A framework for the analysis of failure behaviors in component-based model-driven development of dependable systems

Name of students: Muhammad Atif Javed, Faiz UL Muram
Mälardalen University, IDT Department
Examiner: Sasikumar Punnekat
Supervisor: Barbara Gallina

Date 2011-11-07
ABSTRACT

Currently, the development of high-integrity embedded component-based software systems is not supported by well-integrated means allowing for quality evaluation and design support within a development process. Quality, especially dependability, is very important for such systems.

The CHESS (Composition with Guarantees for High-integrity Embedded Software Components Assembly) project aims at providing a new systems development methodology to capture extra-functional concerns and extend Model Driven Engineering industrial practices and technology approaches to specifically address the architectural structure, the interactions and the behaviour of system components while guaranteeing their correctness and the level of service at run time. The CHESS methodology is expected to be supported by a tool-set which consists of a set of plug-ins integrated within the Eclipse IDE.

In the framework of the CHESS project, this thesis addresses the lack of well integrated means concerning quality evaluation and proposes an integrated framework to evaluate the dependability of high-integrity embedded systems.

After a survey of various failure behavior analysis techniques, a specific technique, called Failure Propagation and Transformation Calculus (FPTC), is selected and a plug-in, called CHESS-FPTC, is developed within the CHESS tool-set. FPTC technique allows users to calculate the failure behavior of the system from the failure behavior of its building components. Therefore, to fully support FPTC, CHESS-FPTC plug-in allows users to model the failure behavior of the building components, perform the analysis automatically and get the analysis results back into their initial models. A case study about AAL2 Signaling Protocol is presented to illustrate and evaluate the CHESS-FPTC framework.
# TABLE OF CONTENTS

1. INTRODUCTION ................................................................. 5
   1.1 Context and motivation ................................................. 5
   1.2 Contributions ........................................................... 5
   1.3 Structure of Thesis ..................................................... 6

2. BACKGROUND ................................................................. 7
   2.1 Dependability ............................................................ 7
       2.1.1 Attributes ......................................................... 7
       2.1.2 Dependability Threats .......................................... 7
       2.1.3 Failure modes .................................................. 8
   2.2 Component-Based Software Engineering ....................... 9
   2.3 Model Driven Architecture .......................................... 11
       2.3.1 Model Transformation Types ................................ 13
       2.3.2 Acceleo Model Transformation Engine .................. 14
   2.4 MDE within CHESS .................................................. 17
   2.5 Failure Behavior Analysis Techniques ......................... 19
       2.5.1 Component Fault Tree ........................................ 19
       2.5.2 Failure Propagation Transformation Notation ........... 20
       2.5.3 Hierarchically Performed Hazard Origin and Propagation Studies ............................................. 20
       2.5.4 State-Event Fault Tree ....................................... 21
       2.5.5 Architecture Analysis and Description Language .... 21

3. PROBLEM FORMULATION AND ANALYSIS ....................... 27

4. METHOD ................................................................. 29
   4.1 Selection of Proper Failure Behavior Analysis Technique ............................................... 29
   4.2 Understanding and Comparison of FPTC rules defined by different authors .................... 30
   4.3 Integration of CHESS-FPTC Plug-in with CHESS Tool-Set .............................................. 31
   4.4 Selection of Transformation Technology ............................................................... 32
       4.4.1 Why we select Acceleo for M2T ................................ 32
       4.4.2 Back Propagation .............................................. 33

5. SOLUTION ................................................................. 34
   5.1 Proposed FPTC ......................................................... 34
       5.1.1 Multiple incoming failure on Input Ports ................. 34
       5.1.2 No Pattern Matching ........................................ 34
       5.1.3 Usage of Wildcard ............................................ 34
       5.1.4 Specificity of Wildcards and Variables ................ 34
       5.1.5 Modified Syntax of FPTC ..................................... 34
   5.2 Meta-Modeling Concepts Provided to Extends CHESS ML ............................................ 35
       5.2.1 Component Model ................................................ 35
       5.2.2 Components ....................................................... 35
       5.2.3 Interfaces .......................................................... 35
       5.2.6 Selection of Platforms ......................................... 36
       5.2.7 Software/Hardware Systems ................................ 36
   5.3 Proposed Design ........................................................ 37
       5.3.1 Design of CHESS-FPTC Plug-in ......................... 37
5.3.2 Integration of FPTC Plug-in in CHESS Platform ................................................................. 38

5.4 Implementation ....................................................................................................................... 39
  5.4.1 Modeling Exemplar System ................................................................................................. 40
  5.4.2 Activate/Deactivate Call Command .................................................................................... 42
  5.4.3 Launch FPTC Analysis ......................................................................................................... 43
  5.4.4 Call Build Instance .............................................................................................................. 44
  5.4.5 Read Model and Generate XML File .................................................................................... 44
  5.4.6 Read Generated Input XML File ......................................................................................... 48
  5.4.7 Apply Propagation ............................................................................................................. 50
  5.4.8 Propagation of Analysis Results Back into Model ............................................................... 52

6. CASE STUDY ............................................................................................................................. 54
  6.1 Overview of AAL2 Signaling .................................................................................................. 54
    6.1.1 AAL2 Signaling Protocol Functions .................................................................................. 54
    6.1.2 AAL2 Signaling Protocol Architecture ............................................................................ 54
  6.2 Experiment Overview .......................................................................................................... 55
    Modeling AAL2 Signaling Protocol in CHESS ....................................................................... 56
    6.2.1 Functional Modeling of AAL2 Signaling Protocol Components ...................................... 57
    6.2.2 Possible Failure Behavior of Components ....................................................................... 57
    6.2.4 FPTC Analysis at Sub-System Level ............................................................................... 63
    6.2.5 FPTC Analysis at System Level ...................................................................................... 68
    2.6.7 Discussion ....................................................................................................................... 71

7. CONCLUSION ........................................................................................................................... 72

8. FUTURE WORK ......................................................................................................................... 73
  8.1 Enhancement in CHESS-FPTC plug-in .................................................................................. 73
  8.2 FPTC Extension for FI4FA ...................................................................................................... 73
Chapter 1

INTRODUCTION

Thesis introduction is presented in this chapter. This chapter is organized as follow: In section 1.1 we provide context and motivation of the thesis. Section 1.2 highlights our contribution for achieving main aim of the thesis and section 1.3 provides the idea about main structure of the thesis.

1.1 Context and motivation

Nowadays development of real-time embedded systems inclines towards usage of Component-Based Development (CBD) and Model Driven Engineering (MDE) approaches. The combination of these two approaches promises to better management of complexity, increase reusability and guarantee easier maintenance, cost reduction and risks associated with the development and implementation. Current component-based run-time environment and their combination with model driven engineering approaches (transformation engine, code generators) specify the functional properties of components but do not non-functional properties in adequate manner [11].

The CHESS (Composition with Guarantees for High-integrity Embedded Software Components Assembly) project aims to address limitation by providing a methodology to engineer dependable high integrity real-time component-based embedded system. The CHESS methodology is meant to allow engineers to address safety, reliability, and other non-functional concerns, while guaranteeing correctness of component development and component composition for real-time embedded systems. CHESS addresses the challenges of property preserving component assembly in real-time and dependable embedded systems.

The CHESS toolset supports CHESS methodology, an industrial-quality Model-Driven Engineering (MDE) infrastructure for specification, analysis and verification of non-functional properties such as dependability in component-based software system. CHESS toolset is expected to be supports integration of dependability property with the component model. Identification of analysis methods for which the model to be analyzed can be somehow derived from the component architecture specification [11].

The CHESS toolset contains a set of plug-ins integrated within Eclipse IDE. CHESS dependability profile (UML profile) allows users to model and analyze the dependability characteristic such as calculating failure behavior of components. The context of this thesis is the CHESS project, which aims at providing a tool-supported component-based and model-driven-based methodology to engineer high integrity real-time component-based dependable embedded systems. Study and comparison of techniques for failure behavior analysis (focus on FPTC-based analysis’ techniques). Analysis, design and implementation of FPTC plug-in in CHESS tool-set.

1.2 Contributions

CHESS-FPTC plug-in integrated with CHESS tool-set will be provided which allows the users to model the system and perform the analysis on it and gets the analysis on the model.

A survey on failure behavior analysis techniques is conducted and selects the proper technique which is better for analyzing the dependability (i.e. failure behavior of a component) of the high integrity
real-time component-based dependable embedded systems. We select Failure Propagation and Transformation Calculus (FPTC) technique for automatic calculation of failure behavior of entire system from failure behavior of its building components. We compare the syntax of FPTC presented different authors and modified it.

We analyze CHESS tool-set for identifying available and missing parts that are necessary for performing FPTC technique within CHESS model. Studying and analyzing of CHESS Modeling Language that is called CHESS ML profile. CHESS ML is built from subsets of standard languages Sys ML, UML, MARTE.

We implement transformation from high level CHESS ML model to low level dependability analysis and back transformation of analysis result on model. We perform Model-to-text transformation for getting the necessary information about model in the form of XML (EXtensible Markup Language) file. Model-to-text transformation is done by using Acceleo transformation engine. Generated XML file contains information like components along with their input and output ports, input failures, FPTC rules and connections between components. This generated XML file is taken as input XML for performing FPTC analysis. After calculating the behavior of the model, analysis results are back propagated into initial model. Back propagation is helpful for users to mitigate failure behavior of system by changing and introducing new components. We provide a case study to evaluate this framework.

**1.3 Structure of Thesis**

Chapter 2 introduces the background that is necessary for understanding the problem as well as for building solution. This chapter explores some basic concepts.

Chapter 3 describes the problem as well as analysis of the problems. For better understanding of the main problem it is sub-divided into various sub problems.

Chapter 4 presents methods to find the solution of problem by selecting proper analysis technique.

Chapter 5 presents solution of the problem by integrating analysis technique (FPTC) within CHESS.

Chapter 6 introduces a case study taken from the telecommunication domain.

Chapter 7 concludes the thesis work by highlighting the benefits and hints of the work.

Chapter 8 indentifies possible extension and direction of future work.
Chapter 2

BACKGROUND

In chapter 2 we provide basic background material which is necessary to achieve the goals of thesis and this chapter is organized as follow: Section 2.1 provides information about dependability; section 2.2 describes component-based software engineering approach. Section 2.3 highlights the basic concepts of model-driven architecture and transformation techniques. Section 2.4 explores the CHESS project in terms of model-driven engineering. In section 2.5 we provide the information about various analysis techniques. Section 2.6 describes the detail about Failure Propagation and Transformation Calculus technique.

2.1 Dependability

Dependability is a concept that stresses on trustworthiness and can be defined as “The ability to deliver service that can justifiably be trusted” [1]. The service provided by a system is its behavior as perceived by its user(s). The function of a system is what a system is expected to do and is described by functional specification. Correct service is delivered when a service performs the system function. A service failure occurs when delivered service deviates from the correct service. Thus, a failure can be defined as transition from correct service to incorrect service. “A dependability or security failure occurs when the given system suffers service failures more frequently or more severely than acceptable.” [1]. Dependability includes attributes, threats which are briefly discussed below:

2.1.1 Attributes

“The dependability attributes define the main facets of dependability that are relevant for the target system and applications” [18]. Dependability includes safety and reliability attributes and these attributes can be defined as:

- **Safety**: absence of sudden failure and disaster effects on environment and user(s).
- **Reliability**: continuity of correct services according to specification.

2.1.2 Dependability Threats

Dependability can be affected by threats named as faults, errors and failures which are introduced below. Figure 1.1 is used to support the explanation of these terms.

- **Fault**

A fault is hypothesized and adjudged cause of an error. A fault has two states active and dormant. When a fault becomes active it causes an error otherwise it remains dormant. A fault is an event which causes change of state of the system and takes system from valid state to erroneous state. In Figure 1.1 t1 represents a fault. Faults of a system can be categorized as external and internal faults. “The prior presence of vulnerability, i.e., an internal fault that enables an external fault to harm the system, is necessary for an external fault to cause an error and possibly subsequent failure(s)” [1].
### Error

“An error is the part of the total state of the system that may (in case the error succeeds, by propagating itself, in reaching the external system state) lead to its subsequent service failure” [1]. The erroneous state (with solid blue color) is presented in Figure 2.1.

![Figure 2.1: Valid and Erroneous States](image)

#### Failure

“A failure is an event that occurs when the delivered service (behavior of a system perceived by a user) deviates from correct service (the system specification)” [1]. Transition state t3 in Figure 1.1 indicates failure. A service may fail because of two reasons: first, service does not act according to the functional specification and second specification does not describe function correctly [1]. A failure can be avoided when there is a chance to bring back or forward the system to a valid state t2 transitions in Figure 2.1 shows this case.

Normally, a failure occurs due to the propagation of (several) errors beyond the system boundary. A failure may cause a fault elsewhere in another system. The causality chain that relates faults, errors and failures is presented in Figure 1.2 below:

![Figure 2.2: Causality chain among threats](image)

#### 2.1.3 Failure modes

A system does not always fail in the same way. The ways a system can fail are its failure modes. A failure mode can be described according to four points of view. These points of views are domain, consistency, delectability and consequences for the environment. Only the domain point of view is related to our scope of thesis so, we only explain domain point of view.

- For **domain** two viewpoints have to be considered named as **content failure** ("The content of information delivered at the service interface deviates from implementing the system..."
According to A. Bondavalli and L. Simoncini (1990) [22], “the behavior of a system as perceived by the system users is usually referred as the service delivered by the system to its users and a failure is a deviation of the delivered service from specified conditions” [22].

- **Service** is specified by two parameters named as value and time.
  - *Correctly-timed delivered service* defines a service delivered on time (time interval during which service is expected).
  - *Correctly-valued service* is composed of values which are correctly implemented.

Similarly to the domain point of view explained before, also according to these authors, failures can be classified into timing and content (value) failures. These authors however, extend the classifications.

- *Not Correctly-timed service* can be classified into early timing, late timing and finite late or omission (means service is never delivered).
- *Not Correctly-valued service* is composed of values for which specified services are not correctly implemented.
- Not Correctly-valued service is classified into *Subtle Incorrect* (user, “on the basis of his knowledge cannot detect as Not Correctly-valued and therefore it can only consider Correctly-valued”), *Coarse Incorrect* (“results are detectable by user”) and Omission (“either provides the correct outputs in response to the inputs it receives or does not provide any output”) [22].

### 2.2 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is a systematic and structured approach that allows engineers to maximize reusability. CBSE is also known as Component-Based Software Development (CBSD). The goal of CBSE is to compose the applications with plug & play software components on the frameworks. The main characteristics of CBSE are presented below:

1. CBSE considers a component as a reusable entity.
2. CBSE supports the development of system as the integration of components.
3. CBSE provides facilities for upgrading and maintaining a system by simply changing the components that needs to be upgraded or replaced.

CBSE is an extension of object-oriented concepts such as encapsulation (information hiding), abstraction (what an element is and how it should be implemented), polymorphism (same operation behave differently on different elements), Inheritance (sharing of operation and attributes among elements based on hierarchical relationships). CBSE approach has many advantages like: increase in productivity, improvement in quality, reduced time to market, broad range of reusability and effective management of complexity [3].

