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
Supervisor: Barbara Gallina
Examiner: Sasikumar Punnekkat

Date: 2011-11-17
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 behavior 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

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

## 2. BACKGROUND

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 .................................................. 16
2.5 Failure Behavior Analysis Techniques ......................... 18
   2.5.1 Component Fault Trees ....................................... 18
   2.5.2 Failure Propagation Transformation Notation ........... 19
   2.5.3 Hierarchically Performed Hazard Origin and Propagation Studies ........................................... 19
   2.5.4 State-Event Fault Trees ...................................... 20
   2.5.5 Architecture Analysis and Description Language ...... 20
2.6 Failure Propagation and Transformation Calculus ........... 21

## 3. PROBLEM FORMULATION AND ANALYSIS .............................................. 26

## 4. METHOD

4.1 Selection of Proper Failure Behavior Analysis Technique .............. 28
4.2 Understanding and Comparison of FPTC rules defined by different authors .................. 29
4.3 Integration of CHESS-FPTC Plug-in with CHESS Tool-Set .................. 30
4.4 Selection of Transformation Technology ......................... 30
   4.4.1 Why we select ACCELEO for M2T ... ........................................ 31
   4.4.2 Back Propagation ................................................. 31

## 5. SOLUTION

5.1 Proposed FPTC ....................................................... 32
   5.1.1 Multiple incoming failures on Input Ports ................. 32
   5.1.2 No Pattern Matching ........................................... 32
   5.1.3 Usage of Wildcard .............................................. 32
   5.1.4 Specificity of Wildcards and Variables ................... 32
   5.1.5 Modified Syntax of FPTC ....................................... 32
5.2 Meta-Modeling Concepts Provided to Extends CHESS ML ........ 33
   5.2.1 Component Model ............................................... 33
   5.2.2 Components ..................................................... 33
   5.2.3 Interfaces ......................................................... 33
   5.2.4 Ports ............................................................... 34
   5.2.5 Comments ........................................................ 34
   5.2.6 Selection of Platform .......................................... 34
## 5.2.7 Software/Hardware Systems

5.3 Proposed Design

5.3.1 Implementation of FPTC Analysis in Progress IDE
5.3.2 Design of the CHESS-FPTC Plug-in
5.3.3 Integration of FPTC Plug-in in CHESS Platform

5.4 Implementation

5.4.1 Modeling Systems within the CHESS tool-set
5.4.2 Activate/Deactivate Call Command
5.4.3 Launch FPTC Analysis
5.4.4 Call Build Instance
5.4.5 Read Model and Generate XML File
5.4.6 Read Generated Input XML File
5.4.7 Apply Propagation
5.4.8 Propagation of Analysis Results Back into Model

## 6. CASE STUDY

6.1 Overview of AAL2 Signaling

6.1.1 AAL2 Signaling Protocol Functions
6.1.2 AAL2 Signaling Protocol Architecture

6.2 Experiment Overview

6.2.1 Functional Modeling of AAL2 Signaling Protocol Components
6.2.2 Possible Failure Behavior of Components
6.2.3 FPTC Analysis at Sub-System Level
6.2.4 FPTC Analysis at System Level
6.2.5 Discussion

## 7. CONCLUSION

## 8. FUTURE WORK

8.1 Enhancement in CHESS-FPTC plug-in
8.2 FPTC Extension for FI4FA
Chapter 1

INTRODUCTION

In this chapter introduction of thesis is presented. This chapter is organized as follow: section 1.1 provides context and motivation. Section 1.2 highlights our contribution for achieving aims of the thesis. Section 1.3 outlines the 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 address extra-functional properties in adequate manner [11].

The CHESS (Composition with Guarantees for High-integrity Embedded Software Components Assembly) project aims to address this limitation by providing a methodology to engineer high integrity real-time component-based dependable 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 tool-set supports CHESS methodology, an industrial-quality Model-Driven Engineering (MDE) infrastructure for specification, analysis and verification of extra-functional properties such as dependability in component-based software systems. CHESS tool-set is expected to support integration of dependability property with the component model [11].

The CHESS tool-set contains a set of plug-ins integrated within Eclipse IDE. CHESS dependability profile allows user to model and analyze the dependability characteristic such as calculating failure behavior of components. In this case important questions arise: how system dependability can be evaluated? How the evaluation of the dependability can be integrated within the CHESS tool-set?

The context of this thesis is to investigate an integrated framework to evaluate the dependability of high integrity real-time component-based embedded systems. Study and comparison of techniques for failure behavior analysis (focus on FPTC-based analysis technique). Analysis, design and implements CHESS-FPTC plug-in in CHESS tool-set.

1.2 Contributions

CHESS-FPTC plug-in integrated with CHESS tool-set is provided which allows users to model the system, perform the analysis and get analysis results on initial model.

To achieve the CHESS-FPTC plug-in, the following steps are done. 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. Failure Propagation and Transformation Calculus (FPTC) technique is
selected for automatic calculation of failure behavior of the entire system from the failure behavior of its building components. The syntaxes of FPTC presented by different authors are compared and our version in the context of the CHESS project is provided.

The CHESS tool-set is analyzed for identifying available and missing parts that are necessary for performing FPTC analysis technique within CHESS model. Study and analyze CHESS Modeling Language that is called CHESS ML profile. CHESS ML is built from subsets of standard languages Sys ML (Systems Modeling Language), UML (Unified Modeling Language), MARTE (Modeling and Analysis of Real-Time and Embedded Systems).

