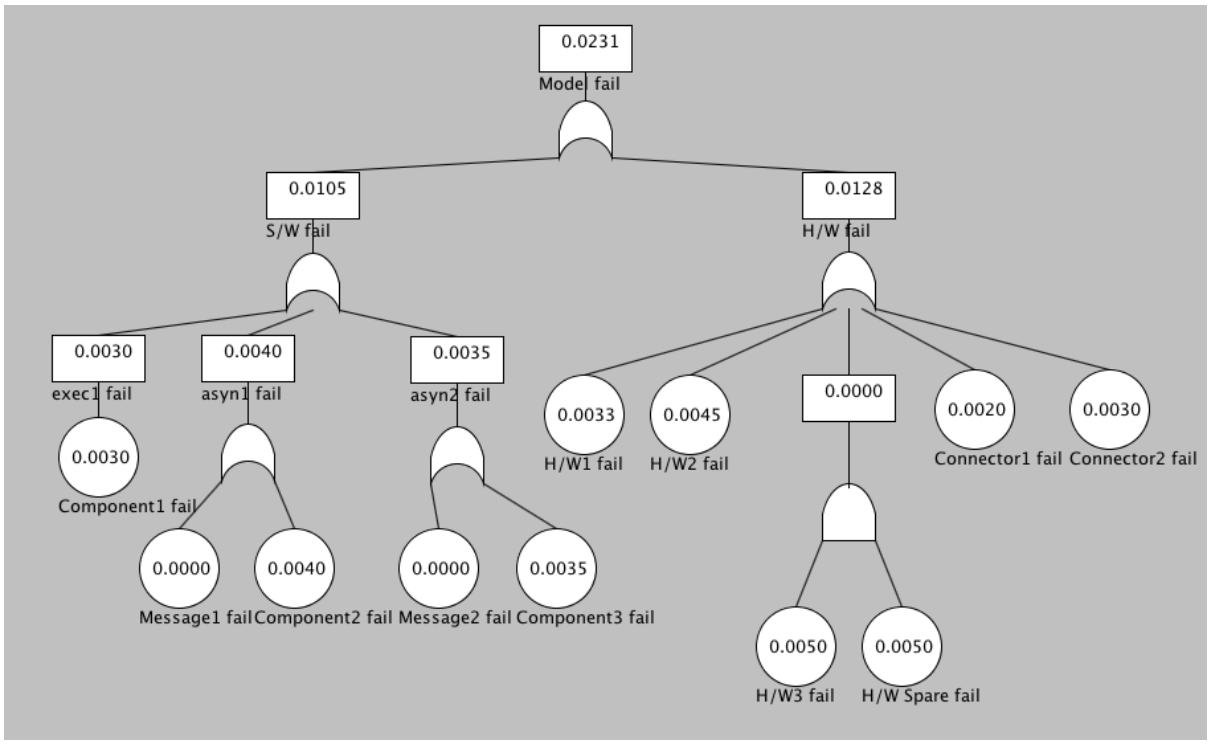


Figure 5.22 Fault tree applying D'Ambrogio's method

As we can see from Figure 5.22, according to the transformation method of D'Ambrogio et al., the Global Fault Tree consists of two parts: the software sub-tree and the hardware sub-tree, which are linked at the top with an Or_Gate. For software, the elements of the sequence diagram are mapped to corresponding intermediate or basic events, and for hardware, all devices of the system are linked with an Or_Gate while the redundant devices are being connected with And_Gate. We believe that this transformation method has a disadvantage, as it does not take into account the allocation of software to hardware components. In reality, the failure of a hardware processor will affect the software components running on it, thus we should take into account the allocation of software to hardware. For instance, if multiple software components are co-allocated to the same hardware processor, the failure of the hardware will affect all the software components running on it. The tree produced by D'Ambrogio et al. transformation does not show this kind of dependency between software and hardware, because their failures are separated into two independent sub-trees. In our view, the failure of a hardware processor should contribute to the failure of the software running on it. Therefore, we take into account the software to hardware allocation when generating the fault tree as explained in section 3.4.2.

Another difference between the transformation from [D'AMBROGIO02] and ours consist in handling the message failures. In their method of mapping, the failure of each message is a separate event with its separate probability of failure, which does not depend on the failure of the hardware connector conveying the message. In our case, we consider that a message can fail due to three events: the failure of the sending component, of the receiving component, and of the connector. Since the failure of the sending and receiving components are already included in the fault tree when transforming the component execution period into

a tree node, we consider that the failure of a message is given by the failure of the hardware connector delivering the message (see section 3.4.2).

The generated fault tree with our method is shown in Figure 5.23. We can see the software to hardware component allocation are being considered, and for message failure, we consider that the hardware connector of each message may fail. Furthermore, in our work there is no need to group the call and return of synchronous messages with their corresponding execution period, because messages and executions are being transformed independently in our work.

As we can see from Figure 5.23, instead of building separate sub-trees for software and hardware independently, the fault tree generated has merged the hardware tree into the software tree, so that software to hardware allocation is being covered in this transformation. The messages are being transformed to basic events with a failure probability given by the corresponding connector that conveys the message.