The fundamental concepts on which CBSE is based are:

- **Component**

Many different definitions of component exist. One of them is given by Szyperski [3]: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to
composition by third party”. Main points of this definition indicate that component can be deployed independently and each component interacts with other component(s) by using interfaces.

Other is given by Bill Councill and George Heineman [5]:

“A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard”. “A component model defines specific interaction and compositions standards”.

Fundamental characteristics of components are presented below:

- **Independent**: A component must be independent from its environment and is deployed without needs of other specific components.
- **Standardized**: In CBSE approach a component should followed deployment and composition rule [2].
- **Deployable**: For a component to be deployable, a component has to be self contained and must be able to perform as a stand-alone entity on some component platform that implements the component model. It means that usually a component is a binary component and cannot be compiled before its deployment.
- **Documented**: A component should be specified formally.
- **Composable**: A component communicates with others through its public interfaces. In addition, it must provide external access to information about itself such as its methods and attributes.

**Interface**

“An interface of a component can be defined as a specification of its access point” [3]. An interface is a set of functional properties which contain set of actions that can be understood by both interface provider (component) and user (other components or other software that interact with provider) [4]. Through access points, clients access the services that are provided by a component. A component may have more than one access point, which contains different services provided by that component. Therefore, a component may have more than one interface. Since components are black box, their implementation detail is not accessible from outside.

A component interfaces defined in standard component technologies can reveal functional properties. Functional properties include behavior part in which, behavior of a component can be specified and signature part in which operations provided by a component are indicated. We can distinguish two kinds of interfaces. “Components can export and import interfaces to and from environments that may include other components” [3].

- Imported interface describes those services that a component requires from its environments.
- An exported interface specifies those services that a component provides to its environment. Generally used graphical symbol for a component with it’s required and provide interface is presented in Figure2.3.
Component Composition

“Component composition is based on the ability to assign properties to the whole based on the properties of the parts and the relationships between the parts”. [3] Composition is a combination of two or more components that yielding a new component behavior at different abstraction levels. Attributes of a new component behavior can be determined by the way components are combined and by the components being combined [5].

Component Model

“A component model is the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types” [3]. A component model defines two things: first, how the single component is constructed and secondly, how components communicate with each other in component-based systems.

2.3 Model Driven Architecture

“MDA (Model Driven Architecture) is a standard proposed by OMG in order to separate application logic from the technology of implementation platform (J2EE, .NET, EJB, CORBA, Web-based platforms etc.)” [23]. MDA flourishes the idea in software engineering process to develop a model completely independent of technology PIM (Platform Independent Model) level, “to develop a model specific to the destination platform, called PSM (Platform Specific Model). Further, it is possible to generate from PSM compatible source code”. A PIM should be retargeted to different platforms.

MDE is derived from MDA. MDE can be considered as broader term that includes all models and modeling tasks needed to carry out a software project from beginning to end.

PIMs models do not have any dependence on the technical platforms. PIMs can represent different functional entities with their interactions, only expressed in term of business logic. PSMs are dependent and specific to the technical platform. PSMs models are useful to generate executable code of same technical platforms.

MDE chain proposed to match MDE principles and organization that enables to build a system from the requirements down to the code. According to the types of models being transformed, seven kinds of transformations are exists the combinations of these shown in Figure 2.4.
1. “The purpose of PIM-to-PIM transformations is to subtract or add some information about models, or to re-organize this information and to represent it in a different form” which is shown in Figure 2.5. These transformations are not always automated.

2. PIM to PSM: These transformations are performed when the PIMs are enough complete to be plunged towards a technical platform. “The operation that consists of adding proper information to a technical platform to allow code generation is PIM to PSM transformation” [24]. The target platform may be the .NET, XML etc. These transformations are always automated. Our thesis work is more focus on PIM to PIM transformation because we need target XML file for our implementation work.

3. PSM to PIM are rather difficult to implement, these transformations are carried out to build PIMs of what exists.

4. PSM to PSM: These transformations are carried out during the phases of optimization, deployment, or reconfiguration.

5. Step five indicates the transformation PSM non-executable to executable code.

The main goal of MDE is the shifting and we will use PIM to PSM in our solution.

---

**System**

“A system is a set of elements in interaction”. A system is generally represented by a set of different models, each one capturing some specific aspects of it. For example, world can be considered as system and map is an example of model. In this sense system indicates the reality.

---

**Model**

A model is a collection of concepts and relations. It is also an abstraction of reality. The basic principle of the MDE is “everything is model” [29]. Two relations are associated with this principle ‘represented by’ and ‘conforms to’ are shown in Figure 2.5. Model can be a copy of original but of smaller scale than original. Or model can be defined a simplified version of complex thing. According to the J.Rothenberg [30]:

“…. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.”

So, reality is represented by a model in an abstract way.
- **Meta-Model**

A meta-model is defined as sets of concepts and relations between the concepts used to describe a model, which is the reality for specific purpose. Then a model *conforms* to meta-model which specifies modeling structure [13].

A model $M_1$ is said to *conforms to* a given meta-model $M_2$ if it can be obtained through a legal collection of concepts as defined by $M_2$. This is the second relation of the basic principle shown in Figure 2.5 at level $M_2$.

- **Meta Meta-Model**

A meta-meta-model is a model that conforms to itself. “A meta-model is a model itself, conforming to a meta-model, called the meta-meta-model”.

“Each meta-model prescribes the abstract syntax for a language by means of elements and relations between them. So, a meta-model in turn has to be specified in a rigorous manner; this is done by means of meta-meta-model, which is intended to be the set of minimum concepts from which all the languages can be derived; thus, coherently, meta-meta-models are used to describe themselves” [16]. OMG has introduced layered architecture of meta-model which is presented in Figure 2.5.

![Layered architecture of meta-model](image)

*Figure 2.5: Layered architecture of meta-model [16]*

Except the top layer in Figure 2.5, each layer conforms above layer. $M_0$ is the lowest level and it represents a system. $M_1$ layer represents model which describe the aspect of the system. $M_2$ layer indicates meta-model and modeling language used for defining $M_1$ is specified in this layer. $M_3$ layer represents meta-meta-model. In MDA best-known meta-model is UML meta-model and Meta meta-model is MOF.

### 2.3.1 Model Transformation Types

In MDE, “transformation is a process that converts source model into a target model related to same system by means of a transformation specification” [15]. A standard transformation can be defined as a set of rules to map source to the target. Each rule describes how to transform source
instances to the identical target. Many languages are available to specify transformation. The transformation types are presented below:

- **Model-to-Model (M2M):** A M2M transformation creates its target as model which conforms to target meta-model. The M2M can act in-place or produce a new document as output. Transformations are executed by transformation engines. MOF, XML and Java source code file are considered as model.

  **M2M transformation engines are:** Atlas Transformation Language (ATL) and Query-View Transformation (QVT-o) operational and Declarative QVT (core and relational).

- **Text-to-Model (T2M):** T2M transformation is used for transforming textual representation to a graphical model. The textual representation must have syntax definition language usually with BNF (Backus-Naur Form). The graphical model should be a meta-model [17].

  **Technology for T2M:** Efttinge and Völter (2006) [32] presented the xText framework for T2M transformation in the context of the Eclipse Modeling Project (EMP). xText is used to automatically derive the Meta-model from the grammar. Then a textual representation of a model following this grammar can be parsed and the meta-model is automatically generated. Not too much information and techniques are available for T2M transformation.

- **Model-to-Text (M2T):** In M2T transformation model is taken as input and a stream of characters is returned without an explicit definition of the target meta-model. A M2T transformation is used for transforming visual representation of code (syntax). The syntax of the target language should be defined with meta-model of the graphical model [17]. M2T transformation is typically implemented through template and rewrite mechanisms.

  **Technologies support M2T transformation are:** xText (grammar based language), Acceleo, JET, Xpand, EMF text (template languages), Kermeta, MOF script (object oriented languages) etc. This thesis, as it will be explained in Chapter 5, makes use of Acceleo to build the solution to problem addressed. Therefore, we present Acceleo in the next section.

### 2.3.2 Acceleo Model Transformation Engine

Acceleo is a code generator implementation of the OMG's model-to-text (M2T) specification. It supports the developer with several properties that can be expected from a high quality code generator IDE: simple syntax, advanced tools, features on the JDT (Java Development Tools), and efficient code generation. Acceleo helps developers in handling life cycle of code generators. Acceleo was designed to work with any meta-model, implementing MOF and new UML version can be used with Acceleo [16]. Acceleo required minimum effort to implement and execute M2T transformation, Figure 2.6 presents the Acceleo text generated principle in which Model is transformed in to text by writing code for generating text in Acceleo template.
Example: Papyrus UML model transformation to XML by using Acceleo

In Acceleo transformation rules are written in templates that produce text-files (source code or documentation) after execution. Acceleo template is called mtl file. Guidelines for performing simple M2T transformation (Papyrus UML model to xml file) are listed below:

Figure 2.7 represents a Papyrus UML model which used for M2T transformation. Model contains a composite component named ‘Car’ and it contains two property rear and engine having Wheel and Engine component type respectively.

In Acceleo transformation structure consists of module (defines reusable transformation libraries), template (defines transformation itself). ‘GenerateXML.java’ is generated class that launches the transformation and generateXML.mtl is a template file but both having same name.

The transformation rules (.mtl file) for M2T is shown in Figure 2.8. “[module GenerateM2T_Acceleo (http://www.eclipse.org/uml2/3.0.0/UML)]]” indicates input meta-model URI.

Template structure starts with “[template public Gnerate_M2T (model: Model)]” and end with [/template]. Template can have a block group’s text producing expressions of a template. Under the template tag follow transformation rules are describe:

- This code will generate Test.xml file. First for block checks that if the component name is ‘Car’ then it is composite component.
- After getting composite component all elements contained by of composite component are taken such as property/component. A for block is used for specifying a text segment that needs to be processed repeatedly over a set of model elements. A for block is used for increment the number of components which shown in Figure 2.8.
The code is written for getting each property name and property type inside component tag as shown in Figure 2.8.

Inside a for block another for block is used which calculates the number of ports that a component have such as 

```plaintext
for(property:Property | component.eAllContents(Property))
```

Body in for block specifies the body of loop.

Inside Comment tag code for getting comment and body of comment contained by a component is provided.

In last for block under Connection tag code for getting connection between components is describe and also presented below.

```plaintext
[comment encoding = UTF-8 ]
[module GenerateM2T_ACCELED('http://www.eclipse.org/uml2/3.0.0/UML')]

[template public Generate_M2T(model : Model)]
[comment @main/]
[file ('Text.xml', false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
[for(component:Component | model.ownedElement->filter(Component))]  
  [if(component.name.contains('Car'))]
  <Components>
    <[Components compositeComponentName="[component.name/]">
      [for(property:Property | component.eAllContents(Property))]
      <Component componentName="[property.name/]">
        <Port port="[port.name/]">
          [for(port:Port | property.type->filter(Component).ownedPort)]
          <Port name="[port.name/]">
            [/for]
        </Port>
      </Component>
    [/for]
  </Components>
  [/if]
[/for]
[/Comment]
[/for]
[/Connections]
[/for]
[/Components>
[/if]
[/file]
[/template]
```

Figure 2.8: Code for generating XML (generateXML.mtl file)

Information enclosed in generated Test.xml file is shown in Figure 2.9 which describe following information.

- The composite component ‘Car’ is enclosed under Components tag
- Component named rear has the component type Wheel present inside component tag. Component tag contains Port tag which describes the port name ‘receive_force’.
- Another Component name engine is enclosed inside the component tag and inside the tag component type ‘Engine’ is displayed. Port contained by Engine component ‘provide_force’ shows under the tag of Port.
Similarly body of comment is presented under comment tag which contained by composite component.