The transformation from high level CHESS ML model to low level dependability analysis and back transformation of analysis results on initial model is implemented. Model-to-text transformation, for getting the necessary information about model in the form of XML (Extensible Markup Language) file is performed. Model-to-Text transformation is done by using ACCELEO transformation engine. Generated XML file contains necessary information for execution of FPTC analysis. This generated XML file is taken as input 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 previous components or introducing new components. A case study to evaluate this framework is provided.

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 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 identifies possible extension and direction of future work.
Chapter 2

BACKGROUND

In this chapter background information that is necessary for understanding the problem as well as for building the solution is presented. This chapter is organized in the following way: Section 2.1 presents dependability; Section 2.2 describes the component-based software engineering approach. Section 2.3 highlights the basic concepts of model-driven architecture and transformation techniques. In section 2.4, the CHESS project in terms of model-driven engineering is discussed. In section 2.5, analysis techniques are presented. Section 2.6 contains comprehensive information of 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. In terms of quantitative point of view: “The ability to avoid service failures that are more frequent and more severe than is acceptable to the user(s)” [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. 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 2.1 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 2.1 t1 represents a fault. Faults of a system can be categorized as external and internal faults.

- **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.
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 $t_3$ in Figure 2.1 indicates failure. A service may fail because of two reasons: first, service does not act according to the functional specification and secondly, 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 $t_2$ 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 2.2 below:

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 views. These points of views are: domains, consistency, detectability and consequences for the environment. The explanation of all four points of views is outside the scope of thesis. In what follow, only the domain point of view is explained.

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 function”) and timing failure (“time of reaching information deviates from implementing the system function”).
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 hierarchal 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. “Components can export and import interfaces to and from environments that may include other components” [3]. A component has two kinds of interfaces that can be distinguish as:

- 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 Figure 2.3.

![Figure 2.3: Component and its interfaces](image-url)
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. From the composition a new component behavior emerges 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, five kinds of transformations are shown in Figure 2.4.

![Figure 2.4: Relation between PIMs and PSMs](image-url)
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 2.4. These transformations are not always automated.

2. PIM to PSM: Theses 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.

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

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

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

In our solution PIM to PSM transformation is used because, target XML file is required for implementation.

➢ **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. In another way, 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 M1 is said to conforms to a given meta-model M2 if it can be obtained through a legal collection of concepts as defined by M2. This is the second relation of the basic principle shown in Figure 2.5 at level M2.

➢ **Meta Meta-Model**

A meta-meta-model is a model that conforms to it-self. “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. M0 is the lowest level and it represents a system. M1 layer represents model which describe the aspect of the system. M2 layer indicates meta-model and modeling language used for defining M1 is specified in this layer. M3 layer represents meta-meta-model. In MDA best-known meta-model is UML meta-model and Meta meta-model is MOF (Meta-Object Facility).

### 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 technologies:** Atlas Transformation Language (ATL), Query-View Transformation (QVT-o) Operational and Declarative QVT (core and relational) and ACCELEO etc.

- **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: Efftinge 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 (Eclipse Modeling Framework) text (template languages), Kermeta, MOF script (object oriented languages) etc. This thesis, as it is explained in Chapter 5, makes use of ACCELEO to build the solution to problem addressed. Therefore, information concerning ACCELEO transformation engine is presented 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 into text (XML Files) by writing code for generating text in ACCELEO template.

![Figure 2.6: ACCELEO Generating Principle [26]](image)

Example: Papyrus UML model transformation to XML by using ACCELEO

In ACCELEO, transformation rules are written in templates (having extension .mtl) that produce text-files (source code or documentation) after execution. Guidelines for performing simple M2T transformation (Papyrus UML model to XML file) are listed below:

Figure 2.7 represents a Papyrus UML model which is used for M2T transformation. Model contains a composite component named Car. Composite component contains two properties named rear and engine having component type Wheel and Engine respectively. Wheel component has port (receive_force) and Engine component contains port (provide_force).
The transformation rules for M2T are shown in Figure 2.8. “[module GenerateM2T_Acceleo (http://www.eclipse.org/uml2/3.0.0/UML)]” indicates input meta-model URI.

```
[module GenerateM2T_ACCEL('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.matches('Car'))]
<components composite=comName="[component.name/]">
 [for(property:Property | component.eAllContents(Property))]
  <Component compName="[property.name/" compType="[property.type.name/"]">
   <Ports>
    [for(port:Port | property.type->filter(Component).ownedPort)]
     <Port name="[port.name/"/>
    [/for]
   </Ports>
  </Component>
 [/for]
 <Comment>
  [for(comm:Comment | component.ownedComment)]
   <Comment body="[comm._body/"]" />
  [/for]
 </Comment>
 <Connections>
  [for(connector:Connector | component.ownedConnector)]
   <Connection source="[connector.end.role.name->sep("" destination="")/"]" />
  [/for]
 </Connections>
[/if]
[/file]
[/template]
```

Figure 2.8: Code for generating XML

Template structure starts with “[template public Generate_M2T (model: Model)]” and ends with [/template]; where ‘Generate_M2T’ is a template name. Transformation rules are enclosed inside ‘template’ tags:

- The name of target file is specified in opening file tag “[file (‘Text.xml’, false, ‘UTF-8’)].”
For retrieving all components (used inside model) for loop is used. Furthermore, for every component, if statement is used for matching component name with Car.