If we compare the two fault trees generated by applying D'Ambrogio et al. and our method respectively, we can see in this given example system where each component is allocated to its own software, and runs a single function, the results (both the number of effective basic events and the calculated failure occurrence probability of the model) of applying D'Ambrogio et al. and our method are the same.

However, if we consider a different example with co-allocation of components on the same hardware device, the results are different. Figure 5.24 shows a revised architecture of

the example system, in which component1 and component2 are co-allocated to the same H/W component (H/W1) while the original H/W2 and Connector1 are removed.

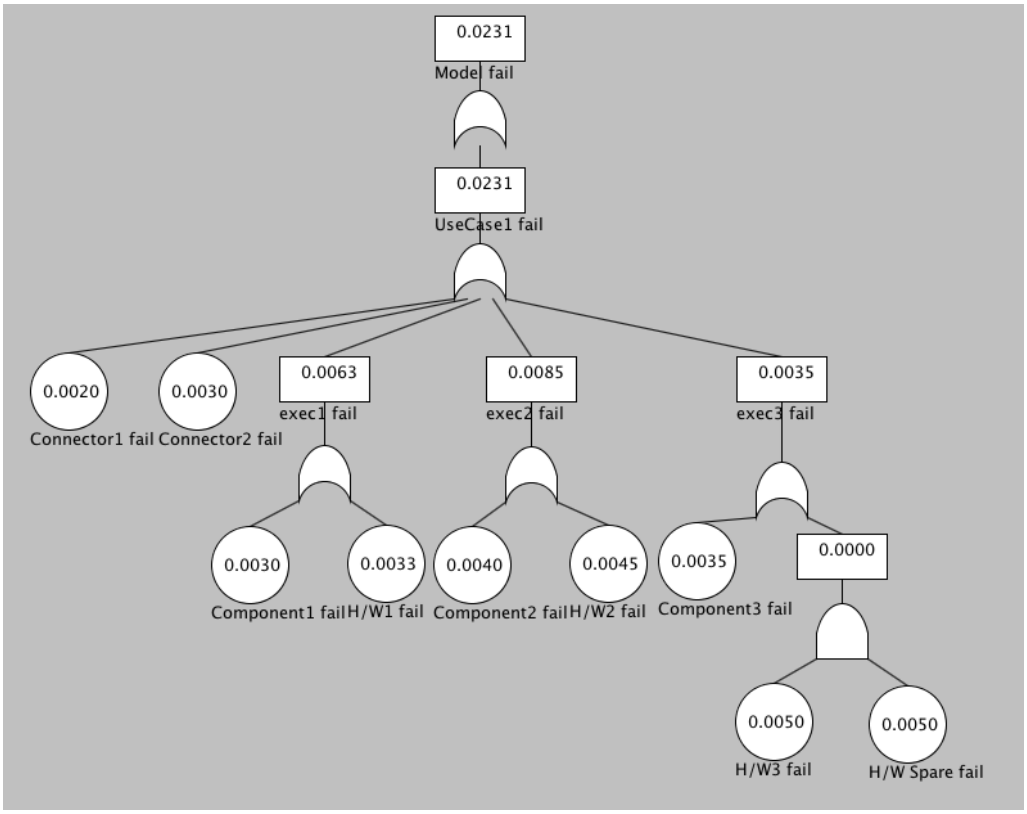


Figure 5.23 Fault tree applying our method

Figures 5.25 and 5.26 show the generated fault tree based on the revised composite structure diagram by applying D'Ambrogio et al. and our method, respectively. As we can see, under the architecture with co-allocation, the results of applying the two different methods are slightly different. The failure occurrence probability by applying D'Ambrogio et al. method is 0.0167, while by applying our method is 0.0200. The basic event "H/W1 fail" appears once in Figure 5.25 but twice in Figure 5.30, so that the total failure occurrence probability calculated by the latter fault tree is slightly higher. This is because both Component1 and Component2 are running on H/W1, so the failure of H/W1 will affect both of the soft-

ware components running on it. This kind of dependency was not demonstrated in D'Ambrogio's transformation.