Connections between components are indicated inside connector tag which provide the information about source component port name ‘provide_force’ and destination component port name ‘receive_force’ shown in Figure 2.9.

```xml
<Components>
  <Component compositeComponentName="Car">
    <Ports>
      <Port name="receive_force" />
    </Ports>
  </Component>
  <Component compositeComponentName="Engine">
    <Ports>
      <Port name="provide_force" />
    </Ports>
  </Component>
  <Comment body="rear wheel [2]" />
</Components>

Figure 2.9: generated xml File (Text.xml)

2.4 MDE within CHESS

As discussed in Section 1.1, in the framework of the CHESS project a component-based model-driven system development approach is being investigated. In the CHESS project model transformation is required for dependability analysis. The work-flow described in Figure 2.10 sketches the thesis work. The work-flow promoted by MDA, system designer develops Platform Independent Model (PIM), and "which is independent of the execution platform that will actually implement the system.

The development process is endured by various kinds of analyses for example dependability. These analyses allow assessing capability of design in system with respect to different aspects. According to MDE principles, analysis models are automatically obtained from high level model of system’s architecture. This approach prevent the user for providing details at analysis level of the system, without this approach providing details can be error-prone process. Then, analysis results are used to enhance the high level CHESS model that has triggered the analysis. CHESS project supports iterative development process in which system model is continuously updated based on analysis results. These analysis results are back propagated in the system model and can be used subsequent of analyses. CHESS ML (a high-level modeling language) is built from subsets of standard languages like UML, SysML and MARTE. The concept of failure instantiated into the CHESS Dependability Profile, which allows enriching a CHESS ML model with information related to dependability and safety.
Overview of CHESS Component Model

In CHESS component model packages are defined as the sets of concepts that are used to define a software component model in CHESS. CHESS model has various views named as Component View, Functional View, ExtraFunctional View, Deployment View, Analysis View and Requirement View. Details of CHESS component model views that are required in FPTC analysis are described below:

- **Component View**: This View is used for modeling the components according to CHESS component model definition. In CHESS model components are mapped with Component Type and Component Implementation stereotype. Realization is used between ComponentImplementation and ComponentType, it is taken from UML.

- **Functional View**: In this view, user specifies the functional attributes on model. In Functional View user can decorate ComponentImplementation and ComponentType with property, connector and ClientServer Ports. ClientServer Ports are used for defining interface and taken from ‘MARTE::GCM’. Components are connected through their interface and require interface to perform their function that is a ClientServerPort. A component can provide (through ‘provInterface’ attribute) or require interfaces (through ‘reqInterfaces’ attribute); it cannot provide and require interfaces at the same time.

- **ExtraFunctional View**: In this view, user specification addresses the extra-functional concerns of real-time dependability. In CHESS, this view imports read-only information from the functional view and extends with extra-functional information gives the guarantee that the definition of the functional entities is not altered. In ExtraFunctional View user can apply comment (FPTCSpecification and FPTC) taken from CHESS::Dependability::FailurePropagation and link.

- **Deployment View**: where the user specifies the allocation of software resources on hardware platform. Hardware entities and their applicable properties are expressed using MARTE stereotypes. Component instances derived from the ExtraFunctional View can be imported to specify their allocation to hardware entities [33].

- **Analysis View**: Analysis View is divided into two types of view name DependabilityAnalysis View and RtAnalysis View. But for FPTC we need only Dependability Analysis View.

  - **Dependability Analysis View**: where the user performs FailurePropagationAnalysis, which is fed with the user model taken from the ExtraFunctional View and the Deployment View. A FailurePropagationAnalysis gathers relevant quantitative and qualitative information for performing failure propagation analysis. The platform attribute (from GaAnalysisContext) has to turn the system to be analyzed, i.e. typically a root component owning component implementation instances [28].
2.5 Failure Behavior Analysis Techniques

The development of real-time component-based embedded dependable systems demands assurance that the system does not pose harm for people and the environment even if some system components fail. This assurance can be evaluated by using failure behavior analysis techniques. This section introduces some relevant failure behavior analysis techniques, a new compositional techniques such as Component Fault Trees (CFT), Failure Propagation Transformation Notation (FPTN), Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS), Fault Propagation and Transformation Calculus (FPTC), State-Event Fault Trees (SEFT), and Architecture Analysis and Description Language (AADL). The introduction mainly borrows from [19]. These techniques are briefly discussed below except FPTC. Since FPTC plays a significant role in thesis, it will be presented in more detail in section 2.6.

2.5.1 Component Fault Tree

Component Fault Trees (CFTs) is an approach to provide better support for hierarchical decomposition. It deals with disassociation between hierarchy of faults in a normal fault tree and the architectural hierarchy in the system components [21]. CFTs are modular version of traditional Fault Tree. CFTs use gates such as OR, AND, and M-out-of-N gates. CFTs also use input and output failure modes (depicted as \( \Delta \)), internal faults events (indicated as circles) as shown in Figure 2.11. CFTs are still logical structures linking output failures to input causes and can be analyzed (qualitatively and quantitatively) by using standard fault-tree algorithms [21]. The CFTs for different parts of the system can be developed separately, or can be stored as part of the component definition in a library of component types, facilitating a greater degree of reusability [21]. CFTs describe the features of partial fault tree for each output failure port. These faults trees can be calculated as a function of internal fault events and input failure ports. For identifying possible failure propagation between components, dependencies are examined. Also, input and output failure ports are matched on basis of their names and types. Example of CFT (Fire Alarm System) is presented in Figure 2.11. CFTs have been used in many industrial projects and supported by ESSaReL window-based tool (Siemens Corporate Technology) for graphical specification and efficient evaluation. CFTs are based on matching of incoming and outgoing failures ports with limited support of architectural dependencies. CFTs have manual tool guided generation of error models for hierarchal components [19].

![Figure 2.11: Example of CFT (Component Fault Tree) [19]](image)

Fire Alarm system contains one or more smoke sensors, a set of sprinkler actuators, a software-based control unit including its executing hardware platform, and a watchdog component that tests the software-based control unit at regular intervals. Once a fire occurs a smoke sensor detects fire and control unit activates set of sprinkling actuators, for more details about fire alarm system reads [19]. In the example of Figure 2.11, the incoming failures are Smoke_detected.omission, Smoke_detected.commission. Sprinkling.omission and Sprinkling.commission are outgoing failures.
2.5.2 Failure Propagation Transformation Notation

Failure Propagation Transformation Notation (FPTN) is a simple and modular notation technique for specification of failure behavior of components. The basic entity of FPTN is FPTN-module shown in Figure 2.12. FPTC module contains a set of standardization sections; first section represents a name (identifier ID) and criticality level (SIL safety integrity level). Second section represents architectural element (alarm unit) defines propagation and transformation of failure, generation and detection of failure in a component [39]. Each module is represented by a box. A module can have more than one sub modules. Set of inputs and outputs can be represented by set of logical equations. If FPTN-module contains other FPTN-modules then incoming failure of one module is connected to the outgoing failure of other module.

In Figure 2.12 inputs has to read in the following way: Smoke_detected:tl, Somke_detected:o, and Smoke_detected:c are incoming failures. Sprinkling:o and Sprinkling:c are outgoing failures. The transformation and propagation are described inside the module with a set of equations. The statements in Figure 2.12 indicate that commission failure propagated as it is but omission failure can cause too late.

FPTN is designed along with system design so, that system model evolves. The information gathered from analysis of FPTN model can be used to detect errors and problems in system design [19].

2.5.3 Hierarchically Performed Hazard Origin and Propagation Studies

Hierarchical Performed Hazard Origin and Propagation Studies (HiP-Hop) technique uses a text format called Tabular Failure Annotation (TFA) to specify the failure behavior of data architecture in a commercial tool set of environments, such as Matlab-Simulink or Simulation X. Failure annotations at component level contain sets of logical expressions, which indicate how output failures of each component can be caused by internal malfunctions and deviations of the component inputs. Failure Mode and Effects Analysis called Interface Focused FMEA (IFFMEA) is used as a means of deriving such failure annotations. Based on these annotations several analysis techniques are suggested. Analysts apply this technique on components to identify plausible output failures such as the omission, commission, value or timing (early, late) failures at each output and then to determine the local causes of such events as combinations of internal component malfunctions and input failures [39]. As an example fault trees in Fault Tree+ format can be generated and analyzed for minimal cut sets to identify critical points of failures. Moreover, FMEA tables can generated and analysis through minimum cut-sets. In practice, HiP-Hops has been successfully applied in many
complex real-world systems in companies such as Volvo and Daimler Chrysler [19]. An example tabular failure annotation of fire alarm system (control unit) is presented in Figure 2.13.

<table>
<thead>
<tr>
<th>Output Failure Mode</th>
<th>Description</th>
<th>Input Deviation Logic</th>
<th>Component Malfunction Logic</th>
<th>$\lambda$ (Failure/hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprinkling</td>
<td>Commission Failure: control unit activities the Sprinkling when it is not needed</td>
<td>Smoke_detected_c</td>
<td>Software Fault or Controller Malfunction</td>
<td>$1.0\times10^{-8}$</td>
</tr>
</tbody>
</table>

**Figure 2.13: Example of HiP-Hops Tabular Failure Annotation [19]**

### 2.5.4 State-Event Fault Tree

State-Event Fault Trees (SEFTs) are formalism that differentiates events (sudden phenomena, in particular state transitions) from states (that last over a period of time) [19]. Syntactically, SEFTs are a visual formalism that stretches out CFTs with probabilistic finite state models. In SEFTs states are presented by rounded rectangles and events as solid bars which are shown in Figure 2.14. In SEFT transition can be triggered informally by another event. Ports are used to depict relationship between architectural elements and its environment. State port along with standard event-based failure port can be used to examine architectural elements in a specific state such as output port state “sprinkling” in Figure 2.14. SEFTs are not supported identification of error model in basic architectural components. Manual tool guided for generation of error models are supported by SEFTs but generation of Standard Fault Trees and FMEA are not currently supported by SEFTs [19]. SEFT is supported by ESSaReL (Windows-based with drag and drop GUI) tool.

**Figure 2.14: Example of State-Event Fault Tree [19]**

### 2.5.5 Architecture Analysis and Description Language

In Architectural Analysis and Description Language (AADL) components are specified through component types and component implementations. Component type is defined by component’s interface through particular interaction points such as event, data and data event port. A component type can be divided into three component categories named as application software, execution hardware and composite system [19]. In application software, AADL component can be a process, thread, data, thread group or subprogram. The execution hardware can be further divided into memory, processor, bus and other devices such as sensors and actuators components. In AADL component implementation describes internal structure of the component. Figure 2.15 presents example of component implementation of fire alarm system. Subcomponent describes those components that are in system. Connections define connected input and output ports which are presented in Figure 2.15. Finally the section properties specified that process AU (alarm unit) is executed on HW hardware, for detailed information read [19].

---

21
AADL’s Error Annex gives “capability to annotate AADL components with dependability related information called AADL error models” [19]. Like AADL architectural models, error models have two levels named as error type level and instance/implementation level. At type level an error model describes a set of error states which can be error free. An error model implementation defines transition of errors between states. An error model of AADL can be stored in a library. Resultantly an AADL error model defines error behavior of similar components. For AADL error propagations are done through name matching of incoming and outgoing error with already defined dependency rules between AADL components. Identification of hazard conditions and safety is not supported by AADL. AADL with Error Annex is supported by OSATE (Eclipse plug-in) tool and generation of FEMA (Failure Mode and Effect Analysis) tables is currently not supported by AADL [19].

Figure 2.15: Example of AADL Fire Alarm System [19]

2.6 Failure Propagation and Transformation Calculus

FPTC is a technique used for automatically calculating the failure behavior of the whole system from the failure behaviors of its individual components [6]. Understanding the behavior of the system in the presence of failures is crucial.

- To evaluate the risks.
- To be able to plan counter-measures within a system to reduced/mitigate the failures.

2.6.1 Failure Behavior of a single component

To use FPTC technique each component must be analyzed in isolation from the rest of the system. The behavior of each individual component in response to potential failure stimuli must be analyzed. In response to its input, a component reacts in one of the following ways:

**Sink:** Sink behavior means that component has the ability to detect and correct, received failure which is produced elsewhere in the system.

**Source:** A component behaves like a source when it generates a failure by itself without getting any failure as an input.
**Propagation:** A component receives a failure input and the type of failure on the output remains the same.

**Transformation:** Transformation behavior means that a component changes the nature of failure from one type to another.

### 2.6.2 Types of failure

Before introducing the failure types used in FPTC, we recall what has to be intended with the term “failure”. As discussed in section 2.1 the failure occurs when delivered service deviates from correct service.

“In the framework of component-based systems, a component can be considered as a system when studied in isolation and as a sub-system when the whole system is considered” [10]. Therefore we have a confidence on when a single component’s input-output behavior is considered the term ‘failure’ of a component is correct. Also it is enough to denote deviation of a system and the introduction of other dependability threats (fault and error) is not necessary. When a system (Z) is considered as whole and ‘X and Y’ are the subsystems and a failure in a subsystem represents a fault elsewhere, which is shown in Figure 2.16. For example, in a system obtained by chaining two components X and Y. “The failure generated by X, if not handled, propagates itself within Y and beyond Y’s boundary until it reaches and handled, propagates itself within Y and beyond Y’s boundary until it reaches and exits the system’s boundary”. In FPTC we used term failure as input-output behavior (fault in input comes from generated elsewhere) [10].

![Figure 2.16: Component Based-System](image)

We discussed earlier classification of failure [9], but now we consider those failure types which are defined by HAZOP/SHARD and have been integrated in FPTC. They identified types of failure through a set of guidewords.

- **Value failure**: Value failure means that the value of the result provided by a component is deviated from the expected range; value Subtle and value Coarse are subtypes of type value failure.
  - **Value Subtle** means a value which is difficult to perceive or understand. “The output deviates from the expected range of values in an undetectable way (by the user)” [10].
  - **Value Coarse** means a value that can be clearly perceived by the user [9, 22]. “The output deviates from the expected range of values in a detectable way (by the user)” [10].

- **Timing failure**: A timing failure represents a failure that is delivered outside its specified sets of time. Timing failure is divided into two subtypes named as early and late failure.
  - **Late** failure means that a component gives the result late with respect expected time interval.
  - **Early** means that a component provides result before from its expected time interval.
Sequence / Provision failure: A sequence failure mode represents a way of assertions on the (possibly infinite) sequence of failures delivered by a component and it also related with timing pattern. Sequence failure can be divided into two subtypes named as Omission and Commission.

- Omission means that a component generates the result with an infinite late (time interval) which deviate from expectation.
- Commission means that a component generates an output but it is not expected from the component [9, 22].

2.6.3 FPTC Syntax and Semantics

Syntax

FPTC syntax allows user to specify the behaviours of the components. This behaviour consists of a collection of propagation and transformation expressions. Expression consists of two parts: left hand side part (LHS) and right hand side part (RHS). Left hand side of expression indicates input behaviour of a component and right hand side of expression shows output behaviour of component [10, 23].

The formal syntax of FPTC according to EBNF is presented below [7]:

\[
\text{behaviour} = \text{expression} + \\
\text{expression} = \text{tuple} \rightarrow \text{tuple} \\
\text{tuple} = \text{one} \\
\text{one} = '{\ast}' \quad \text{(no failure)} \\
\text{one} = '{\_}' \quad \text{(wildcard)} \\
\text{one} = '{\text{alphachar}} \quad \text{(variable)} \\
\text{one} = '{\text{fault}} \quad \text{(explicit fault)} \\
\text{one} = '{\text{fault}}, '{\text{fault}} + '{\text{}} \quad \text{(set of fault)} \\
\text{fault} = \text{‘Early’ | ‘Late’ | ‘Value’ | ‘Omission’ | ‘Commission’ | ……}
\]

In FPTC wildcards and variables are used for reducing specification burden. Using wildcard in an expression means that user does not need to care about failure type in specific position or type of input behavior cannot play decisive role to generate output behavior [7]. Wildcard is denoted by ‘_’. Variable binds input token from that it is matched and propagate it as output. Variable should appear on right hand side of an expression. A variable should not be a fault type and any value given on input token other than fault type is treated as a variable. Normal behavior of a component means that ‘no failure’ and it is denoted by ‘\ast’. The four possible FPTC behaviors of component are presented below:

\[
\text{late} \rightarrow \ast \quad \text{(Sink)} \\
\ast \rightarrow \text{early} \quad \text{(Source)} \\
\text{late} \rightarrow \text{late} \quad \text{(Propagate)} \\
\text{omission} \rightarrow \text{value Subtle} \quad \text{(Transform)}
\]

24
Semantics

The algorithm for calculating failure behavior defined by Wallace [7] can be described informally as follows:

1. **Analysis of failure behavior of the components**: The failure response of a component to its input is analyzed in isolation from the rest of the system. From this analysis it is possible to settle if a component: propagates received input failure as it is, transforms input failure to another type of failure as its output, generates a failure by itself in the absence of input failure and avoid failure to propagate as output behaves like a sink [10].

2. **Specification of the component’s behavior**: The component’s behavior analyzed at step 1 must be specified as a collection of propagation or transformation expressions, using FPTC syntax.

3. **Calculate failure behavior of the whole system**: The inter-dependent components are considered as a token (failure)-passing network. Calculate a fixed-point. To guarantee convergence, at each node of the network it must be ensured that exactly one expression must be match with input failure behavior.

Let’s suppose a simple system which consists of two components as presented in Figure 2.17. For calculating failure behaviour of the system, FPTC technique is performed on these components. Initially it is assumed that no failure is injected on a component before applying propagation rules.

Component X receives ‘*’ as an input and LHS expression is compared with input failure. Since LHS of expression is matched then late is propagated as output failure from component X. Component Y receives late as an input failure. Component Y transforms late failure into early failure and result of FPTC analysis is shown in Figure 2.18.
2.6.4 Benefits of FPTC

The benefits of FTPC technique are listed below:

- Analyzing failure behavior of individual component (means that bottom to top) is easier rather than computing behavior of an entire system.
- Manually calculating failure behavior of system is time consuming so, FPTC is less expensive due to its automated calculation. But, CFT and SEFT are manual tool guided techniques.
- If failure behavior of system is calculated for its components then impact of change component on the system is easily determined and enhancement of the system will be cheaper.
- FPTC focus on the qualitative evaluation but other techniques focus on quantitative evolution.
Chapter 3

PROBLEM FORMULATION AND ANALYSIS

This chapter formulates the problem addressed in this thesis and organized as follow: Section 3.1 identifies and describes the main problem which is going to be solved and section 3.2 highlights the problem analysis.

3.1 Problem Formulation

Currently, the development of real-time component-based embedded software systems is not supported by well-integrated means allowing for quality evaluation and design support within a development process.

Quality and in particular dependability is crucial for such systems which very often can be classified as mission or safety critical systems. These systems are expected to comply with the standards which are being introduced (i.e. Functional Safety Standard IEC 61508, and Quality System Standard ISO 9001 etc.). Techniques to provide evidence that the systems are compliant with the standards are essential. These techniques to be effective have to be coherently integrated within a development process. In the context of a model-driven and component-based development approach, this thesis investigates an integrated framework to evaluate the dependability of high integrity of real-time component-based embedded systems.

More specifically, as mentioned in the introduction chapter, the context of this thesis is the CHESS project, which aims at providing a tool-supported component-based and model-driven-based methodology to engineer high integrity real-time component-based dependable embedded systems.

The achievement of an integrated framework for the evaluation of dependability is the main goal of this thesis in order to provide a contribution to the main problem which is the lack of adequate means, discussed above. To achieve our main goal, the main problem is divided into two main sub-problems, which are given in the following section.

3.2 Problem Analysis

The problem concerning the absence of well-integrated means to support the dependability evaluation of high integrity real-time component-based embedded systems can be decomposed into two main sub-problems.

- **SP1: Which technique?**
  
  High integrity real-time component-based embedded systems, as mentioned in the previous section, require high quality and in particular dependability. To evaluate how dependable a system is, adequate techniques are needed. Different techniques are available to analyze system’s dependability: some of these techniques provide a qualitative evaluation; some others provide a quantitative evaluation; some are modular; some allow for automatic evaluation and some techniques are manual. In front of the various techniques available, therefore, to perform a careful selection of a potential and adequate technique the current state of the art has to be investigated and analyzed. Adequate criteria have to be identified to analyze the techniques.

  For instance, since the focus is on dependability, an important criterion is granularity of the analysis. Which dependability threats can be considered?
Another important criterion is the usefulness of the analysis in supporting design decisions? In particular, the analysis should be useful to answer the following questions:

**Which Component Composition?**
Component composition requires evaluating the behavior of components, specifications of a component’s behavior, describing correct usage of the services provided by a component, usage of required services by a component as well as communication among components. Different types of components have different properties and different composition principles exists which results in various problem.

**Dependability Means?**
Which is the dependability means needed to avoid failures at end of system? What are the components to be replaced?

- **SP2: How should the technique is integrated?**
  Since the technique is expected to be integrated within the CHESS development process, which as said is a model-driven based one, it is crucial to understand which MDE technology should be used.
  Moreover, to perform the necessary transformations, the following questions require an answer:
  - which concepts are relevant for the analysis and need to me transformed?
  - which information should be back-propagated?
  - what are the classes and methods should we have to create to perform FPTC analysis?
Chapter 4

METHOD

In order to provide solution of the problem it is better to understand the presented method and reasons why we select these solutions to solve the problems. Section 4.1 briefly explains way of selection of FPTC analysis technique for analyzing system dependability attributes, Section 4.2 highlights the ambiguity found in FPTC rules and also provides a solution to remove the exits ambiguities. Section 4.3 presents the way of integration of FPTC plug-in in CHESS tool-set. In section 4.4 we explain way of selection of Acceleo transformation engine for performing M2T transformation.

4.1 Selection of Proper Failure Behavior Analysis Technique

We select FPTC technique to analyze the failure behavior of system after making the comparison of different analysis techniques which are described in section 2.5. The comparison between FPTC and other failure analysis techniques are carried out in three domains (i.e. modeling support, process support and tool-support) for better understanding of advantages and disadvantages of these techniques.