If component name is matched, then all properties owned by composite component are retrieved by using for loop. For every property, information concerning property name and it’s type is printed in opening Component tag. In Ports tags, for block is used for retrieving and printing names of ports which are owned by property.

In Comment tags, for block is used for printing the body of every comment used inside composite component.

In Connections tags, for block is used for printing information of every connector used inside component. For each connector, written code prints source and destination port names.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Components>
  <Components compositeCompName="Car">
    <Component compName="rear" componentType="Wheel">
      <Ports>
        <Port name="receive_force" />
      </Ports>
    </Component>
    <Component compName="engine" componentType="Engine">
      <Ports>
        <Port name="provide_force" />
      </Ports>
    </Component>
    <Comment>
      <Comment body="rear wheel [2]" />
    </Comment>
  </Components>
<Connections>
  <Connection source="provide_force" destination="receive_force" />
</Connections>
</Components>
```

Figure 2.9: generated xml File (Text.xml)

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

- In opening Components tag, component name “Car” is printed in compositeCompName attribute.
- In opening Component tag, information concerning property name and property type is printed inside compName and compType attributes. Since, two properties are used in composite component; information concerning each property is printed in separate Component tags.
- Moreover, for every property, Ports tags contain information concerning owned port(s); port name is printed inside name attribute.
- In Comment tags, information concerning comment body is printed in body attribute.
- Information concerning each connection is printed inside Connection tags. For every connection, source and destination port names are printed in source and destination attributes.

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), “which is independent of the execution platform that will actually implement the system” [12].
The development process is endured by various kinds of analyses for example dependability. These analyses allow assessing capabilities of designing systems with respect to different aspects. According to MDE principles, analysis models are automatically obtained from high level model of system’s architecture [12]. This approach prevent the user for providing details at analysis level of the system; without this approach providing details can be error-prone process. Analysis results are used to intensify the high level CHESS model. CHESS project supports iterative development process in which system model is continuously updated based on analysis results, which are back propagated in the system model. 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 allows enriching CHESS ML model with information related to dependability and safety.

![Figure 2.10: CHESS Workflow for system development and analysis](image)

**Overview of CHESS Component Model**

In CHESS component model, packages are defined as a set of concepts that are used to define a software component model in CHESS. CHESS model has various views named as ComponentView, FunctionalView, ExtraFunctionalView, DeploymentView, Analysis View and Requirement View etc. Details of CHESS component model views that are required in FPTC analysis are described below:

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

- **FunctionalView:** In this view, user specifies the functional attributes on model. In FunctionalView, user can decorate ComponentType and ComponentImplementation with property, connector and ClientServer Ports. ClientServer Ports are used for defining interface(s) and is taken from ‘MARTE::MARTE_DesignModel::GCM’. Components are connected through their interfaces, a component can provide services through ‘provInterface’ attribute and require services through ‘reqInterface’ attribute.

- **ExtraFunctionalView:** 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, which gives the guarantee that the definition of the functional entities is not altered. In ExtraFunctionalView, user can decorate comments with FPTCSpecification and FPTC stereotypes.

- **DeploymentView:** In this view, user specifies the allocation of software resources on hardware platform. Hardware entities and their applicable properties are expressed using MARTE stereotypes. Component instances are derived from the ExtraFunctionalView which can be imported to specify their allocation to hardware entities [20].

- **AnalysisView:** Analysis View is divided into DependabilityAnalysisView and RtAnalysisView. FPTC Analysis is invocated from DependabilityAnalysisView.
- **DependabilityAnalysisView**: From this view, user triggers FailurePropagationAnalysis, which is fed with the user model taken from the ExtraFunctionalView and the DeploymentView. 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 some relevant failure behavior analysis techniques. This section introduces some relevant failure behavior analysis techniques, new compositional techniques such as Component Fault Trees (CFTs), Failure Propagation Transformation Notation (FPTN), Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS), Fault Propagation and Transformation Calculus (FPTC), State-Event Fault Trees (SEFTs), and Architecture Analysis and Description Language (AADL). The introduction mainly borrows from [19, 21]. 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 Trees

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 and can be analyzed (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 fault 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. Moreover, input and output failure ports are matched on basis of their names and types. Example of CFTs (Fire Alarm System) is presented in Figure 2.11. CFTs have been used in many industrial projects and supported by ESSaReL (Embedded Systems Safety and Reliability Analyzer) window-based tool for graphical specification and efficient evaluation. CFTs are based on matching of incoming and outgoing failure ports with limited support of architectural dependencies. CFTs is a manual tool guided technique for generation of error models for hierarchal components [19].

![Figure 2.11: Example of CFTs [19]](image-url)
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. Detail of fire alarm system is available at [19]. In Figure 2.11, the incoming failures are Smoke_detected omission, Smoke_detected commission; however, outgoing failures are Sprinkling omission and Sprinkling commission.

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. FPTN supports qualitative evaluation. The basic entity of FPTN is FPTN-module as shown in Figure 2.12. FPTN 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) which defines generation, detection, propagation and transformation of failure(s) in a component [38]. Each module is represented by a box; a module can have more than one sub modules. Set of inputs and outputs are represented by a 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 be read in the following way: Smoke_detected:tl, Smoke_detected:o, and Smoke_detected:c are incoming failures. Sprinkling:o and Sprinkling:c are outgoing failures. The transformation and propagation is 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.

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-HOPS) 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 indicates how output failures of each component can be caused by internal malfunctions. Failure Mode and Effects Analysis called Interface Focused FMEA (IF FMEA) is used as a means of deriving such failure annotations. Analysts apply this technique on components to identify plausible output failures such as the omission, commission, value or timing failures at each output to determine the
local causes of such events as combinations of internal component malfunctions and input failures [38]. 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 be generated and can be analyzed through minimum cut-sets. In practice, HiP-HOPS has been successfully applied in many complex real-world systems [19]. An example of 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>λ (failure/hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprinkling - c</td>
<td>Commission Failure: control unit activities in sprinkling when it is not needed</td>