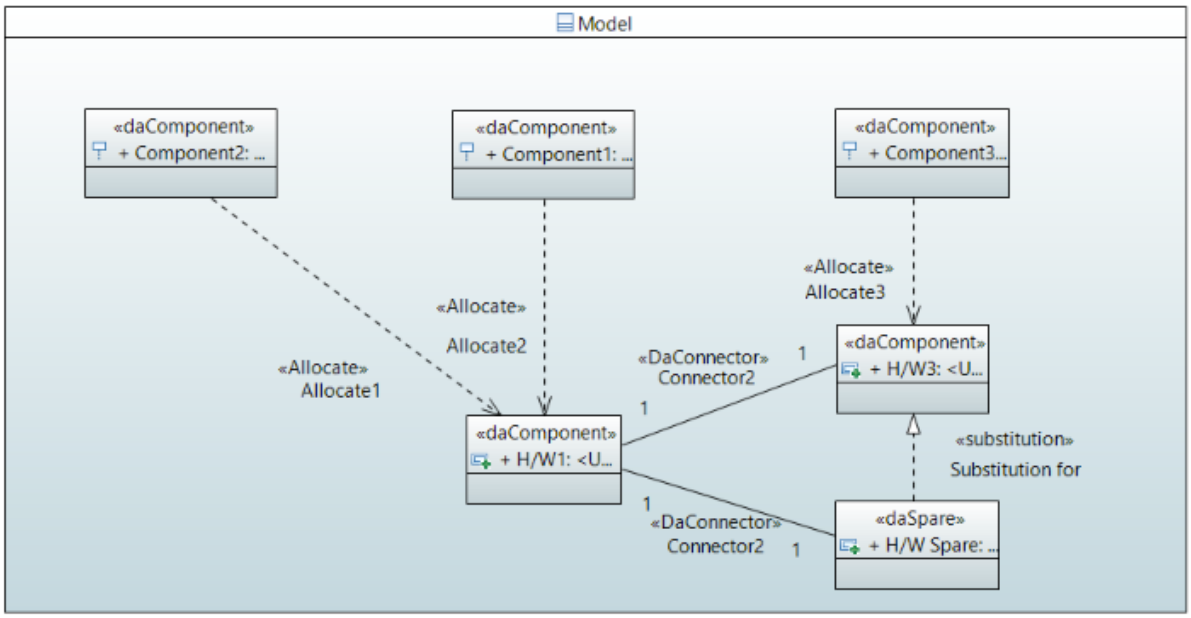


Figure 5.24 Revised Composite Structure diagram with co-allocation

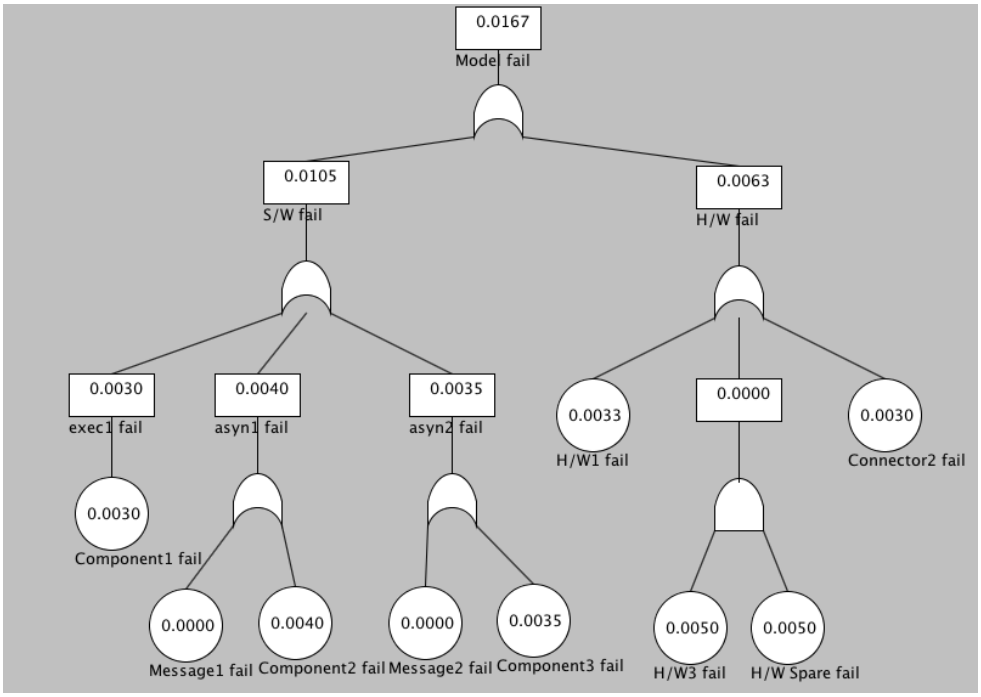


Figure 5.25 Fault tree applying D'Ambrogio's method with co-allocation structure

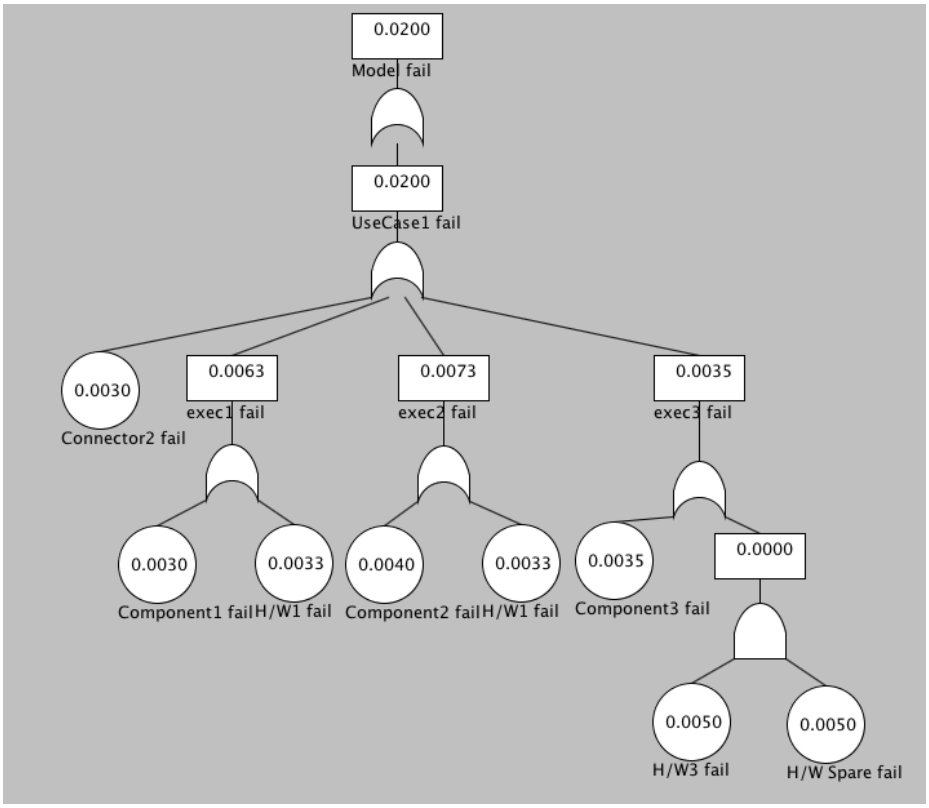


Figure 5.26 Fault tree applying our method with co-allocation structure

5.3 Case Study 2

For the second case study we took an example of a Message Redundancy Service (MRS) from [DAM], as shown in Figure 5.27 and 5.28 below. Figure 5.29 shows the rebuilt composite structure diagram for MRS.