**Modeling support:** It relates ability of system safety engineers in specifying the failure behavior of an architectural element and the failure propagation between dependent architectural elements [19]. The main goal of all these languages is to characterize the failure logic of individual components. The models FPTN, FPTC, HiP-HOPS, and CFT only represent event-based failure behavior, whereas SEFT and AADL’s Error Annex describe both states-based and event-based behavior. CFT and SEFT represent only graphical modeling but FPTN, FPTC and AADL represents graphical as well textual modeling. Hip-HOPS represent the textual failure mode.

**Process support:** Process support consists of steps that are needed to perform safety evaluations at an architectural level. First step is hazard conditions and safety requirements have to be identified and formally specified. Second step is modeling of architectural elements within failure/error model. Regarding identification and specification of hazard conditions and safety requirements, only CFTs and HiP-HOPS describe some simple methodological support with the Software Hazard Analysis and Resolution in Design (SHARD) and Functional Failure Analysis (FFA) [19]; but AADL, FPTN, FPTC and SEFT do not support it. AADL and FPTC both support cyclic loop but CFT and SEFT are not support cyclic loop. AADL’s Error Annex and Hip-HOPS provides good support for modeling architecture specification including architectural dependencies. But other modeling formulism are general failure propagation languages and do not target the specific architecture.

**Tool support:** Good theoretical and failure behavior analysis techniques are not useful if they are not supported by tool. Additionally, if we consider complex systems, when manually performing analysis there are many chances of errors. AADL with error Annex, Hip-HOPS and FPTC support for automatic generation of error models for hierarchical software components but CFT, SEFT are manually tool-supported.

We conclude that FPTC technique is useful to analyze failure behavior of high integrity real-time component-based dependable embedded software systems because; it automatically calculates the behavior of the entire system from its building components. Therefore, less time and cost is required while using FPTC technique for providing analysis solution to industries. Some techniques like FTA start the analysis at system level and work backward to identify causes at component level. But, FPTC start the analysis at component level and it keeps the model and reality synchronized, and also localizes the effect of any changes. But in other technique a later change to one component might alter
not only the types of failure generated, but also where those failures are propagated to, thus it become necessary some non-local re-working like in FPTN. FPTC is more robust than other techniques, in requiring that each component be analyzed in isolation for all possible failure responses, not just those in the currently-known context.

4.2 Understanding and Comparison of FPTC rules defined by different authors

After studying and understanding FPTC technique we indentified that several inconsistencies exist in FPTC rules/syntax; because different authors define different terms and rules regarding calculation of failure behavior of the system. This issue is solved by making comparison between rules defining by different authors.

These differences are presented below:

1) Wildcards and variables both are used to combine similar patterns in to one expression Source Forge [8]. According to Wallace (2005) a wildcard pattern means do not care about what type of failure (or non-failure) token occurs in a certain position and a variable means something similar, except that its value is bound for use on the RHS of the expression. Wildcard cannot be used on RHS of an expression. For example (early, _) \(\rightarrow\) (early, commission) But according meta-model which is presented by R.F. Paige et al [6] wildcard can be used on RHS of expression.

2) If overlapping of expression is occurred then most specific expression should be selected. Consider two expressions that are written below:

\[
\begin{align*}
(\text{late, } \_ ) & \rightarrow (\text{omission, value Subtle}) \\
(\text{late, commission}) & \rightarrow (*, *)
\end{align*}
\]

Suppose that late is presented as first input failure and commission is given as second input failure, in this case more specific expression is selected. Second expression is more specific as compared to first one. This FPTC rule is presented in almost all research material but method for calculating specificity is only described in Source Forge [8]. The formula for calculating specificity is given below:

\[
\text{Specificity} = \text{LHS\_cardinality} - (\text{no. of wildcards on LHS} + \text{no. of variables on LHS})
\]

3) According to Wallace (2005) LHS and RHS side of an expression must at least one, but according to meta-model presented by Richard F. Paige et al (2009) [6] indicates that cardinality of LHS and RHS expression is [0…*]. It means that according to meta-model expression/ transformation rule may have no LHS and RHS.

4) Consider an example of Figure 4.1; a Component Z has two incoming connections. In case of multiple incoming connections both expressions should be applied. First expression may be selected because ‘*’ is occurring as first input failure and late is present as second input failure. Similarly other one is selected due to commission is occurring on first failure and ‘*’ is at second place.
According to Source Forge [8], if a component has multiple incoming connections, expression may be applicable for each combination of inputs. The cross product of these inputs is calculated, for producing a set of expression. In a component, failure behavior at most one expression is applicable for each permutation [8]. Information found from other resources could not clarify the handling of multiple incoming connections.

5) Handling the sets of input failure is clearly defined by Source Forge [8] but material found in others resources is not concise enough. Consider an example

\[(*, \text{early}) \rightarrow (\text{late})\]

\[(*, \text{omission}) \rightarrow (\text{late})\]

For avoiding duplication in an expression, above expression can be written as

\[(*, \{\text{early, omission}\}) \rightarrow (\text{late})\]

After this comparison we concluded that wildcard is not used on the RHS of the expression, because using of wildcard on RHS of expression is pessimistic. For calculating specificity of wildcards and variables we should use the above algorithm in our implementation. Each component must have at least one expression (rule). For handling multiple incoming inputs we should calculate combination of input failure regarding each input port.

### 4.3 Integration of CHESS-FPTC Plug-in with CHESS Tool-Set

To well integrate the FPTC plug-in with CHESS tool-set (called as CHESS-FTPC plug-in), following steps has to be performed:

1. Understanding of CHESS-ML language
2. Identify how to extend CHESS-ML to cover FPTC concepts.
3. Identify available and missing features in CHESS tool-set which are necessary for implantation of FPTC with-in CHESS.
4. For implementation of the FPTC technique following steps has to be considered:
   - Creation of separate java classes for patterns matching
   - Applying transformation and propagation rules
   - Writing the method for calculating specificity of multiple incoming failures should be done.
4.4 Selection of Transformation Technology

We have to select transformation engine for providing solution of our main problem and it is the objective of the thesis to used MDE approach to engineer high integrity real-time component-based dependable embedded systems. We have to select proper technique which supports Eclipse plug-in and can be integrated with CHESS tool-set.

4.4.1 Why we select Acceleo for M2T

We select Acceleo as transformation technique to perform model-to-text transformation because it follows MDA principles including increased reliability and quality of code. Acceleo required minimum effort to implement and execute M2T transformation quickly. Some features of Acceleo are:

- Open source
- PIM support
- Quick implementation
- Fully integrated with Eclipse platform as Eclipse plug-ins
- Provide better usability and maintainability
- Compatibility with all editors using EMF meta-models

For M2T transformation Acceleo is suitable technology to support code generation as compare to Xpand and JET in terms of interoperability and scalability. JET has some extensibility, customizability but it is not suitable because of because of the lack of diversity in its various communities and, as a result, uncertain long-term sustainability [31]. For the same reasons, Acceleo is the preferred choice for M2T transformations in terms usability and maintainability is better than for Java APIs.

Format of Input xml file: After selecting Acceleo technique for M2T transformation, another issue has to resolve that is format of target file (input xml) file which must be appropriate for calculating behavior of the system. The information which is required in input XML for performing FPTC analysis is described as:

- Name of the system on which analysis should be perform, input port of the system and as well as input values on the input port, output port on which output value should be propagated.
- Information about all sub-components which contained by the system/composite component.
- Name of sub-components, sub-component ids, name of input/output ports, ports ids of all sub-components.
- Transformation rules/ expressions which are contain by the each component, and these expressions must be separated as LHS pattern and RHS patterns.
- Connection between each port of all components which indicates the link between output ports of one component to input port of other component.
- Through the connections between components failure in propagated from one component to another component.

Input xml file should have contain all necessary information for performing FPTC such as input/output ports of components, connection between these ports, transformation rules of every
component in system. Input xml file should also have input failure type of system as well as all of its components.

4.4.2 Back Propagation

Back propagation is necessary for displaying the analysis result of the system through which user can track output failure of each component and reduced factor of propagating the failure(s) on output port of the system. For performing back propagation, each out-going port of a component must have annotated comment having FPTC specification stereotype.
Chapter 5

SOLUTION

Chapter 5 is organized as follow: Section 5.1 provides modified version of FPTC syntax and rules after removing ambiguities. Section 5.2 describes meta-modeling concepts to extend CHESS-ML. Proposed design explain in section 5.3, section 5.4 presents implementation of FPTC as plug-in and integration of it with CHESS tool-set.

5.1 Proposed FPTC

As we discussed in chapter 4, inconsistence exists in FPTC syntax and rules. Therefore, after understanding of FPTC technique and making comparison of different available literature, we are able to remove the inconsistencies that exist in FPTC syntax. So, we proposed the following solution regarding FPTC syntax.

5.1.1 Multiple incoming failure on Input Ports

If multiple input failure are coming on input port of the component, the combination of these input failure are calculated as discussed in section 4.2.

5.1.2 No Pattern Matching

If no input failure is matched to the LHS of patterns (rules) of a component then according to Wallace two possibilities exist:

- First option: component behaves like a sink and no failure is propagated at output port of the component.
- Second option: Input failure is propagated as it is on output port of that component.

We use second option in our implementation of FPTC analysis because it is more appropriate solution then first.

5.1.3 Usage of Wildcard

We have not used wildcard on RHS of expression (rule) in our solution according to comparison proposed in section 4.2.1. Instead of using ‘_’ (underscore) symbol we decided to use Wildcard term as it is.

5.1.4 Specificity of Wildcards and Variables

During pattern matching we select that expression which is more specific. For calculating specificity of wildcards and variables we use the algorithm in our source code which has been described in section 4.2. Source code regarding specificity will describe in Implementation section 5.4.

5.1.5 Modified Syntax of FPTC

We modified the syntax of FPTC technique proposed by Wallace (2005) and remove the inconsistencies. To avoid creating ambiguity we used term failure instead of term fault. The term
‘Value’ is used for a failure but we sub-divided Value into ‘valueSubtle’ and ‘valueCoarse’. After removing the ambiguities, modified syntax of FPTC is presented below:

\[
\text{behaviour} = \text{expression} + \\
\text{expression} = \text{LHS} \rightarrow \text{RHS} \\
\text{RHS} = \text{portname} \cdot \text{bR} | (\text{portname} \cdot \text{bR} (, \text{portname} \cdot \text{bR} + ) ) \\
\text{LHS} = \text{portname} \cdot \text{bL} | (\text{portname} \cdot \text{bL} (, \text{portname} \cdot \text{bL} + ) ) \\
\text{bR} = \text{noFailure} | \text{variable} | \text{failure} \\
\text{bL} = \text{wildcard} | \text{bR} \\
\text{failure} = \text{early} | \text{late} | \text{commission} | \text{omission} | \text{valueSubtle} | \text{valueCoarse} \\
\]

5.2 Meta-Modeling Concepts Provided to Extends CHESS ML

CHESS Dependability Profile allows enrich CHESS ML model which is related to safety and dependability. The definition of CHESS dependability profile begins from identification of dependability features/properties that UML profile should address which relates dependability analysis of hardware and software systems. Language extensions (like the UML profiles) were introduced and utilized to capture the required extra-functional concerns. Language extensions are used to identify the component types and assign local dependability parameters to hardware and software artifacts in the model.

Over-view of the CHESS elements which are necessary for FPTC are presented below:

5.2.1 Component Model

The component model package defines a set of concepts that can be used to model CHESS software component model.

To perform FPTC analysis Components are decorated with two types of stereotype **ComponentType** (it maps the component type stereo of the CHESS component model) and **ComponentImplementation** (maps the component implementation stereo type of the CHESS component model). The behaviors of a component (rules) are described in ‘ComponentImplementation’. Relization is the relation that is taken from ComponentImplementation to ComponentType.

5.2.2 Components

Components are the basic block of the system. In CHESS components are classified with respect to different characteristics, which may result in different behaviors with respect to dependability properties. For example, components can be classified as composite component (Components may be composed of subcomponents, with multiple levels of depth.), software components and hardware components, stateful components (have the internal state) and stateless components (do not have internal state and in these components occurrence of faults immediately leads to failure). For dependability point of view, component has affected by failure [27].

5.2.3 Interfaces

Components are connected through their interface (specifying in a ComponentView of CHESS), and they communicate in order to implement the system’s functionality. Components have required and provided interface. Interface has operations through which we identified input and output port of the component. When a component uses the functionality provided by another
component, propagation of failure may occur. This kind of relationships is then necessary to derive basic failure propagation paths between interacting components [27].

5.2.4 Ports

CHESS component model have different types for hardware and software system. Software system has Client Server Port and hardware system contains Data Flow Port. Upon activation of ExtraFunctionalView we can apply the ports on a component.

- **Client Server Port**: For performing FPTC analysis on software level we use the Client Server Port. `ClientServerPort` has attributes named as `reqInterface` and `proInterface` for specifying required and provided interface.
- **Data Flow Port**: We used data flow port at hardware level. It allows the information flow between hardware components such as its direction.

5.2.5 Comments

- **fPTC**: stereotype is used for providing set of expression/rule for a component. `fPTC` stereotype has three attributes that are `inputPort` (have the information of input port of annotated element), `outputPort` (have the output port of annotated element means component), `fptc` (contain the expression as a string). For this stereotype annotated element is `ComponentImplementation`.
- **FPTCSpecification**: stereotype allows users to defined input failure on a component which is specifying in a `ComponentView`. It is used in comment and comment is connected to input port of the component. Two attributes of this stereotype are required of FPTC analysis named `failure` (in failure user select different type of failures which are relevant to FPTC analysis), `partWithPort` attribute is used to identify that the following comment is attached with which port.
- **Assign**: stereotype specifies in DeploymentView of CHESS and it is used for allocation of input failure on hardware system within a comment. It has two important attribute which are used for FPTC analysis on hardware level. Attribute named “from:Element” is taken a value from softwaresystem and “to:Element” attribute as allocate a value on port of the hardware system.

5.2.6 Selection of Platforms

- **failurePropagationAnalysis**: It allows to specify the platform instance specification for running e FPTC analysis. `failurePropagationAnalysis` specify in DependabilityAnalysisView which is a specialization of MARTE::GQAM::GaAnalysisContext. It has the attribute `Platform:GaResourecsPlatform` for selecting the instance specification of the system on which analysis has to be performed.

5.2.7 Software/Hardware Systems

- **CHGaResourcePlatform**: Extends the MARTE::GQAM::GaResourcePlatform and applies on hardware and software system. `CHGaResourcePlatform` stereotype specify in both ComponentView and DeploymentView. In ComponentView it used for decorating software system and in DeploymentView it is used for decorating hardware system.
5.3 Proposed Design

Our design concentration is not just limited to implementation of failure propagation analysis; but, this technique must be compliant with CHESS methodologies and supported in CHESS platform.

5.3.1 Design of CHESS-FPTC Plug-in

CHESS-FPTC plug-in is classified into four packages as shown in Figure 5.1. Classes enclosed in ‘se.mdh.chess.fptc.analysis.impl’ and ‘se.mdh.chess.fptc.analysis.launch.xml’ packages are reused after modification of previous source code; as well as adding new methods to make it compliant with CHESS tool-set [40].

- **se.mdh.chess.fptc.analysis.launch**
  - **Activate Plug-in:** Activator is a class which is loaded initially and it provides information to Eclipse Run-time that plug-in is associated with Eclipse Platform.
  - **Enable/Disable Call Command:** Classes named as ‘FPTCSourceProvider’ and ‘SelectionListenerForFPTC’ are used for enable/disable the command for calling FPTC Analysis (call is only enabled in dependability analysis view).
  - **Analysis Command:** ‘FPTCAnalysisCommand’ class is activated upon selecting ‘FPTC’ call command from eclipse run-time configuration. In this class, IProgressMonitor is used for calling activities (CallBuildInstances, RunTransformations, PerformAnalysis, BackPropagation) which are involved in performing FPTC analysis.

- **se.mdh.chess.fptc.analysis.impl**
  - **Read Input XML File:** ‘InputXMLFileReader’ class is concerned with reading generated XML file. Generated XML file contains information about composite component, inner child components (their transformation rules), data ports and connections. This information is stored in data structures in classes (DataPort, FaultPattern, SoftwareComponent and TransformationRule) which are enclosed in se.mdh.chess.fptc.analysis.launch.xml package.
  - **Propagation:** ‘ApplyPropagation’ class is mainly concerned with computing input combinations, pattern matching and applying more specific transformation rule. This class collects information from classes (DataPort, FaultPattern, InputFault, InputFaultCombination, SoftwareComponent and TransformationRule) which are enclosed in se.mdh.chess.fptc.launch.xml package.
  - **Write Output XML File:** ‘OutputXMLFileWriter’ class is used for collecting analysis results and enclose them in an XML file.

- **se.mdh.chess.fptc.analysis.launch.xml** package contains multiple classes (DataPort, FaultPattern, InputFault, InputFaultCombination, InputPortFaultVal, SoftwareComponent...
and TransformationRule) for storing and collecting information from data structures (List, Set and Map).

- **se.mdh.chess.fptc.analysis.transformation** package contains ACCELEO templates and java classes for performing M2T transformation.

  - **Transformation:** ACCELEO templates (.mtl) are used for writing M2L code. Separate ACCELEO template files are used for writing M2L code for software system (XML_SW.mtl) and hardware system (XML_HW.mtl). Main ACCELEO template is created for identifying whether selected platform belongs to XML_SW (Component View) or XML_HW (Deployment View) and further activates corresponding template.

![Figure 5.1: CHESS-FPTC Plug-in](image)

### 5.3.2 Integration of FPTC Plug-in in CHESS Platform

In CHESS model, FPTC plug-in interacts with ‘it.unipd.chess’, ‘it.unipd.chess.editor’, ‘it.unipd.chess.chessmlprofile’, and ‘org.chess.commands’ plug-ins as shown in Figure 5.2.
39

Figure 5.2: Integration of FPTC Plug-in in CHESS Platform

- **org.chess.commands** plug-in contains CHESS commands. Failure propagation analysis commands are enclosed in this plug-in.

- **it.unipd.chess.chessmlprofile** plug-in contains meta-models for CHESS and packages for failure propagation analysis; which contains implementation details for stereotypes and contains list of failures used in failure propagation analysis. This plug-in contributes in decorating CHESS modeling elements with failure propagation stereotypes.

- **it.unipd.chess.editor** plug-in contains implementation details for CHESS editor. Failure propagation analysis must be applicable at CHESS editor.

- **it.unipd.chess** plug-in contains implementation details for CHESS Project. This plug-in contributes in creating CHESS Project, Adding CHESS Nature to projects and applying CHESS Views and Profiles. This plug-in is useful for identifying current CHESS View.

5.4 Implementation

To start implementation of FPTC Plug-in in CHESS tool-set, first activity is the modeling of example system. Second activity is created for Activate/deactivate the call command since, according to the CHESS methodology user can only run the analysis from dependability analysis view. Launch FPTC analysis activity call the FPTC analysis. According to CHESS methodology analysis is performed on component’s instances; we create Call Build Instance Command activity for generating instances of components. Generate an Input XML activity performs to generation of input XML file from model by using ACCELEO M2T transformation techniques.

The Input XML file becomes an input of read generated Input XML file activity; after reading Input XML file apply propagation activity works and perform the FPTC analysis. The result of apply propagation is written on Output XML file. The Output XML file contains the information about input/ output failure of every component as well as the whole system. Back propagation of analysis activity, propagates the analysis result back on initial model where user can see the output failures of individual components as well as the entire system. List of activities for implementation is shown in Figure 5.3.
5.4.1 Modeling Exemplar System

Component view allows us to model software systems according to CHESS model definition. For modeling an exemplar system, we create a new class diagram named as SW_Components which contains components with their corresponding interfaces as shown in Figure 5.4. The example model contains three components named as (Source, Middle and Destination).
Each interface contains a set of operations. Kind of interface and direction of operations is used for distinguishing between input and output ports in ClientServerPort.

<table>
<thead>
<tr>
<th>Kind of Interface</th>
<th>Direction of Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Port</td>
<td>Required</td>
</tr>
<tr>
<td>Output Port</td>
<td>Required</td>
</tr>
<tr>
<td>Input Port</td>
<td>Provided</td>
</tr>
<tr>
<td>Output Port</td>
<td>Provided</td>
</tr>
</tbody>
</table>

Table 5.1: Distinctive Table for ClientServerPorts

Component implementation is used for decorating components with Input/Output ports and transformation rules. Transformation rules are specified in fptc attribute inside a comment having FPTC stereotype as shown in Figure 5.5.

System is modeled after decorating component implementation of each component used inside a system. Stereotype named as CHGaResourcePlatform is used for specifying composite component.
or system. Each input port of composite component or system is decorated with comment having FPTCSpecification stereotype which contains list of incoming failure(s) in failure attribute. FPTCSpecification stereotype is also used with each outgoing port of component in a system for writing analysis results.

![Figure 5.6: Modeling of Composite Component or System](image1)

Deployment view is used for software to hardware allocation and modeling of hardware systems. Sample hardware system is shown in Figure 5.7.

![Figure 5.7: Hardware System](image2)

After finished with decoration of composite component, instances of ‘SW_System’ are generated by selecting CHESS ➔ Build Instance command.

### 5.4.2 Activate/Deactivate Call Command

According to CHESS-methodology, analysis is launched from dependability analysis view. Implementation of activate/deactivate of call command is a step towards reducing chances of mistakes for launching analysis.

CHESSProfileManager contains information of CHESS-Views. Call command is activated upon matching of current view with dependability analysis view; otherwise, call command is set as deactivated as shown in source code placed in Figure 5.8.
public void updateStatus(IEditorPart activeEditor) {
    try {
        if (activeEditor instanceof CHESSEditor) {
            DesignView currentView = ((CHESSEditor) activeEditor).getDiagramStatus().getCurrentView();
            if (currentView.getName().equals(CHESSProfileManager.DEPENDABILITY_ANALYSIS_VIEW)) {
                setEnabled(true);
            } else {
                setEnabled(false);
            }
        } else {
            setEnabled(false);
        }
    } catch (Exception e) {
        return;
    }
}

Figure 5.8: Use of CHESSProfileManager for matching Current View

Plugin.xml file also contributes in implementation of activate/deactivate of call command based on certain checks enclosed in visibleWhen tags as shown in Figure 5.9.

  <command
    commandId="se.mdh.chess.fptc.analysis.FPTCAnalysisCommandID"
    label="FPTC"
    style="push">
    <visibleWhen
      checkEnabled="false">
      <with
        variable="activeEditorId">
        <or>
          <equals
            value="it.unipd.chess.editor">
          </equals>
          <equals
            value="org.eclipse.uml2.uml.editor.presentation.UMLEditorID">
          </equals>
        </or>
      </with>
    </visibleWhen>
  </command>

Figure 5.9: Activate/Deactivate Call Command

5.4.3 Launch FPTC Analysis

For launching FPTC Analysis, user must create a component with applied failurePropagationAnalysis stereotype in Dependability Analysis View and provide information for platform attribute as shown in Figure 5.10.
FPTC Analysis command is activated by selecting FPTC from CHESS → Analysis → Dependability menu as shown in Figure 5.11.

**Figure 5.11: Launching FPTC Analysis Command**

### 5.4.4 Call Build Instance

In this activity, we recall CHESS build instance command for reducing the chances of mistake by getting latest version instances for selected platform.

### 5.4.5 Read Model and Generate XML File

We are using ACCELEO (M2T) engine for transforming CHESS model into XML file. By using ACCELEO M2T engine, we extract necessary information from CHESS model for performing FPTC Analysis.

**Platform**

- **CHESS Modeling Element:** Dependability Analysis view is used for selection of target platform and launching FPTC Analysis as shown in Figure 5.10.
- **ACCELEO Code:**