<td>Smoke_detected - c</td>
<td>Software Fault or Controller Malfunction</td>
<td>1.0x10^-8 5.0x10^-7</td>
</tr>
</tbody>
</table>

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

### 2.5.4 State-Event Fault Trees

State-Event Fault Trees (SEFTs) are a formalism that differentiates events (sudden phenomena, in particular state transitions) from states (that last over a period of time) [19]. SEFTs support quantitative evaluation. 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 SEFTs transitions 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 “sparkling” in Figure 2.14. SEFTs do not provide support for identification of error model in basic architectural components. SEFTs are manual tool guided technique for generation of error models [19]. SEFTs are 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. In AADL component implementation describes internal structure of the component. Figure 2.15 presents example of component implementation of fire alarm system. Connections are defined through
connected input and output ports as shown in Figure 2.15. Properties section specifies the execution of process AU (alarm unit) on HW hardware, for detailed information read [19].

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 error 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. AADL error propagations are performed through matching of incoming and outgoing error(s) with already defined dependency rules between AADL components. AADL with Error Annex is supported by OSATE (Eclipse plug-in) tool. AADL supports quantitative as well as qualitative evaluation [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

By using 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 sub-systems and a failure in a sub-system represents a fault elsewhere, which is shown in Figure 2.16. For example, a system is 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” [10]. In this thesis, the term failure is used as input-output behavior in FPTC analysis.

Classification of failure [9] is discussed earlier (Section 2.1.3), but now those failure types are considered 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 to expected time interval.
  - **Early** means that a component provides result before its expected time interval.

![Figure 2.16: Component Based-System](image)
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 is 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. An 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 a component [10, 23].

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

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

In FPTC wildcards and variables are used for reducing specification burden. Usage of ‘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 to which it is matched and propagates it at output. Variable should appear on right-hand-side of an expression. A variable should not be a failure type and any value given on input token other than failure type is treated as a variable. Normal behavior of a component means ’no failure’ and it is denoted by ‘*’. The four possible FPTC behaviors of component are presented below:

- late $\rightarrow$ * $(Sink)$
- * $\rightarrow$ early $(Source)$
- late $\rightarrow$ late $(Propagate)$
- omission $\rightarrow$ value Subtle $(Transform)$

#### Semantics

The architectural model (graph), constituted of inter-connected components, is interpreted as a token-passing network. Tokens corresponding to failure modes are introduced by sources, transformed and
propagated by the network nodes, and removed by sinks. Token sets, used to annotate the arrows of the architectural model, represent the flows of failures. They contain all possible failures that can be propagated along a dependency path. To calculate the failure behavior, the maximal token set on any flow in the network is calculated by performing a fixed-point calculation defined by Wallace [7].

The whole process to obtain the failure behavior can be summarized through the following steps:

1. Begin by assuming that every connection between components consists of a single token, and have normal (means having no failure) behavior. Failures flow along with dependency path.
2. For every component, c:
   - Determine which, expression may be applied from the failure behaviors defined by c. For example try to find at-least one expression whose left-hand side matches with the input failure(s).
   - Apply the selected expression. The new set of failure produced by c is calculated as the union of the existing set of failures produced by c and the right-hand side of the selected expression [8].
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.

![Figure 2.17: Simple System](image)

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.

![Figure 2.18: Result of FPTC technique](image)
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, CFTs and SEFTs are manual tool guided techniques.
- If failure behavior of system is calculated from its building components then impact of change of a component on the system is easily determined and enhancement of the system will be cheaper.
- FPTC is qualitative analysis technique.
- FPTC carry out failure propagation on systems with cycles by using fixed-point analysis.
Chapter 3

PROBLEM FORMULATION AND ANALYSIS

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

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 be transformed?
- which information should be back-propagated?
- which classes and methods should be created for performing FPTC analysis?
Chapter 4

METHOD

This chapter presents methods to find the solution of problem by selecting proper analysis technique. This chapter is organized as follow: Section 4.1 briefly explains reasoning behind selection of FPTC analysis technique after making the comparison of various analysis techniques. Section 4.2 highlights the ambiguities in FPTC syntax. Section 4.3 presents the integration aspects of CHESS-FPTC plug-in in CHESS tool-set. In section 4.4, motivations behind selection of ACCELEO transformation engine are presented.

4.1 Selection of Proper Failure Behavior Analysis Technique

FPTC analysis technique is selected 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 CFTs only represent event-based failure behavior, whereas SEFTs and AADL’s Error Annex describe both states-based and event-based behavior. CFTs and SEFTs represent only graphical modeling; however, FPTN, FPTC and AADL represent 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 which have to be identified. 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 SEFTs do not support them. AADL and FPTC both support cyclic loop but CFTs and SEFTs do not support cyclic loop. AADL’s Error Annex and Hip-HOPS provides good support for modeling architectural specifications including architectural dependencies. But other modeling formulism are general failure propagation languages and do not target the specific architecture. AADL supports quantitative as well as qualitative evaluation. But HiP-HOPS, CFTs and SEFTs support quantitative evaluation. FPTN and FPTC are qualitative analysis techniques.

Tool support: Good theoretical and failure behavior analysis techniques are not useful if they are not supported by a tool. Additionally, if complex systems are considered, there are many chances of errors in manual analysis techniques. AADL with error Annex, Hip-HOPS and FPTC support automatic generation of error models for hierarchical software components but CFTs, SEFTs are manually tool-supported.

It is concluded that FPTC technique is useful to analyze failure behavior of high integrity real-time component-based embedded dependable 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. FPTC is more robust than other techniques and it support qualitative evaluation.

4.2 Understanding and Comparison of FPTC rules defined by different authors

After studying and understanding FPTC analysis technique, it is 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 into one expression Sourceforge [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 to 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:

\[
(late, _) \rightarrow (omission, value Subtle)
\]

\[
(late, commission) \rightarrow (*, *)
\]