As we can see from the diagrams below, the Use Case Diagram shows the main use case realized by the scenario given in the Sequence Diagram in Figure 5.28. MessageReplicator receives messages from Clients, then specifies target receivers and the file to deliver. The Redundancy Manager, which is in charge of the actual delivery, creates replicas called Payloads. Each Payload sends back to RM a result, that can be either of approval or of rejection. We assume the failure occurrence probability of each component as in table 5.3. The generated fault tree is shown in Figure 5.30:

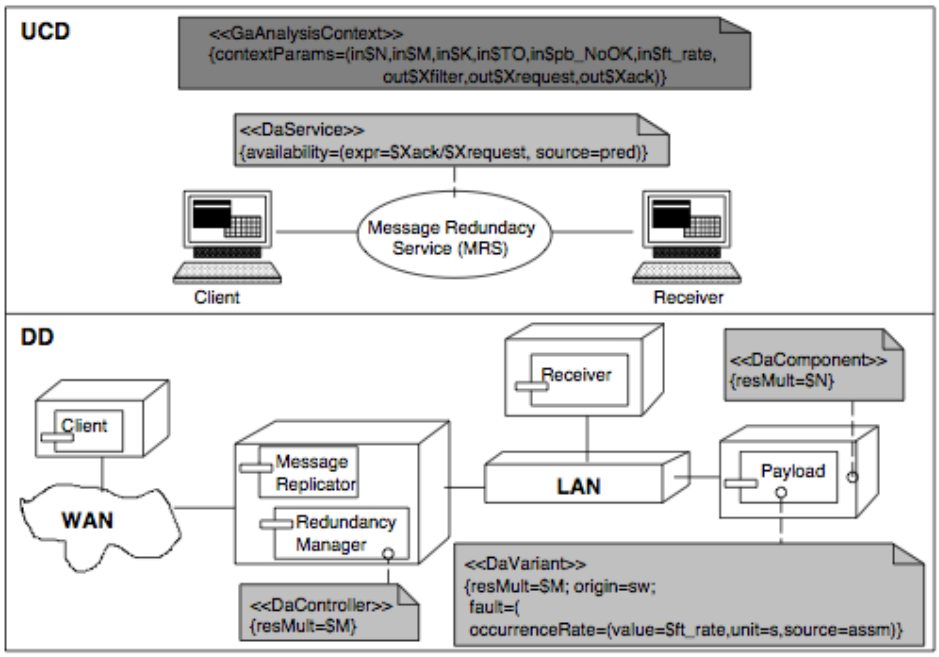


Figure 5.27 Message Redundancy Service overview (UCD) and architecture (DD)