```xml
<failurePropagationAnalysis>
  <Component>
    AnalyzeFPTC
  </Component>
  <platform> [SW_System_instSpec] </platform>
</failurePropagationAnalysis>
```
Once, fptc string is found, it is further break down into multiple transformation rule based-on semi colon.

\[
\text{Result:}
\]

\[
\text{Connections}
\]

Connectors are used for connecting components with each other. In CHESS model, connection is established between components by connecting output port of one component to the input port of another component.

\[
\text{CHESS Modeling Element:} \quad \text{Connectors are used for connecting component inside composite component or system as shown in Figure 5.6.}
\]

\[
\text{ACCELEO Code:} \quad \text{We are getting connections information from InstanceSpec package; which is generated through build-instance command and is selected as platform in FailurePropagationAnalysis stereotype at dependability analysis view.}
\]
Information about connections inside Composite Component/System is listed inside connections tags. Each connection attribute contains information about source and destination port name.

Flow Port

- CHESS Modeling Element:
  In CHESS model, Flow Ports (MARTE::Design_Model::GCM::FlowPort) are used for decorating hardware components as shown in Figure 5.7.

ACCELEO Code:
In flow port, direction attribute is used for identifying kind of port (‘in’ direction is used for an input port. Similarly, ‘out’ direction is used for an output port).

Client Server Port

- CHESS Modeling Element
  ClientServerPort (MARTE::Design_Model::GCM::ClientServerPort) is used for decorating ports of software components as shown in Figure 5.5.
ACCELEO Code for Identifying Direction of Client Server Port

Kind of interface and direction of operations is used for identifying the kind of Client Server Port. If kind of interface used in port is required and direction of operations in interface are ‘out’; or, kind of interface is provided and direction of operations enclosed in interface are ‘in’ then it is an input port as shown in table 5.1.