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 materials but method for calculating specificity is only described in Sourceforge [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 cannot be null, 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 or 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 late is occurring as first input failure and ‘*’ is present as second input failure. Similarly, other expression can be selected due to presence of ‘*’ as first input failure and commission as second failure.
According to Sourceforge [8], if a component has multiple incoming connections, expression(s) may be applicable for each combination of inputs. The cross product of these inputs is calculated, for producing a set of expressions. In a component, 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 Sourceforge [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, it is concluded that wildcard cannot be used on the RHS of the expression, because usage of wildcard on RHS of expression is pessimistic. For calculating specificity of wildcards and variables above algorithm should be used in the implementation. Each component must have at least one expression (rule). For handling multiple incoming inputs, combination of input failure regarding each input port should be calculated.

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

To 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, CHESS editor and CHESS views.
2. Identify how to extend CHESS-ML to cover FPTC concepts.
3. Identifying available and missing features in CHESS tool-set which are necessary for implementation of FPTC analysis.

### 4.4 Selection of Transformation Technology

Proper transformation engine must be selected for providing solution of the problem because aim of the CHESS project is to engineer high integrity real-time component-based dependable embedded systems by using MDE approach. Therefore, a technique which supports Eclipse platform and can be integrated with CHESS tool-set should be selected.
4.4.1 Why we select ACCELEO for M2T

ACCELEO as transformation technique to perform model-to-text transformation is selected because it follows MDA principles including increased reliability and quality of code. ACCELEO requires 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 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 of usability and maintainability.

Format of Input xml file: After selecting ACCELEO technique for M2T transformation, another issue to be resolve is to identify the format of target file (input XML) which must be appropriate for calculating the behavior of the system. Following information should be enclosed in input XML file:

➢ Name of the system at which analysis will be performed, input port(s) with corresponding input failure(s) and output port(s) of the system.
➢ Name of sub-component, input port(s), output port(s), transformation rules/expressions for every sub-component used in the system.
➢ For every connector used in system, source and target ports are identified.

4.4.2 Back Propagation

Back propagation is necessary for displaying the analysis result of the system at initial model, through which user can track output failure(s) of each component. For performing back propagation, each out-going port of a component should be annotated with comment having FPTC specification stereotype.
Chapter 5

SOLUTION

This chapter presents solution of the problem (discussed in chapter 3) by integrating analysis technique (FPTC) within CHESS. The chapter in particular, is organized in the following way: Section 5.1 illustrates modified syntax of FPTC. Section 5.2 describes meta-modeling concepts to extend CHESS ML. In section 5.3, the design of CHESS-FPTC plug-in is described. Section 5.4 explains the implementation details of CHESS-FPTC plug-in.

5.1 Proposed FPTC
As discussed in chapter 4, inconsistencies exist in FPTC syntax and rules. Therefore, after understanding of FPTC technique and making comparison between available literatures, the inconsistencies that exist in FPTC syntax are removed. So, the following solution regarding FPTC syntax is proposed.

5.1.1 Multiple incoming failures on Input Ports
If multiple input failures are coming on input port(s) of the component, the combination of these input failures 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 noFailure 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.

5.1.3 Usage of Wildcard
User should not use wildcard on RHS of expression (rule) in our FPTC, according to comparison proposed in section 4.2.1. Instead of using ‘_’ (underscore) symbol ‘wildcard’ is used as it is.

5.1.4 Specificity of Wildcards and Variables
In case of more than one matching patterns (LHS of transformation rule) with incoming failures, more specific transformation rule is selected. For calculating specificity of wildcards and variables, an algorithm is written which is described in section 4.2. Source code written for selecting more specific transformation rule is presented section 5.4.

5.1.5 Modified Syntax of FPTC
The syntax of FPTC analysis technique, proposed by Wallace (2005) is modified by removing inconsistencies. To avoid ambiguity, term failure is used instead of fault. The term ‘Value’
represents failure; instead of value failure ‘valueSubtle’ or ‘valueCoarse’ is used. After removing the ambiguities, modified syntax of FPTC is presented below:

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

5.2 Meta-Modeling Concepts Provided to Extends CHESS ML

CHESS dependability profile allows enriching CHESS ML model which is related to safety and dependability. The definition of CHESS dependability profile begins from identification of dependability features/properties addressed by UML profile which relates dependability analysis of hardware and software systems. Language extensions (UML, MARTE, Sys ML profiles) are 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 related to FPTC analysis 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.

For perform FPTC analysis Components are stereotyped with ComponentType (it maps the component stereotype of the CHESS component model) and ComponentImplementation (maps the component implementation stereotype of the CHESS component model) stereotypes. The behaviors of a component (rules) are described in ‘ComponentImplementation’. Realization relation is drawn from ComponentImplementation towards ComponentType.

5.2.2 Components

Components are the basic blocks of the system. In CHESS model, 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 towards failures). From dependability point of view, component is affected by failure(s) [27].

5.2.3 Interfaces

Components are connected through their interfaces (specified in a ComponentView of CHESS), and they communicate in order to implement the system’s functionality. Components have required and provided interfaces. An interface has operation(s) which contributes in identifying input and output port(s) of the component.
5.2.4 Ports

CHESS component model supports modeling of software and hardware systems. Software components are decorated with Client Server Port(s) and hardware components are decorated with Flow Port(s).

- **Client Server Port**: For performing FPTC analysis at software level, software components are decorated with Client Server Ports. *ClientServerPort* has attributes named as *reqInterface* and *proInterface* for specifying required and provided interfaces.

- **Flow Port**: Hardware components are decorated with flow port(s) for performing FPTC analysis at hardware level. It allows the information flow between hardware components such as its direction.

5.2.5 Comments

- **FPTC**: stereotype is used for decorating a comment with set of expression/rule for a component. In FPTC stereotype, an expression is written as a string in fptc attribute. For this stereotype annotated element is ComponentImplementation. Another possibility is to decorate component implementation with FPTC stereotype (instead of decorating comment with FPTC stereotype).

- **FPTCSpecification**: stereotype is specified in ExtraFunctionalView and is applied at comment(s) which are annotated with every outgoing port of composite as well as its child components. Moreover, input failure(s) are provided by annotating input port(s) of composite component with comments having FPTCSpecification stereotype. In FPTCSpecification stereotype, used attributes are *failure* (in failure user can select different type of failure(s) which are relevant to FPTC analysis) and *partWithPort* (is used to specify the instance).

- **Assign**: stereotype is specified in DeploymentView of CHESS and it is used for allocation of failure(s) from software system to hardware system. It has two important attribute named as “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 Platform