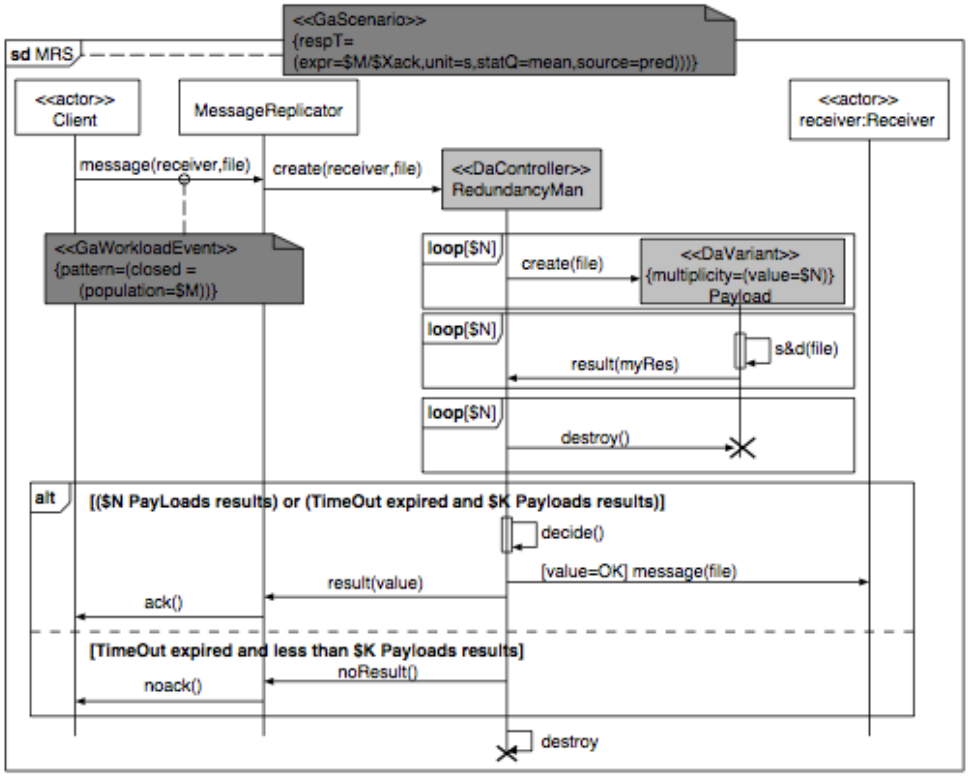


Figure 5.28 MRS scenario (SD diagram)

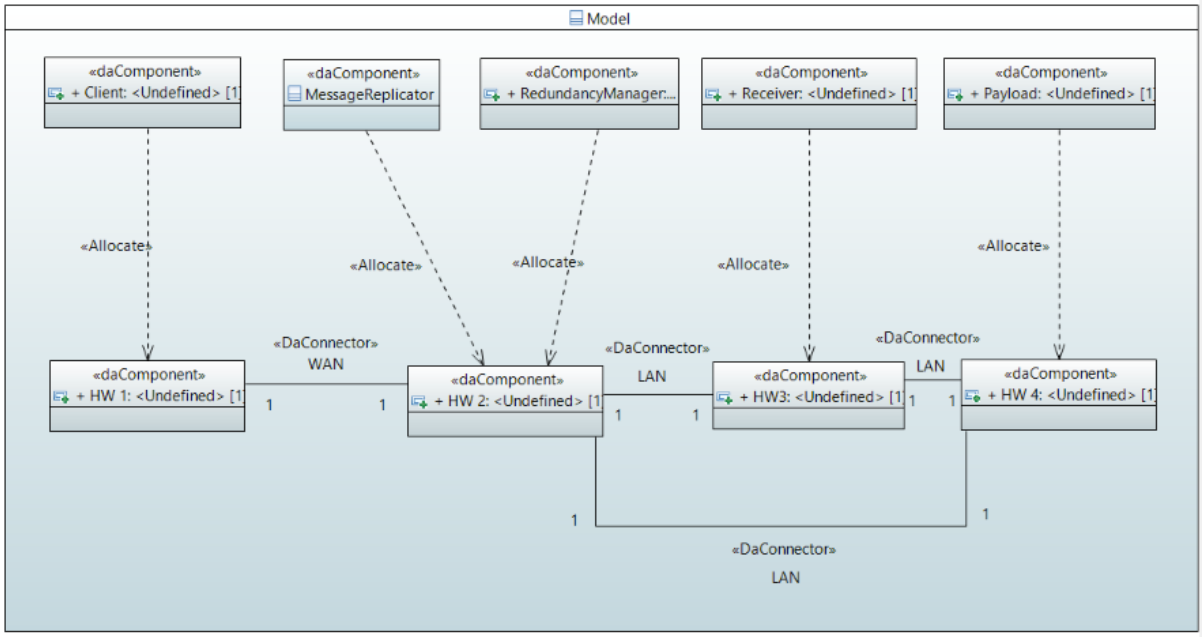


Figure 5.29 Rebuilt Composite Structure diagram for Case Study 2

Component name	Failure Occurrence Probability
《DaComponent》 :Client	0.005
《DaComponent》 :MessageReplicator	0.003
《DaComponent》 :ReduncancyManager	0.003
《DaComponent》 :Receiver	0.004
《DaComponent》 :Payload	0.0045
《DaComponent》 :HW 1	0.002
《DaComponent》 :HW 2	0.0025
《DaComponent》 :HW 3	0.0015
《DaComponent》 :HW 4	0.0035
《DaConnector》 :LAN	0.001
《DaConnector》 :WAN	0.001