```
<InputPorts>
  [for(port:Port | comp.ownedPort)]
  [for(int:Interface | comp_package.ownedElement > filter(Interface))]
  [if(port.getValue(port.getAppliedStereotype('MARTE::MARTE_DesignModel::GCM::ClientServerPort'), 'reqInterface').toString().contains(int.name)) and (int.ownedOperation.ownedParameter.direction.toString()) -> includes('out'))=true and (int.ownedOperation.ownedParameter.direction.toString()) -> includes('in'))=false or
  (port.getValue(port.getAppliedStereotype('MARTE::MARTE_DesignModel::GCM::ClientServerPort'), 'provInterface').toString().contains(int.name)) and (int.ownedOperation.ownedParameter.direction.toString()) -> includes('in'))=true and (int.ownedOperation.ownedParameter.direction.toString()) -> includes('out'))=false]
  <InputPort varName="[port.name]", name="[port.name]", id=""/>
  [/for]
  [/for]
</InputPorts>
```

### Generated Text:

```
<InputPorts>
  <InputPort varName="SW_System_R2", name="SW_System_R2", id="", inputValues="early"/>
  <InputPort varName="SW_System_R1", name="SW_System_R1", id="", inputValues="omission"/>
</InputPorts>
```

### 5.4.6 Read Generated Input XML File

- This activity is involved in reading XML file to parse the file by creating new instance of `DocumentBuilder` class for obtaining document. `DocumentBuilder` class defines the API to obtain DOM Document instances from an XML document. By using this class, an application programmer can obtain a document from XML. An instance of this class can be obtained from the `DocumentBuilderFactory.newDocumentBuilder` method. Once an instance of this class is obtained, XML can be parsed from a variety of input sources such as Files, URLs and InputStream.

- The Element interface represents an element in an XML document. `Elements` interface inherits from `Node`; the generic `Node` interface is used to retrieve the set of all attributes. The root element of the XML file is FPTCInput.