- **failurePropagationAnalysis**: It allows to specify the target platform instance specification for executing FPTC analysis. failurePropagationAnalysis stereotype is specified in DependabilityAnalysisView (MARTE::GQAM::GaAnalysisContext). Platform attribute in failurePropagationAnalysis stereotype is used for selecting target platform instance at which analysis should be performed.

5.2.7 Software/Hardware Systems

- **CHGaResourcePlatform**: Extends the MARTE::GQAM::GaResourcePlatform and is applied at hardware and software systems. In ComponentView, CHGaResourcePlatform is used for decorating software system(s) and in DeploymentView it is used for decorating hardware system(s).
5.3 Proposed Design

Our design concentrations of FPTC plug-in is not just limited to implementation of failure propagation analysis but this plug-in must be compliant with CHESS methodologies and supported in CHESS platform.

5.3.1 Implementation of FPTC Analysis in Progress IDE

FPTC Analysis is implemented in PRIDE (Progress IDE), which is a project of Mälardalen University [25]. Architectural design of FPTC analysis (in PRIDE) is shown in Figure 5.1.

In PRIDE, ProCom component model is used for modeling of system(s). ProCom model is based on Ecore meta-model. M2T transformation is performed in ProCom model by using ACCELEO transformation engine (version 2.3) which produces FPTC input (XML file). Information enclosed in XML file is used for performing FPTC analysis. In ProCom, results of FPTC analysis are produced in an output XML file.

![Figure 5.1: FPTC Analysis in ProCom](image)

5.3.2 Design of the CHESS-FPTC Plug-in

In CHESS, system is modeled by using CHESS modeling elements. CHESS modeling language is composed of UML, MARTE and Sys ML languages. CHESS model is transformed into FPTC input (XML file) by using ACCELEO transformation engine (version 3.1). Previous ACCELEO transformation code cannot be re-used due to different models, modeling elements and version of ACCELEO engine. In CHESS, transformation results are propagated back at initial model(s) which is not supported in PRIDE.

![Figure 5.2: Design of CHESS-FPTC plug-in](image)
CHESS-FPTC plug-in is composed of four packages as shown in Figure 5.3. We have inherited some design decisions and re-used some source code (more specifically, classes enclosed in se.mdh.chess.fptc.analysis.impl and se.mdh.chess.fptc.analysis.launch.xml packages) after modifying and adding some new methods to make it compliant with our modified FPTC syntax.

- **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 to make the FPTC call command enable or disable (call is only enabled in DependabilityAnalysisView).
  - **Analysis Command:** FPTCAnalysisCommand class is activated upon executing FPTC command from eclipse run-time configuration. In this class, IProgressMonitor is used for monitoring the 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), input and output port(s) and connections. This information is stored (by using 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 generates an output XML file which contains analysis results (failure behavior of system as well as its child components).

- **se.mdh.chess.fptc.analysis.launch.xml** package contains multiple classes (DataPort, FaultPattern, InputFault, InputFaultCombination, InputPortFaultVal, SoftwareComponent, TransformationRule) which are used for storing and retrieving corresponding information; information is stored in data structures.

- **se.mdh.chess.fptc.analysis.transformation** package contains ACCELEO templates and java classes for performing M2T transformation.
  - **Transformation:** ACCELEO templates (.mtl extension files) are used for writing M2T code. Separate ACCELEO templates are used for writing M2T code for software system (XML_SW.mtl) and hardware system (XML_HW.mtl). Main ACCELEO template is used for identifying whether selected platform belongs to
XML_SW (ComponentView) or XML_HW (DeploymentView); based on the selection, main template further activates corresponding template for performing M2T transformation.

5.3.3 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 5.4.
Figure 5.4: 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. This plug-in contains the support for the decoration of the CHESS modeling elements with failure propagation stereotypes.

- **it.unipd.chess.editor** plug-in contains the support for CHESS Editor. FPTC analysis is performed at CHESS model which is modeled by using CHESS editor.

- **it.unipd.chess** plug-in contains the support for creating CHESS Project, adding CHESS Nature, applying CHESS Views and Profiles to the project.

### 5.4 Implementation

Implementation of FPTC analysis requires modeling of the system(s) by using CHESS modeling elements. The implementation of CHESS-FPTC plug-in is structured in several activities (as shown in Figure 5.5), which are summarized below:

1. According to the CHESS methodology user can only launch the analysis from DependabilityAnalysisView. For reducing chances of mistakes during invocation of FPTC Analysis, Activate/deactivate of FPTC call command is implemented.
2. Launch FPTC analysis is concerned with executing FPTC analysis.
3. FPTC analysis is performed on component’s instances; Call Build Instance Command is concerned with generating instances of CHESS modeling elements (which are used in the model).
4. Generate Input XML file is concerned with performing M2T transformation and generation of input XML file from user model.
5. Generated XML file is used as input in read Input XML file; InputXMLFileReader.java class reads generated XML file.
6. After reading Input XML file, apply propagation is performed at which failure(s) are propagated towards output port(s) of the components.
7. An output XML file is generated which contains results of propagation.
8. Finally, analysis results are propagated back at the user model(s).
5.4.1 Modeling Systems within the CHESS tool-set

ComponentView allows us to model software systems according to CHESS model definition. First step involved in modeling of a system is to create a new class diagram (named as SW_Components), which contains components stereotyped with ComponentType or ComponentImplementation, interfaces and relationships among them as shown in Figure 5.6.
Each interface contains a set of operations. Kind of interface and direction of operation(s) is used to distinguish whether a ClientServerPort is an input or output port as elaborated in Table 5.1.

<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>Input Port</td>
<td>Provided</td>
</tr>
<tr>
<td>Output Port</td>
<td>Required</td>
</tr>
<tr>
<td>Output Port</td>
<td>Provided</td>
</tr>
</tbody>
</table>

Table 5.1: Distinctive Table for ClientServerPorts

ComponentImplementation types are decorated with transformation rules. In FPTC Stereotype, transformation rules are specified in the fptc attribute. Owned comment of ComponentImplementation can be stereotyped with FPTC as shown in Figure 5.7; another possibility is to apply FPTC stereotype at ComponentImplementation (as shown in Figure 5.14).
System is modeled after decorating each component implementation. Stereotype named as CHGaResourcePlatform is used for specifying composite component or system as shown in Figure 5.8. Each input port of a composite component or system is annotated with a comment having FPTCSpecification stereotype, which contains list of incoming failure(s) inside failure attribute; this information is used as input for performing FPTC analysis. Each output port of a component is annotated with comment (which is stereotyped with FPTCSpecification) for displaying analysis results.

![Figure 5.8: Modeling of Composite Component or System](image)

DeploymentView is used for modeling of hardware systems and for assigning software instances to hardware instances. Hardware system is shown in Figure 5.9.

![Figure 5.9: Hardware System](image)

After decorating 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, FPTC analysis must be launched from DependabilityAnalysisView. Implementation of activate/deactivate of call command is a step towards making failure propagation analysis error prone by reducing chances of mistakes.
Plugin.xml file contributes in activate or deactivate of FPTC call command based on certain checks enclosed in visibleWhen tags as shown in Figure 5.10.