Table 5.3: Input Failure Occurrence Probability of MRS

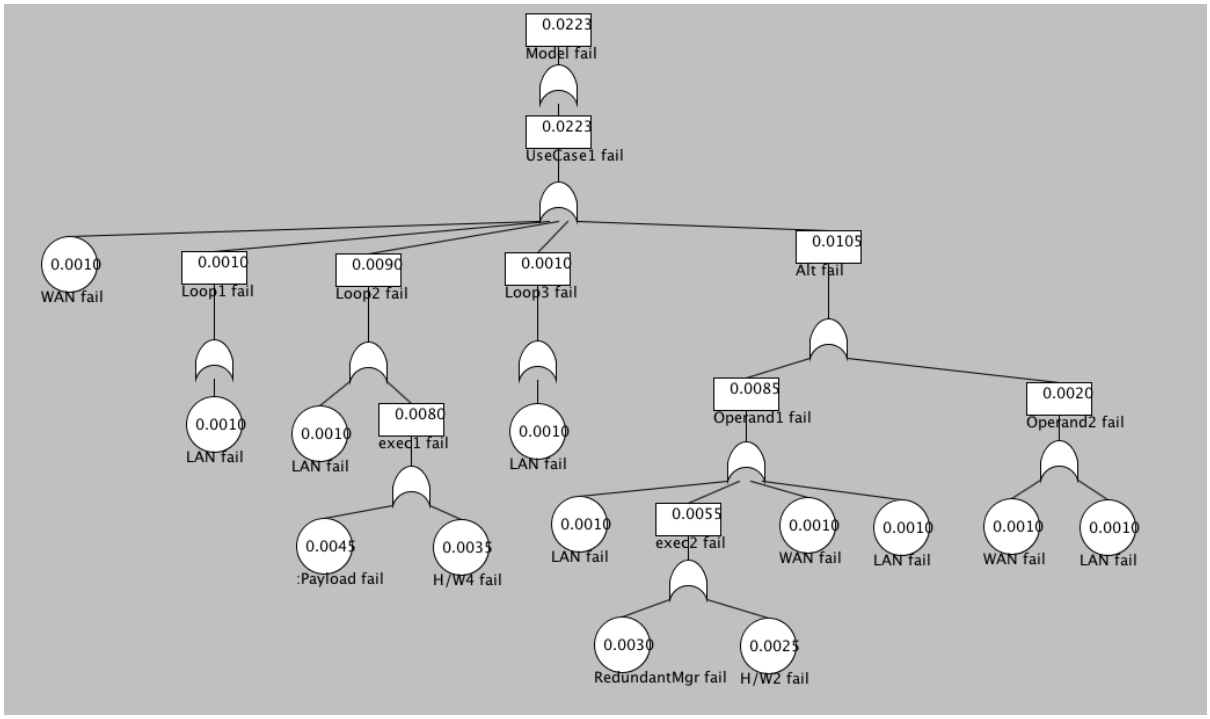


Figure 5.30 Generated fault tree of MRS

The analysis tool we used in this work, FaultCAT, provides two kinds of reports, the component report and calculation report. The two reports produced by FaultCAT for the system of our Case Study 2 are shown in Figure 5.31 and 5.32 below.

As we can see from these Figures below, the component report shows the detailed failure probability of each component within the fault tree. More specifically, what it is shown are all the events in the fault tree. Gates are not shown since they do not have their own probability or information field, as shown in Figure 5.31. The calculation report shows the calculations done by each gate by showing each Gate and how it calculates the probability from each of its children.

Figure 5.31 Component report of Case Study 2

```
CALCULATION REPORT
Shows the calculations done by the Gates
================================================

LOOP1 OR( 0.001 )
exec1 OR(0.0045 + 0.0035- ( 0.0045 * 0.0035)
LOOP2 OR(0.001 + 0.00798425- ( 0.001 * 0.00798425)
LOOP3 OR( 0.001 )
exec2 OR(0.003 + 0.0025- ( 0.003 * 0.0025)
operand1 OR(0.001 + 0.0054925- ( 0.001 * 0.0054925) + 0.0064870075 + 0.001 -
( 0.0064870075 * 0.001 ) + 0.0074805204925 + 0.001 - ( 0.0074805204925 * 0.001 ))
operand2 OR(0.001 + 0.001- ( 0.001 * 0.001)
Alt OR(0.0084730399720075 + 0.001999- ( 0.0084730399720075 * 0.001999)
UseCase1 OR(0.001 + 0.001- ( 0.001 * 0.001) + 0.001999 + 0.008976265749999999 - (
0.001999 * 0.008976265749999999 ) + 0.010957322194765747 + 0.001 -
( 0.010957322194765747 * 0.001 ) + 0.011946364872570982 + 0.010455102365103459 -
( 0.011946364872570982 * 0.010455102365103459 ))
Model OR( 0.022276566770040838 )
================================================
```

Figure 5.32 Calculation report of Case Study 2

As shown in Figure 5.32, unlike conventional logic gate diagrams where inputs and outputs hold the binary values of either true (1) or false (0), the gates in a fault tree output probabilities rare elated to the set operations of boolean logic. The probability of a gate's output event depended on the input event probabilities. An And_Gate represents a combination of independent events so that the probability of any input event to an And_Gate is unaffected by any other input event to the same gate. In set theoretic terms, this is equivalent to the intersection of the input event sets, and the probability of the and gate output is given by: P (A and B) = P (A ∩ B) = P(A) P(B); An OR_Gate, on the other hand, corresponds to set union: P (A or B) = P (A ∪ B) = P(A) + P(B) - P (A ∩ B). [FAULTTREE]