```java
DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document document = builder.parse(xmlFile);
Element rootElement = document.getDocumentElement();
```

- Generally, `NodeList` interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented. We get all the elements of the component by `Components` tag.

- After this, we get all attributes of composite component by `getAttribute` which retrieve the value by name. Named represents the attribute which are to be retrieve such as
compositeVarName, compositeName and compositeCompId which are shown below in the code.

```java
NodeList component = rootElement.getElementsByTagName("Components");
Element componentGraphElement = (Element) componentGraphNode;
String compositeVarName = componentGraphElement.getAttribute("compositeComp");
String compositeName = componentGraphElement.getAttribute("compositeName");
String compositeID = componentGraphElement.getAttribute("compositeCompId");
```

- In following line of code we get input ports of the composite component by reading InputPort tag. Moreover, we get attributes of input port under for loop because each input port has its own attributes. These attributes are VarName, Name, portId and inputValues.
- In order to make separation between to input values we write split method. We also get input ports of other components in this way except, InputValue because input value is only used for composite component and this value is provided by user.
- After reading inputvalues, output ports of the composite component are read by similar source code.

```java
Element inputEntryElement = (Element) inputEntryNode;
NodeList inputs = inputEntryElement.getElementsByTagName("InputPort");
for (int i = 0; i < inputs.getLength(); i++) {
    Node inputNode = inputs.item(i);
    Element inputElement = (Element) inputNode;
    String inputVarName = inputElement.getAttribute("VarName");
    String inputName = inputElement.getAttribute("Name");
    String portId = inputElement.getAttribute("id");
    String inputValuesStr = inputElement.getAttribute("inputValues");
}
```

- Other component inside composite component are read by getting component tag and component attributes are get by ‘getAttribute’ method same as we did for composite component.
- After this, we write code for reading input and output ports of the component. Transformation rules are read by getting transformationRules tag. For reading of all expression one by one which are contained by a component we apply for loop.
- Moreover, all lhs patterns of each expression are get by getAttribute method. Similarly, rhs patterns are also read and code regarding to read lhs patterns are presented below. After reading attributes values, these values are set in TransformationRule.java class.

```java
NodeList transformationRules = componentElement.getElementsByTagName("TransformationRules");
Element constraintEntryElement = (Element) constraintEntryNode;
NodeList constraints = constraintEntryElement.getElementsByTagName("Expression");
for (int i = 0; i < constraints.getLength(); i++) {
    Node constraintNode = constraints.item(i);
    Element constraintElement = (Element) constraintNode;
    String lhsPattern = constraintElement.getAttribute("lhsPattern");
    String rhsPattern = constraintElement.getAttribute("rhsPattern");
}
```

- Connections between two ports get by reading Connection tag. For loop is applied for getting numbers of connections. First source port is read (where connection is started) then destination port is read (from where connection becomes end). The below code indicated the reading of connection between source and destination port. After reading the connection and its attributes

---

49
values these values are set in `DataPort.java` class but we are not showing algorithm of set values in below code.

```java
NodeList connectionsList = rootElement.getElementsByTagName("Connection");
for (int i = 0; i < connectionsList.getLength(); i++) {
    Node connectionNode = connectionsList.item(i);
    Element connectionElement = (Element) connectionNode;
    String src = connectionElement.getAttribute("srcVarName");
    DataPort srcPort = _composite.findDataPort(src);
    String dest = connectionElement.getAttribute("destVarName");
    DataPort destPort = _composite.findDataPort(dest);
}
```

### 5.4.7 Apply Propagation

This class is concerned with calculating input combinations, applicability of transformation rule, identify more specific transformation rule and apply transformation rule:

- **Compute input combinations:**
  
  Each component must contain at-least one input and output port for performing FPTC analysis. If component contains more than one incoming ports, then possible combinations of ports is computed for to be compliant with left hand side of transformation rules. Source code written for computing input combinations is written below:

```java
private List<InputFaultCombination> computeInputCombinations(
    SoftwareComponent comp) {
    List<InputFaultCombination> combinations = new ArrayList<InputFaultCombination>();
    for (DataPort inputPort : comp.getInputDataPorts()) {
        List<InputFaultCombination> curCombinations = new ArrayList<InputFaultCombination>();
        for (FailureType fault : InputPort.getFaults()) {
            InputFaultCombination curCombination = new InputFaultCombination();
            curCombinations.add(curCombination);
            curCombination.setInputFault(new InputFault(inputPort, fault));
        }
        if (curCombinations.isEmpty())
            return new ArrayList<InputFaultCombination>();
        combinations = computeCombinations(combinations, curCombinations);
    }
    return combinations;
}
```

- **Applicability of Transformation Rule:**
  
  Each component must contain at-least one transformation rule. Left hand of each transformation rule is matched with possible input combinations for identifying then applicability of transformation rule. Source written for checking that whether transformation rule is applicable or not is presented below:
private boolean matchTransformationRule(TransformRule rule, SoftwareComponent comp, InputFaultCombination combination) {
    // check that it is applicable
    List<FailureType> varMappings = new HashMap<String, FailureType>();
    for (FaultPattern faultPattern : rule.getLHSFaultPatterns()) {
        DataPort dataPort = faultPattern.getDataPort();
        FailureType fault = combination.getFault(dataPort);

        if (!patternMatches(rule, faultPattern, fault))
            return false;

        String faultVar = faultPattern.getVar();
        if (faultVar != null)
            varMappings.put(faultVar, fault);
    }
    return true;
}

- Identify more specific rule

Each component must have at least one matching pattern. If would also be possible that a component have more than one matching patterns. In case of more than one matching patterns, more specific transformation rule is selected as shown below:

SoftwareComponent comp = combination.getComponent();
for (TransformationRule rule : comp.getTransformationRules()) {
    if (matchTransformationRule(rule, comp, combination) == true){
        comp.addMatchingTransformationRule(Integer.toString(rule.getSpecificity()), rule);
    }
}

int maxSpecificity = 0;
if (comp.getMatchingTransformationRules().isEmpty() == false){
    for (String set : comp.getMatchingTransformationRules().keySet()){
        if (maxSpecificity < Integer.parseInt(set.trim()))
            maxSpecificity = Integer.parseInt(set.trim());
    }
    applyTransformationRule(comp.getMatchingTransformationRules().get(Integer.toString(maxSpecificity)), comp, combination);
    return true;
}

- Apply transformation rule

If transformation rule is matched, then right hand sides of failures enclosed at right hand side of transformation rule are propagated towards output ports of a component as shown below:
5.4.8 Propagation of Analysis Results Back into Model

Results of FPTC analysis are propagated back into the CHESS model. For this process, each output port of a composite component as well as its inside component(s) are connected with the comment having $FPTCSpecification$ stereotype. Attribute named $failure$ is used for specifying failure(s) at ports.

After performing FPTC analysis, output failure(s) of a component are added into the $failure$ attribute inside $FPTCSpecification$ stereotype. In CHESS model, Software Components are modeled in ComponentView. So, initial step is to get the ComponentView by using getPackagedElement method. Later, we get all elements inside ComponentView and stored them in a list.

```java
// apply rules
for (FaultPattern faultPattern : rule.getRHSFaultPatterns()) {
    DataPort dataPort = faultPattern.getDataPort();
    DataPort outputDataPort = comp.getOutputDataPort(dataPort.getId());
    $FailureType$ faultType = faultPattern.getFaultType();
    if (faultType != null) {
        outputDataPort.addFault(faultType);
        continue;
    }
    String variable = faultPattern.getVar();
    if (variable != null) {
        $FailureType$ fault = varMappings.get(variable);
        if (fault != null)
            outputDataPort.addFault(fault);
        continue;
    }
}
```

5.4.8 Propagation of Analysis Results Back into Model

Results of FPTC analysis are propagated back into the CHESS model. For this process, each output port of a composite component as well as its inside component(s) are connected with the comment having $FPTCSpecification$ stereotype. Attribute named $failure$ is used for specifying failure(s) at ports.

After performing FPTC analysis, output failure(s) of a component are added into the $failure$ attribute inside $FPTCSpecification$ stereotype. In CHESS model, Software Components are modeled in ComponentView. So, initial step is to get the ComponentView by using getPackagedElement method. Later, we get all elements inside ComponentView and stored them in a list.

```java
Package comView = (Package) umlModel.getPackagedElement("COMPONENT_VIEW");
EList<PackageableElement> packList = comView.getPackagedElements();

for (int i = 0 ; i < packList.size(); i++)
    if(packList.get(i).getName().equals("compositeComponent").getComponentName()){
        PropagateBackComp(CompositeComponent);
        for(SoftwareComponent subComp : compositeComponent.getChildren()) {
            PropagateBackComp(subComp);
        }
    }
```

A loop is used for finding the composite component from the list. Once, a composite component is found, another method named $PropagateBackComp$ is called for adding the resulting failure(s) of analysis. This method is also called for each component inside system/composite component.

```java
public void PropagateBackComp(SoftwareComponent comp){
    for (final DataPort outputDataPort : comp.getOutputDataPorts()){
        if(GetComment(outputDataPort) !=null){
            EList<$FailureType$> failure = (EList<$FailureType$>)
                comment.getValue("fptcspecification", "failure");
            failure.clear();
            for($FailureType$ failureType : outputDataPort.getFaults()){
                failure.add(failureType);
            }
        }
    }
}
```
PropagationBackComp method contains parameter value for SoftwareComponent. For each DataPort of a component, GetComment method is called which either returns null or applied comment on particular DataPort. Once comment is found, previous failure(s) on failure attribute are erased and new failure(s) are added.