Moreover, traditional solution of fault trees involves the determination of the so-called minimal cut sets, which is also included in FaultCAT tool. Cut sets are the unique combinations of component failures that can cause system failure. Specifically, a cut set is said to be a

minimal cut set if, when any basic event is removed from the set, the remaining events collectively are no longer a cut set [KECECIOGLU1991]. In the example fault tree (Figure 5.30) there are only Or_Gates, meaning that any of the basic event fails will cause system failure. Therefore, each of the individual basic event is considered to be part of the minimal cut set. Figure 5.33 shows the report of minimal cut sets provided by FaultCAT.

```
FAULT TREE MINIMAL CUT SETS
Shows the minimal cut sets for each tree
=============================================

( WAN fail ) + ( LAN fail ) + ( RedundantMgr fail ) + ( H/W2 fail ) + ( LAN
fail ) + ( WAN fail ) + ( LAN fail ) +
( LAN fail ) + ( :Payload fail ) + ( H/W4 fail ) + ( LAN fail ) + ( LAN fail ) +
( WAN fail )

=============================================
```

Figure 5.33 Minimum cut set of Case Study 2

Chapter 6. CONCLUSION

6.1 Accomplishments

This thesis has proposed and implemented a model transformation by using ATL transformation language, in which we have transformed UML 2.4 software models composed of sequence diagrams, composite structure diagrams, and use case diagrams, along with applied MARTE/DAM stereotypes into Fault Tree Models.

The UML modeling tool we used in this work is Papyrus, which is an open-source tool based on the Eclipse environment. One of the reasons we selected Papyrus was because the ATL engine was able to directly accept the Papyrus UML model extended with profiles as the source model.. Our ATL transformation model, UML2FT, also accepted the custom target Fault Tree metamodel, which we created with the ECORE tool. Using the Fault Tree metamodel, ATL was able to produce an XMI file of the target model, which could then be processed by a simple JAVA program and then be read and analyzed by an existing fault tree analysis tool. We selected the open-source tool FaultCAT as our Fault Tree analysis tool because it accepts input in an XML format, very similar with the output generated by our transformation. There are more sophisticated fault tree analysis tools available, but the problem is that they have their own input language different from XML, requiring a more complex translation of the generated fault tree.

The ATL UML2FT module we have designed, implemented and tested consists of 7 matched rules to handle conditional mappings, and 4 lazy rules for all unconditional mappings. We also used 32 helpers that may be called by transformation rules to achieve the transformation.

6.2 Limitation

There are a few limitations with the ATL model transformation of this thesis. First of all, the scope of the thesis includes only regular fault trees as our target fault tree model. We have not considered extended versions, such as dynamic fault tree introduced by Dugan et al., [DUGAN2002] which extend regular fault trees with dynamic gates (spare gates, priority AND gates, etc.) for the modeling of fault tolerant hardware systems. Extensions to regular fault tree is left as future work. The main reason we focused on regular fault trees is due to the fact that the analysis tool we use (FaultCAT) only supports this kind of models.

Another limitation is that we did not use conditional events in our work, even though regular fault trees may include such elements. This limitation is also due to the FaultCAT tool. A case were we could have used conditional events in our transformation is to model UseCases as Intermediate events associated with an occurrence probability that should be mapped to a conditional event technically.

The third limitation of this work is due to the modeling tool we use, Papyrus, which in some aspects may not be complete enough to cover all those functionalities we need in our transformation. For example it does not allow users to model communication path in deployment diagrams, and it does not support the connector between ports of classes in the composite structure diagram. Also in Papyrus, the lifelines in sequence diagrams could only represent properties in composite structure diagram. All this limitations have made our implementation not so smooth and straightforward. Some work-around solutions were designed for dealing with the problem of the tool.

6.3 Directions for Future Work

Although the transformation process for UML+MARTE/DAM Models to Fault Tree models was successfully realized in this thesis, there remain several opportunities for future work:

1. Transformation integration: As demonstrated throughout this work, the entire transformation process requires several manual steps. For reasons of convenience, it would be extremely beneficial if all these steps (including running ATL transformation, running output processing program) could be integrated in to one single program, as an extension to Papyrus. Thus, creating transformation instances as well as executing them would be easier for a user.

2. Extend the generated fault tree to other alternatives, such as dynamic fault trees, or other elements in standard fault trees, such as conditional events.

3. Supporting other analysis tools: As we indicated previously, the reason we chose to use FaultCAT as our analysis tool is that it is the only one we could get that directly supports XML input/output, but it is not the best one in doing Fault Tree analysis. Though other tools have their own language, since we have already transformed UML model to a tree structured model, the gap between our target model and those of other tools will not be that big anymore. We may transform our XML fault tree model to other model supported by different tools in the future.

4. Improving the source model: Our current source model was built with Papyrus, but the tool has some bugs that may limit our implementation. We have already reported theses

bugs to the forum. As soon as they fix them, we would like to revise our source model, to eliminate the work around the bugs.

REFERENCES

[ATL2012] ATL User Documentation, http://wiki.eclipse.org/ATL/User_Guide, 2012

[BERNARDI2008] S. Bernardi, J. Merseguer, D.C. Petriu, "Adding Dependability Analysis Capabilities to the MARTE Profile", in Proc. of the 11th international conference on Model Driven Engineering Languages and Systems (MODELS'2008), LNCS Vol. 5301 (K. Czarnecki, editor), pp. 736 - 750, Springer, 2008.

[BERNARDI2011] Bernardi, Simona, Merseguer, José, Petriu, Dorina C.: A dependability profile within MARTE. Software and System Modeling. 10(3), pp. 313–336 (2011).

[BERNARDI2012] Bernardi, Simona, Merseguer, José, Petriu, Dorina C,, "Dependability modeling and analysis of UML-based software systems", ACM Computing Surveys, Vol. 45, Issue 1, November 2012, (48 pages)

[BERNARDI2013] Bernardi, Simona, Merseguer, José, Petriu, Dorina C, Model-Driven Dependability Assessment of Software Systems, ISBN 978-3-642-39511-6, Springer, October 2013.

[D'AMBROGIO2002] D'Ambrogio, A., G. Iazeolla, and R. Mirandola. "A method for the prediction of software reliability." Proceedings of the 6-th IASTED software engineering and applications conference (SEA2002), Cambridge, MA. 2002.

[DOMIS2008] Domis, Dominik, and Mario Trapp. "Integrating safety analyses and component-based design." Computer Safety, Reliability, and Security. Springer Berlin Heidelberg, 2008. 58-71.

[DUGAN2002] Ganesh J Pai and Joanne Bechta Dugan, "Automatic Synthesis of Dynamic Fault Trees from UML System Models", The 13 th International Symposium on Software Reliability Engineering, 2002

[EMF2013] Eclipse Kepler (4.3) Documentation, http://help.eclipse.org/kepler/index.jsp, 2013

[FAULTCAT] Fault Tree Creation and Analysis Tool User Manual V1.0, http://www.iu.hio.no/FaultCat/manual.htm, 2003.

[FAULTTREE2006] Fault Tree Analysis. Edition 2.0, International Electrotechnical Commission. IEC 61025 ED. 2.0, 2006.

[FAULTTREE+] FaultTree+ V11 Technical Specification, http://www.isograph-software.com/_techspecs/psa32techspec.pdf, Isograph Reliability Software, 2005.

[FMEA1996] Goddard Space Flight Center (GSFC) (1996-08-10). Performing a Failure Mode and Effects Analysis, Goddard Space Flight Center, http://www.everyspec.com/NASA/NASA-GSFC/GSFC-Code-Series/GSFC_431_REF_000370_2297/, 1996.

[FTHANDBOOK] Michael Stamatelatos, William Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick, Jan Railsback, "Fault Tree Handbook with Aerospace Applications", Version 1.1, NASA, http://www.hq.nasa.gov/office/codeq/doctree/fthb.pdf

[GRUNSKE2005A] Lars Grunske, Bernhard Kaiser, "Automatic generation of analyzable failure propagation models from component-level failure annotations", Fifth International Conference on Quality Software, (QSIC 2005), pp.117-123, 2005.

[GRUNSKE2005B] Lars Grunske, Bernhard Kaiser, "An Automated Dependability Analysis Method for COTS-Based Systems", In COTS-Based Software Systems, Lecture Notes in Computer Science, Volume 3412, pp 178-190, 2005.

[HARPER2007] C. Parker and A. Parkinson. "Fault Tree Analysis of UML Designs.", http://www.omg.org/news/meetings/workshops/SWA_2007_Presentations/04-2_Harper-Parkinson.pdf, 2007.

[HASSAN2005] A. Hassan, K. Goseva-Popstojanova, and H. Ammar, "UML Based Severity Analysis Methodology", Reliability and Maintainability Symposium, 2005.

[HU2011] Wensheng Hu and Zhouhui Deng, "A Method of FTA Base On UML Use Case Diagram", Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference, 2011

[JOSHI2007] Joshi, Anjali, Steve Vestal, and Pam Binns. "Automatic generation of static fault trees from AADL models." Workshop on Architecting Dependable Systems of The 37th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks, Edinburgh, UK. 2007.

[KECECIOGLU1991] Kececioglu, D., Reliability Engineering Handbook, Volume 2, Prentice Hall, Inc., New Jersey, 1991.

[LAUER2009] Christoph Lauer, Reinhard German and Jens Pollme, “Fault Tree Generation from EMF Models”, http://www.cs.kent.ac.uk/events/conf/2009/wads/Slides/lauer.pdf, 2009.

[LAUER2011] Lauer, Christoph, Reinhard German, and Jens Pollmer. "Fault tree synthesis from UML models for reliability analysis at early design stages." ACM SIGSOFT Software Engineering Notes 36.1 (2011): 1-8.

[MARTE2011] UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, 2011

[MDA2003] Object Management Group, "MDA Guide Version 1.0.1" , document omg/03-06-01, 2003.

[MOF2011] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification,Version 1.1 - January 2011.

[OPENFTA] OpenFTA, Version 1.0, User Manual, 2005.

[PAPYRUS2011] PAPYRUS USER GUIDE SERIES, version 1.0.0, 2011

[QVT2011] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011

[SELIC2003] Selic, Bran (2003) “The pragmatics of model-driven development”. IEEE Software, Vol. 20, Issue 5, pp.19–25, May 2003.

[SPT2005] Documents Associated With UML Profile For Schedulability, Performance, And Time, Version 1.1, 2005

[XIANG2011] Jianwei Xiang, Yanoo K, Maeno Y, and Tadano K. "Automatic synthesis of static fault trees from system models." Secure Software Integration and Reliability Improvement (SSIRI), 2011 Fifth International Conference on. IEEE, 2011.