```java
public static Comment GetComment(DataPort outputDataPort)
{
    for (Comment comment : comp.getOwnedComments())
    {
        if ((fptcsSpecification =
            comment.getAppliedStereotype("CHESS::Dependability::FailurePropagation::FPTCSpecification")) != null)
            if (comment.getValue(fptcsSpecification, "partWithPort").toString().contains("name: "+outputDataPort.getName()))
                return comment = comment;
    }
    return null;
}
```

GetComment method either returns comment owned by particular DataPort when FPTCSpecification stereotype is applied and DataPort name is matched with partWithPort attribute of comment or null otherwise. In Figure 5.12 CHESS model after performing FPTC analysis and back propagation of analysis result into the model is shown.

![Figure 5.12: The analysis results are propagated back on the model](image-url)
Chapter 6

CASE STUDY

In this chapter we present a case study to illustrate and evaluate the CHESS-FPTC framework that was introduced in Chapter 5.

The case study briefly describes ATM Adaptation Layer type 2 (AAL2) signaling protocol which is mainly used in telecommunication domain. This chapter is particular, is organized in the following way: section 6.1 provides a general introduction about AAL2 signaling protocol. Section 6.2 illustrates how to use CHESS-FPTC integrated plug-in to obtain the failure behavior of the AAL2 signaling protocol.

6.1 Overview of AAL2 Signaling

Note: This part of the thesis is mainly taken from [34], [35], [36] and [37].

The AAL2 is an adaptation layer type on top of an ATM network to carry voice. AAL2 provides the means for bandwidth-efficient transmission of low rate, short and variable length packets in delay sensitive applications. In the framework of some UTRAN (Universal Terrestrial Radio Access Network) interfaces, where AAL2 is used as a bearer, there is a need for a signaling protocol for establishment, maintenance and release of AAL2 connections (by exchanging control information) between the user and network or between two network elements. An AAL2 connection can be considered as logical concatenation of one or more AAL2 links between two service end-points and an AAL2 link is a communication path between two AAL2 nodes that are uniquely identified by a Channel Identifier (CID).

The signaling protocol used for AAL2 is called AAL2 Signaling protocol and it is standardized by ITU-T (ITU Telecommunication Sector). AAL2 signaling protocol supports multiple users.

6.1.1 AAL2 Signaling Protocol Functions

AAL2 signaling provides many functions. The most important functions are described below:

**Error Handling:** this function is in charge of detecting and reporting signaling procedures errors and other type of failure which are detected from AAL2 signaling end-point to AAL2 management.

**Connection management:** this function is in charge of the establishment and release of AAL2 connections.

**Reset:** this function is in charge of handling unusual cases. Three types of resets exist which are:
1. Reset all AAL2 paths that are connected with an end-point.
2. Reset a single AAL2 path and it’s regarding channels.
3. Reset a particular AAL2 channel in an AAL2 path.

**Block/Unblock:** this function is responsible of blocking and unblocking paths between two adjacent nodes. A block path is not used for new connection.

6.1.2 AAL2 Signaling Protocol Architecture

The architecture of AAL2 signaling protocol is presented in Figure 6.1. The architecture is constitutes of three main nodes: two AAL2 End-points to model the sender and the receiver and
one AAL2 Switch to model the necessity of routing the information. These nodes are in turn composed by the following building blocks:

- **AAL2 Served User (SU)** gets information from user and to map an incoming connection to corresponding radio channel.
- **AAL2 Signaling Stack** is either presented at Served User side or at intermediate side (AAL2 Switch). It provides several functions like: allocation of memory for various data structure, maintenance of the state of all AAL2 connections managed by the stack. Additional information can be found in [34].
  
a) At end-point AAL2 signaling stack provides the services (such as establishment of the connection and release of connection) to the served user. It behaves as multiplexer (*Signaling Stack Mux (SS_Mux)*) at one end and de-multiplexer (*Signaling Stack DeMux (SS_DeMux)*) at the other end.

b) At intermediate level AAL2 signaling stack provides routing and bridging support. *Signaling Stack (SS)* is used with Signaling Transport Converter and make signaling switch which breaks down the cell in to individual voice packet and vice versa. SS also provides routing function such as selection of path(s) according to destination end-point. SS also performs the encoding and decoding function.

- The **Signaling Transport Converter (STC)** exists below the AAL2 signaling stack. STC provides independence between *Underlying Signaling Transport* and AAL2 signaling protocol. STC provides data transfer service, convert one data type to other data type, congestion reporting, indication of maximum length of PDU (Data Protocol Unit). STC services also provide the assurance of error free transformation of data and delivery of data in a sequence [35].

- **Underlying Signaling Transport (UST)** provides the information of the availability of signaling network and remote signaling entities, numbering sequence/reassembling, segmentation and retransmission of service.

### 6.2 Experiment Overview

List of activities for creating AAL2 signaling protocol model in CHESS tool-set are shown in Figure 6.2. First activity is concerned with modeling of AAL2 signaling protocol in CHESS Functional View.
Second activity is related to decorate components in CHESS Extra-Functional View. In CHESS Dependability Analysis View, target platform is specified in platform attribute with-in failure propagation analysis stereotype for launching analysis. As an output of FPTC analysis, results are propagated-back into AAL2 signaling model.

Figure 6.2: Activity List for Experiment Overview

Modeling AAL2 Signaling Protocol in CHESS

As introduced in section 6.1, the architecture of AAL2 signaling protocol consists of three main nodes. We take a component-based approached and we identify a layered component-based architecture by analyzing AAL2 signaling protocol architecture. We identify three possible layers as shown in Figure 6.3. The bottom layer is constituted of atomic components (L1); the middle layer (L2) is constituted of composite components derived by composing the atomic components; finally, the top layer (L3) is constituted by the whole system, obtained by composing sub-system.

We identify six atomic components named SU, SS_Mux, STC, SS, SS_DeMux and UST as shown in layer L1. These atomic components makes sub-system/composite component named Originating Service End (OSE), Signaling Switch (SSH), and Destination Service End (DSE) at layer L2. Each sub-system represents one node of AAL2 signaling system architecture.

Figure 6.3: Layered Architecture
6.2.1 Functional Modeling of AAL2 Signaling Protocol Components

According to the CHESS methodology [28], architecture of AAL2 signaling protocol is modeled in ComponentView of CHESS tool-set. IServedUser, ISignalingStack, ISignalingTransportConverter, IUnderlyingSignalingTransport and IAAL2 interface; component types and component implementations of each component are placed. Relation between components and interfaces are modeled in the form of dependencies and realizations as shown in Figure 6.4. For distinguishing between incoming and outgoing ports of components we define direction of each operation as discussed in implementation section 5.4.1(Table 5.1).

![Figure 6.4: Components and Interfaces in AAL2 Signaling System](image)

6.2.2 Possible Failure Behavior of Components

As we introduce in section 2.7, a component may behave in four different ways source, propagator, transformer and sink. This section analyses possible failure behaviors of the atomic components within AAL2 signaling protocol. Since we do not have at disposal meaningful data and since we do not have the possibility to study directly the components through direct testing, the analysis follows a HAZOP-like approach. More specifically, each failure type available in FPTC is used as guideword to prompt possible failure behaviors of each component. We imagine possible deviations (failures) of the component’s expected behavior with respect to timing, value and sequence failures.

The result of our analysis is given

- Failure behavior of *ServedUser* component
  - *ServedUser* gets correct information concerning an AAL2 connection on time on the input port but does not map it onto the radio channel on time. In this case, it transforms noFailure into a late failure.
    
    \[\text{noFailure} \rightarrow \text{late}\]

  - *ServedUser* gets correct information concerning an AAL2 on time on the input port but does not map it at all. In this case, it transforms noFailure into an omission failure.
    
    \[\text{noFailure} \rightarrow \text{omission}\]
- \textit{ServedUser} gets correct information concerning an AAL2 on time on the input port and behaves correctly.

\textit{servedUser} \rightarrow \textit{noFailure}

- \textit{ServedUser} gets correct information concerning an AAL2 on time on the input port but does not map it onto a correct radio channel. In this case, it transforms noFailure into a valueCoarse failure.

\textit{noFailure} \rightarrow \textit{valueCoarse}

The previous four rules must be re-written in the following way to be compliant with [7]:

\textit{noFailure} \rightarrow \{\textit{omission, valueCoarse, late, noFailure}\}

- \textit{ServedUser} gets incorrect information concerning an AAL2 on time on the input port and as a consequence maps it onto an incorrect radio channel. In this case, it propagates a valueCoarse failure.

\textit{valueCoarse} \rightarrow \textit{valueCoarse}

The result of our analysis is then modeled within the CHESS tool-set. In particular, according to CHESS methodology, the behavior \textit{ServedUser} component is defined at component implementation level (ServedUser_Impl) as shown in Figure 6.4.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure6.4.png}
\caption{Served User at component Implementation level}
\end{figure}

- \textit{Failure behavior of SignalingStackMultiplexer} (SignalingStack_Mux) component:
  - \textit{SignalingStack_Mux} behaves properly.

\textit{noFailure} \rightarrow \textit{noFailure}

- \textit{SignalingStack_Mux} receives correct information in input but it is not able to allocate memory to data structure. Due to this reason, information is lost. In this case, it transforms noFailure into a valueCoarse failure.

\textit{noFailure} \rightarrow \textit{valueCoarse}

The above two rules must be re-written in the following way to be compliant with [7]:

\textit{noFailure} \rightarrow \{\textit{valueCoarse, no Failure}\}

- \textit{SignalingStack_Mux} does not receive all the input information needed (omission failure) to be able to perform multiplexing. Therefore, the multiplexing cannot be performed and an omission failure is propagated.

\textit{omission} \rightarrow \textit{omission}
- **SignalingStack_Mux** receives correct information in input but due to a flip bit in the information concerning the maximum length for PDU, it considers a wrong packet size and therefore it generates a commission failure.

  \[\text{omission} \rightarrow \text{omission}\]

  The above two rules must be re-written in the following way:

  \[\text{omission} \rightarrow \{\text{omission, commission}\}\]

- **SignalingStack_Mux** does not receive all the input information needed on time (late failure) to be able to perform multiplexing. Therefore, the multiplexing cannot be performed as quick as needed and the late failure is propagated.

  \[\text{late} \rightarrow \text{late}\]

- **SignalingStack_Mux** receives correct information in input but it does not maintain correctly the status of the AAL2 connections.

  \[\text{noFailure} \rightarrow \text{valueCoarse}\]

  The behavior **SignalingStack_Multiplexer** component is defined at component implementation level **SignalingStack_Mux_Impl** as shown in Figure 6.5.

> Behavior of **SignalingTransportConverter** (STC):

- **SignalingTransportConverter** receives correct information in input but waits to send signals to underlying transport network when network is crowded which causes the delay of delivery.

  \[\text{noFailure} \rightarrow \text{late}\]

- **SignalingTransportConverter** behaves properly.

  \[\text{noFailure} \rightarrow \text{no Failure}\]

  The previous two rules must be re-written in the following way to be compliant with [7]:

  \[\text{noFailure} \rightarrow \{\text{late, no Failure}\}\]
- **SignalingTransportConverter** receives correct but late information in input and being unable to reduce the delay it propagates a late failure.

  \[ \text{late} \rightarrow \text{late} \]

- **SignalingTransportConverter** receives commission sequence failure and tries to manage the sequence numbering and behaves like a sink (it transforms commission into noFailure).

  \[ \text{commission} \rightarrow \text{noFailure} \]

- **SignalingTransportConverter** receives omission sequence failure and tries to manage the sequence numbering and succeeds in getting the missing information, however not in time. SignalingTransportConverter succeeds in mitigating the failure.

  \[ \text{omission} \rightarrow \text{late} \]

The behavior **Signaling Transport Converter** component is defined at component implementation level **SignalingTransportConverter_Impl** as shown in Figure 6.6.

![Figure 6.6: Signaling Transport Converter Implementation](image)

- The behavior of **SignalingStack_DeMux**:

  - **SignalingStack_DeMux** does not receive all the input information needed (omission failure) to be able to perform de-multiplexing. Therefore, the de-multiplexing cannot be performed and an omission failure is propagated.

    \[ \text{omission} \rightarrow \text{omission} \]

  - **SignalingStack_DeMux** receives correct information in input but it is not able to allocate memory to data structure. Due to this reason, information is lost. In this case, it transforms noFailure into a valueCoarse failure.

    \[ \text{noFailure} \rightarrow \text{valueCoarse} \]

  - **SignalingStack_DeMux** behaves properly.

    \[ \text{noFailure} \rightarrow \text{no Failure} \]
The above two rules must be written as:

\[ \text{noFailure} \Rightarrow \{\text{valueCoarse}, \text{no Failure}\} \]

- **SignalingStack_DeMux** does not receive all the input information needed on time (late failure) to be able to perform de-multiplexing. Therefore, the de-multiplexing cannot be performed as quick as needed and the late failure is propagated.

  \[ \text{late} \Rightarrow \text{late} \]

- **SignalingStack_DeMux** receives correct information in input but it does not maintain correctly the status of the AAL2 connections.

  \[ \text{noFailure} \Rightarrow \text{valueCoarse} \]

The behavior **SignalingStack_DeMux** component is defined at component implementation level **SignalingStack_DeMux_Impl** as shown in Figure 6.7.

![Figure 6.7: Signaling Stack De-Multiplexer Implementation](image)

- The Behavior of **UnderlyingSignalingTransport**:  
  - **UnderlyingSignalingTransport** behaves properly.
    \[ \text{noFailure} \Rightarrow \text{no Failure} \]
  
  **UnderlyingSignalingTransport** receives correct information in input on time, generates a late failure, for instance, it cannot manage load on the network, by choosing a non-crowded routing path.

  \[ \text{noFailure} \Rightarrow \text{late} \]

  The previous two rules must be re-written in the following way:

  \[ \text{noFailure} \Rightarrow \{\text{late, no Failure}\} \]

  - **UnderlyingSignalingTransport** stops a late failure when, for instance, it can manage load on the network, by choosing a non-crowded routing path.

    \[ \text{late} \Rightarrow \text{noFailure} \]

  - **UnderlyingSignalingTransport** receives correct but late information in input and being unable to reduce the delay it propagates a late failure.
late \(\Rightarrow\) late

- UnderlyingSignalingTransport receives correct but late information in input and being unable to perform its function in time it increases the delay (delay can be considered as an omission).

late \(\Rightarrow\) omission

The above three rules can be re-written to be complaint with [7]:

late \(\Rightarrow\) [late, omission, noFailure]

The possible behavior of UnderlyingSignalingTransport at component implementation level (UnderlyingSignalingTransport_Impl) is presented in Figure 6.8.

![Figure 6.8: Underlying Signaling Transport Implementation](image)

- Behavior of SignalingStack:
  - SignalingStack behaves properly.
    noFailure \(\Rightarrow\) noFailure
  - SignalingStack receives correct information in input but it does not manage priority of tasks correctly then system waits for service. In this case, it transforms noFailure into a late failure.
    noFailure \(\Rightarrow\) late
  - SignalingStack receives correct information in input but it is not able to allocate memory to data structure. Due to this reason, information is lost. In this case, it transforms noFailure into a valueCoarse failure.
    noFailure \(\Rightarrow\) valueCoarse

The three rules are re-written in the following way:

noFailure \(\Rightarrow\) [late, valueCoarse, noFailure]

- SignalingStack receives correct information in input but due to a flip bit in the information concerning the maximum length for PDU, it considers a wrong packet size and therefore it generates a commission failure.

omission \(\Rightarrow\) omission
SignalingStack receives the information with an incorrect CID (valueCoarse). SignalingStack tries to recognize the channel but does not succeed and cannot provide its service. In this case, it transforms valueCoarse into an omission failure:

\[
\text{valueCoarse} \rightarrow \text{omission}
\]

Behavior of SignalingStack at component implementation level (SignalingStack_Impl) is shown in Figure 6.9.

![Figure 6.9: Signaling Stack Implementation](image)

### 6.2.4 FPTC Analysis at Sub-System Level

To calculate the failure behavior at system level, we proceed as follows: first we calculate the failure behavior at sub-system level (layer 2 of Figure 6.3) and then based on this calculus we can proceed and calculate the behavior at system level (layer 3 of Figure 6.3).

**OriginatingServiceEnd Sub-System/Composite Component:** is responsible of connection establishment. It receives the information from the user and sends the information to UnderlyingSignalingTransport component. OriginatingServiceEnd sub-system contains three atomic components (ServedUser_Impl, SignalingStack_Mux_Impl, and SignalingTransportConverter_Impl). The behavior of each individual component contained by OriginatingServiceEnd sub-system is already discussed in section 6.2.3. The input port of ServedUser_Impl is connected to input port of OriginatingServiceEnd. We provide noFailure on OriginatingServiceEnd sub-system at FPTC specification comment attached with its input port.

To evaluate the failure behavior of this composite component, we first calculate its behavior in response to a normal behavior. The model of the composite component before the analysis is illustrated in Figure 6.10. As Figure 6.10 shows, the composite component is fed with a noFailure behaviour (see the FPTC specification comment attached with the composite component input port).
The result of the analysis performed on the *OriginatingServiceEnd* sub-system is shown in Figure 6.11. As Figure 6.11 shows, the *OriginatingServiceEnd* sub-system produced \{late, noFailure\} failures on the output port.

This result is motivated as follows: *ServedUser_Impl* gets noFailure on input port and generates a set of failures \{noFailure, late, valueCorase, omission\} on its output port, called *ServedUser_Impl_P1*.

*SignalingStack_Mux_Impl* gets a set of failure \{noFailure, late, valueCorase, omission\} and since it gets a set of failure so, it is necessary to make the combination of input failure to be complaint.
with [7]. A noFailure behavior matches with the left-hand-side of its first rule, it transforms noFailure into \{valueCoarse, noFailure\}, omission failure matches with the left-hand-side of its second rule, it transforms omission into \{omission,commission\}, and late is matched with third rule and propagates late. Therefore, *SignalingStack_Mux_Impl* produces a set of failure \{commission, noFailure, valueCoarse, omission, late\} on output port.

*SignalingTransportConverter_Impl* receives a combination of input failure and omission matches to left-hand-side of forth rule of *SignalingTransportConverter_Impl* and transforms omission into late. *SignalingTransportConverter_Impl* transforms commission into onFailure and noFailure transforms into \{noFailure, late\}. Finally, \{noFailure, late\} is perceived on the output port of *OriginatingServiceEnd* sub-system.

Since, after performing analysis we receive the behavior of *OriginatingServiceEnd*. We can stop the propagation of failure on the output port *OriginatingServiceEnd* by using component failure mitigation techniques like NVP, RecoveryBlock and Control+Monitor.

**AAL2 SignalingSwitch:** is in charge of switching the packets. The switching is based on CID. SignalingSwitch sub-system consists of three components two *SignalingTransportConverter_Impl* and one *SignalingStack_Impl* component. We reuse *SignalingTransportConverter_Impl* and place two *SignalingTransportConverter_Impl* instances named *SignalingTransportConverter_Impl_inst1*, *SignalingTransportConverter_Impl_inst2* having same component type as shown in Figure 6.12.

Input port of *SignalingSwitch* sub-system is connected with input port of *SignalingTransportConverter_Impl* component but output port of *SignalingTransportConverter_Impl* is connected input port of *SignalingStack_Impl*. Similarly output port of other *SignalingTransportConverter_Impl* is connected output port of *SignalingSwitch* sub-system.

To evaluate the failure behavior of this composite component, we calculate its behavior in response to \{noFailure, late\} a set of failure behavior. The model of the composite component before the analysis is illustrated in Figure 6.12. As Figure 6.12 shows, the composite component is fed with a \{noFailure, late\} a set of failure behavior (see the FTPC specification comment attached with the composite component input port).

*SignalingSwitch* receives the input failure \{noFailure, late\} at SSH_R1 input port and provides failures to its inner component *SignalingTransportConverter_Impl* as shown in Figure 6.12.
FPTC analysis result at SignalingSwitch sub-system level is shown in Figure 6.13. SignalingSwitch sub-system receives \{noFailure, late\} a set of failure at its input port and propagates \{noFailure, late\} as it is on output port.

SignalingTransportConverter_Impl receives \{noFailure, late\} and produces \{noFailure, late\} on output port.

SignalingStack gets \{noFailure, late\} a set of failure on its input port but it produces \{noFailure, valueCoarse, late\} behavior as an output failure.

SignalingTransportConverter_Impl receives \{noFailure, valueCoarse, late\} a set of failure; transforms it \{noFailure, late\} and sends them to SignalingSwitch.
The FPTC analysis result at SignalingSwitch sub-system level is shown in Figure 6.13. SignalingSwitch sub-system receives \{noFailure, late\} failure at its input port and produces \{noFailure, late\} on output port.

**DestinationServiceEnd Sub-System/Composite Component:** contain three components ServedUser_Impl, SignalingTransportConverter_Impl and SignalingStack_DeMux_Impl (used for managing de-multiplexing) as shown in Figure 6.14. DestinationServiceEnd sub-system as Figure 6.14 shows, the composite component is fed with a set \{noFailure, late\} behavior (see the FTPC specification comment attached with the composite component input port).

The analysis result of DestinationServiceEnd sub-system is shown in Figure 6.15. SignalingTransportConverter_Impl component inside DestinationServiceEnd sub-system receives \{noFailure, late\} at its input port STC_Impl_R1 which is attached to input port of DestinationServiceEnd sub-system. SignalingTransportConverter_Impl produces \{noFailure, late\}.

SignalingStack_DeMux_Impl receives \{noFailure, late\} behavior on it input port and transforms it into \{noFailure, valueCoarse, late\} as we can see in Figure 6.15. ServedUser_Impl gets \{noFailure, valueCoarse, late\} on input port and propagates \{noFailure, valueCoarse, omission, late\} towards output port of DestinationServiceEnd.
6.2.5 FPTC Analysis at System Level

AAL2 signaling system composed of three sub-systems/composite components named as *OriginatingServiceEnd, SignalingSwitch, DestinationServiceEnd* and two *UnderlyingTransportConverter* components as shown in Figure 6.16.
For running FPTC analysis at the system level of AAL2 system, it is necessary to identify the behavior of all sub-systems. Failure behavior of sub-systems can be identified by running FPTC analysis on sub-system level by injecting different failures (as we described previously). Behaviors of all sub-system are as follow:

**Behavior of OriginatingServiceEnd Composite component/sub-system**

After having performed the analysis we obtain the behavior shown in Table 6.1.

<table>
<thead>
<tr>
<th>Behavior of OriginatingServiceEnd sub-system</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSE_R1.noFailure -&gt; OSE_P1.onFailure, OSE_P1.late;</td>
</tr>
<tr>
<td>OSE_R1.valueCoarse -&gt; OSE_P1.valueCoarse</td>
</tr>
</tbody>
</table>

Table 6.1: Behavior of OriginatingServiceEnd Sub-System

**Behavior of SignalingSwitch Composite component/sub-system**

After having performed the analysis we obtain the behavior shown in Table 6.2.

<table>
<thead>
<tr>
<th>Behavior of SignalingSwitch sub-system</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSH_R1.noFailure -&gt; SSH_P1.noFailure, SSH_P1.late;</td>
</tr>
<tr>
<td>SSH_R1.commission -&gt; SSH_P1.noFailure, SSH_P1.late;</td>
</tr>
<tr>
<td>SSH_R1.omission -&gt; SSH_P1.late;</td>
</tr>
<tr>
<td>SSH_R1.late -&gt; SSH_P1.late;</td>
</tr>
</tbody>
</table>

Table 6.2: Behavior of SSH Sub-System

**Behavior of DestinationServiceEnd Composite component/sub-system**

After having performed the analysis we obtain the behavior shown in Table 6.3.

<table>
<thead>
<tr>
<th>Behavior of DestinationServiceEnd sub-system</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSE_R1.noFailure -&gt; DSE_P1.noFailure, DSE_P1.valueCoarse, DSE_P1.omission, DSE_P1.late;</td>
</tr>
<tr>
<td>DSE_R1.commission -&gt; DSE_P1.valueCoarse, DSE_P1.omission, DSE_P1.late, DSE_P1.noFailure;</td>
</tr>
<tr>
<td>DSE_R1.omission -&gt; DSE_P1.late;</td>
</tr>
<tr>
<td>DSE_R1.late -&gt; DSE_P1.late;</td>
</tr>
</tbody>
</table>
Table 6.3: Behavior of DSE Sub-System

After getting the behavior of all three sub-systems, we create component implementation diagrams and at implementation level we define rule of these components. We create the composite structure diagram of AAL2 signaling system as we do at sub-system level is shown in Figure 6.16. In AAL2 system/composite component, output port of \textit{OriginatingServiceEnd} is attached with input port of \textit{UnderlyingSignalingTransport}. Similarly output port of \textit{UnderlyingSignalingTransport} is attached with input port of \textit{SignalingSwitch} component. We reuse \textit{UnderlyingSignalingTransport} component so, Input port of this component is attached with output port of \textit{SignalingSwitch} and so on. Input failure (noFailure) is provided on input port (AAL2\_R1) of AAL2\_system and run the FPTC analysis on AAL2\_system.

![Composite Structure Diagram of AAL2 System](image)

\textbf{Figure 6.16: AAL2 System before analysis}

FPTC analysis result is shown in Figure 6.17; output failure \{omission, valueCoarse, late, noFailure\} is delivered at output port (AAL2\_P1) of AAL2\_System. \textit{OriginatingServiceEnd} generates a failure \{noFailure, late\} and behaves like a source.

\textit{UnderlyingSignalingTransport} propagates \{noFailure, omission, late\} towards \textit{SignalingSwitch}. \textit{SignalingSwitch} produces \{noFailure, late\}.
UnderlyingSignalingTransport component transforms them into \{noFailure, omission, late\}. But DestinationServiceEnd produces \{noFailure, valueCoarse, omission, late\} a set of failure and send it to output port of AAL2_System. It means that user waits for connection establishment.

This case study has shown that thanks to the usage of FPTC plug-in, designers can be supported in their design decisions, thanks to the possibility of evaluating the robustness of their architectures. Even though, currently, the FPTC analysis can be performed only on flat architectures, the case study has illustrated how to use the tool to analyse layered architectures.

After having identified the failure behavior of each component, design decisions can be taken. For instance, components which behave like sources can be combined by new components which behave like sinks. More specifically, the designer may decide to mitigate the omission failure generated by the DestinationServiceEnd component by applying one of the following failure specific mitigation techniques: Control+Monitor, NVP (N-version programming) and Recovery Blocks, as discussed in [38].
Chapter 7

CONCLUSION

Component-based and model-driven development approaches provide the assurance of better complexity management, reusability, easier maintenance and reduction of risk factors in the development and implementation of the safety critical systems; these approaches focus on the functional attributes of the components. But, CHESS tool-set supports an industrial-quality MDE infrastructure for specification, analysis and verification of functional as well as non-functional properties such as dependability in component-based software systems.

In this thesis, both research and practical work is presented. In the research work, our focus is on component-based, model driven approaches and system’s dependability threats.

According to the causality chain that inter-relates the dependability threats, a fault produces an error when it becomes active if not handled may lead to the system’s failure. Regarding dependability threats in component-based systems, we only consider term failure to avoid misconception. A service failure occurs when delivered service deviates from the correct service. CBSE approach provides the maximum reusability of the components. Component Composition is a combination of two or more components that yielding a new component behavior at different abstraction levels.

We conduct a survey on various techniques that are available to analyze system’s dependability: some of these techniques provide a qualitative evaluation; some others provide a quantitative evaluation; some are modular; some allow for automatic evaluation and some others are not allowing the automation. But FPTC technique is qualitative as well as automatic. FPTC also provide bottom to top analysis strategy means that it start analysis on atomic components, and then analyze the behavior of sub-system level and finally, behavior of the system is analyzed. During our research on FPTC, we find ambiguities about FPTC syntax and rules presented by different authors. Therefore we modified the syntax of FPTC and remove the ambiguities.

In the context of CHESS project, the thesis aim is providing a tool-supported component-based and model-driven-based methodology to engineer high integrity real-time component-based dependable embedded systems.

According to MDE principles, analysis models are automatically obtained from high level model of system’s architecture. This approach prevent the user for providing details at analysis level of the system, without this approach providing details can be error-prone process. These analysis results are back propagated in the system model. Model transformations are the means by which models can be manipulated. We use M2T transformation technique to get the information about high-level design model in textual format and this information is used to perform analysis on model. FPTC plug-in interacts with several other CHESS plug-ins to compliant with CHESS tool-set.

By using CHESS-FPTC plug-in user can identify failure behavior of the entire system as well as behavior of its building components. After analyzing the system behavior, we can replace the components which cause failure in system. So, CHESS-FPTC provides help to engineer real-time component-based embedded dependable systems. CHESS-FPTC can be enhanced and possible extension and it is described in future section.
Chapter 8

FUTURE WORK

The chapter discusses future work that could be considered to enhance as well as extend the CHESS-FPTC plugin. This chapter is organized as follow: Section 8.1 highlights the enhancements that could be envisaged and Section 8.2 presents potential extensions.

8.1 Enhancement in CHESS-FPTC plug-in

The CHESS-FPTC plug-in provides the solution, an integrated tool-support to engineer high integrity real-time component-based dependable embedded systems, following a component-based and model-driven-based methodology. But some enhancement could be possible in future like run FPTC on system level (nested composite component) automatically. Currently FPTC analysis is semi automatic means FPTC analysis works on flat level sub-system level/ composite component). In enhancement, there no need to provide input failure on every sub-system level. In particular input failure will be provided on System level/nested composite component and analysis will be perform automatically on system as well as all sub-systems level contained by that system.

8.2 FPTC Extension for FI4FA

A useful and small extension of the CHESS-FPTC plug-in could be to support the FI4FA analysis technique [10]. This extension would be possible by considering additional failure types. FI4FA focuses on failures avoidable through transaction-based mitigations.
REFERENCES


APPENDIX

<?xml version="1.0" encoding="UTF-8"?>
<FPTCInputs>

<Components compositeComp="C0" compositeName="SW_System" compositeCompId="0">
  <InputPorts>
  <InputPort varName="SW_System_R2" name="SW_System_R2" id="SW_System_IP1" inputValues="early" />
  <InputPort varName="SW_System_R1" name="SW_System_R1" id="SW_System_IP3" inputValues="omission" />
  </InputPorts>
  <OutputPorts>
  <OutputPort varName="SW_System_P1" name="SW_System_P1" id="SW_System_OP1" />
  </OutputPorts>
  <Component varName="C1" compName="Source_impl" id="1">
    <InputPorts>
    <InputPort varName="Source_R2_impl" name="Source_R2_impl" id="Source_impl_IP1" />
    <InputPort varName="Source_R1_impl" name="Source_R1_impl" id="Source_impl_IP3" />
    </InputPorts>
    <OutputPorts>
    <OutputPort varName="Source_P1_impl" name="Source_P1_impl" id="Source_impl_OP2" />
    </OutputPorts>
    <TransformationRules>
    <Expression expNum="exp1" lhsPattern="Source_R1_impl.omission,Source_R2_impl.wildcard" rhsPattern="Source_P1_impl.omission" />
    <Expression expNum="exp2" lhsPattern="Source_R1_impl.variable,Source_R2_impl.wildcard" rhsPattern="Source_P1_impl.variable" />
    <Expression expNum="exp3" lhsPattern="Source_R1_impl.omission,Source_R2_impl.early" rhsPattern="Source_P1_impl.commission" />
    </TransformationRules>
  </Component>

  <Component varName="C2" compName="Middle_impl" id="2">
    <InputPorts>
    <InputPort varName="Middle_R1_impl" name="Middle_R1_impl" id="Middle_impl_IP1" />
    </InputPorts>
    <OutputPorts>
    <OutputPort varName="Middle_P1_impl" name="Middle_P1_impl" id="Middle_impl_OP3" />
    </OutputPorts>
    <TransformationRules>
    <Expression expNum="exp1" lhsPattern="Middle_R1_impl.commission" rhsPattern="Middle_P1_impl.early" />
    <Expression expNum="exp2" lhsPattern="Middle_R1_impl.early" rhsPattern="Middle_P1_impl.late" />
    <Expression expNum="exp3" lhsPattern="Middle_R1_impl.omission" rhsPattern="Middle_P1_impl.early" />
    </TransformationRules>
  </Component>

  <Component varName="C3" compName="Destination_impl" id="3">
    <InputPorts>
    <InputPort varName="Destination_R1_impl" name="Destination_R1_impl" id="Destination_impl_IP1" />
    </InputPorts>
    <OutputPorts>
    <OutputPort varName="Destination_P1_impl" name="Destination_P1_impl" id="Destination_impl_OP4" />
    </OutputPorts>
    <TransformationRules>
    <Expression expNum="exp1" lhsPattern="Destination_R1_impl.early" rhsPattern="Destination_P1_impl.early" />
    </TransformationRules>
  </Component>

  <Connections>
  <Connection srcVarName="Source_P1_impl" destVarName="Middle_R1_impl" />
  <Connection srcVarName="Middle_P1_impl" destVarName="Destination_R1_impl" />
  <Connection srcVarName="Destination_P1_impl" destVarName="SW_System_P1" />
  <Connection srcVarName="SW_System_R2" destVarName="Source_R2_impl" />
  <Connection srcVarName="SW_System_R1" destVarName="Source_R1_impl" />
  </Connections>
</Components>
</FPTCInputs>