```xml
<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
                    value="org.eclipse.uml2.uml.editor.presentation.UMLEditorID">
                </equals>
            </or>
        </with>
    </visibleWhen>
</command>
```

**Figure 5.10: Activate/Deactivate Call Command**

CHESSProfileManager class contains information of CHESS-Views. Call command is activated upon matching of current view with DependabilityAnalysisView (as shown in source code placed in Figure 5.11). If current view is not DependabilityAnalysisView then call command remains deactivated.

```java
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);
        }
        refreshElements();
    } catch (Exception e) {
        return;
    }
}
```

**Figure 5.11: Use of CHESSProfileManager for matching Current View**

### 5.4.3 Launch FPTC Analysis

For launching FPTC Analysis, component is stereotyped with FailurePropagationAnalysis (at DependabilityAnalysisView) and information concerning target platform instance is specified in platform attribute (as shown in Figure 5.12).
5.4.4 Call Build Instance

Call build instance command is used for generating instances of composite component/system. According to CHESS methodology, information enclosed in build-instances package should be used for performing analysis. After execution of FPTC analysis, Call Build Instance command is executed for reducing the chances of mistakes by getting latest version of instances.

5.4.5 Read Model and Generate XML File

ACCELEO engine is used for transforming CHESS model into XML file. By using ACCELEO M2T engine, required information for execution of FPTC analysis is fetched from CHESS model and printed in XML file. Some transformation rules which are written in ACCELEO templates for retrieving information (in corresponding XML tags) from different CHESS modeling elements are presented below:

- **Platform**
  DependabilityAnalysisView is used for selection of target platform.
  - **CHESS Modeling Element:**
    In FailurePropagationAnalysis stereotype, platform attribute is used for selection of target platform(s) as shown in Figure 5.12.
ACCELEO Code:

```java
[if(subpackage.name.str("DependabilityAnalysisView"))]
[for(element:Element | subpackage.ownedElement())]
[for(str:Stereotype | element.getAppliedStereotype("CHESS::Dependability::FailurePropagation::FailurePropagationAnalysis"))]
[let selectedPlatform String = element.getValue(str, "platform").toString()]
[for(platform:CHGResourcePlatform | CHGResourcePlatform.allInstances())]
[[if(comp_package.name.matches("modelComponentView"))]
[XML_SW(comp)]
[elseif(comp_package.name.matches("modelDeploymentView"))]
[XML_HW(comp)]
[/for]
[/let]
[/for]
[/for]
[/if]
```

**Result:** ACCELEO code identifies that whether selected platform belongs to ComponentView or DeploymentView and activates either XML_SW or XML_HW template. XML_SW template is used for generating input XML file for software system (which is decorated in ComponentView). In other case, XML_HW template is used for generating input XML file for hardware system (which is decorated in DeploymentView).

**Transformation Rule**

Transformation rules are enclosed in `fptc attribute` within `FPTC` stereotype.

**CHESS Modeling Element:**

Transformation rules are enclosed inside owned UML comment of ComponentImplementation which is stereotyped with `FPTC` as shown in Figure 5.7. Another possibility is to decorate a ComponentImplementation with `FPTC` stereotype as shown in Figure 5.14. In `FPTC` stereotype, transformation rules are written in `fptc` attribute.

![Figure 5.14: Component Implementation](image)

**ACCELEO Code:**

ACCELEO code is written to verify that whether FPTC stereotype is applied at ComponentImplementation and it’s `fptc` attribute is not an empty string; upon verification, ACCELEO code written inside `if` block is executed. Otherwise, `else` block is executed which search owned comment of a ComponentImplementation which is stereotyped with FPTC.
Once, fptc string is found, it is further break down into transformation rule(s) based on semicolon.

\[\text{if}(\text{rule}.get\text{AppliedStereotype}\text{(CHESS::Dependability::FailurePropagation::FPTC)})\neq \text{null} \text{ and } (\text{comp.}
\text{getValue}\text{(comp.get\text{AppliedStereotype}\text{(CHESS::Dependability::FailurePropagation::FPTC)}, \text{'}fptc\text{'}),null})\]
\[\text{for}(\text{rule}\text{String} | \text{comp.get}\text{Value}(\text{comp.getAppliedStereotype}(\text{CHESS::Dependability::FailurePropagation::FPTC}), \text{'}fptc\text{'}),\text{toString}())\]
\[\text{TransformationRules}(\text{rule})\]
\[\text{else}\]
\[\text{for}(\text{rule}\text{String} | \text{comp.ownedComment}.\text{getValue}(\text{str}, \text{'}fptc\text{'}),\text{toString}())\]
\[\text{TransformationRules}(\text{rule})\]
\[\text{if}\]
\[\text{if}\]

\textbf{Result:}
\textbf{TransformationRules}
\textbf{Expression lhsPattern}="\text{Middle_R1_impl.commission}" \textbf{rhsPattern}="\text{Middle_P1_impl.early}" /
\textbf{Expression lhsPattern}="\text{Middle_R1_impl.early}" \textbf{rhsPattern}="\text{Middle_P1_impl.late}" /
\textbf{Expression lhsPattern}="\text{Middle_R1_impl.omission}" \textbf{rhsPattern}="\text{Middle_P1_impl.early}" /
\textbf{TransformationRules}

\textbf{Connectors}
Connectors are used for connecting components with each other. Each connector has source and destination ends.

- **CHESS Modeling Element:** Connectors are used for connecting components inside composite component or system as shown in Figure 5.8.
- **ACCELEO Code:**
  Information concerning connections is retrieved from InstanceSpec package which is generated through build-instance command and selected as target platform in FailurePropagationAnalysis stereotype (at DependabilityAnalysisView).

\textbf{Generated Text:} Information about connections inside composite component/system is listed inside connections tags. Each connection attribute contains information about source and destination ends. Information about source and destination owner name is used for identifying unique ends (in presence of multiple same instances).
FlowPort

In CHESS model, Flow Ports (MARTE::Design_Model::GCM::FlowPort) are used for decorating hardware components

- CHESS Modeling Element:
  In DeploymentView, FlowPort is used for decorating hardware components as shown in Figure 5.9.

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

ClientServerPort

ClientServerPort (MARTE::Design_Model::GCM::ClientServerPort) is used for decorating ports of software components.

- CHESS Modeling Element
  In ComponentView, ClientServerPort is used for decorating software components as shown in Figure 5.7.

- ACCELEO Code
  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 directions of operations in interface are out or, kind of interface is provided and directions of operations enclosed in interface are in then port is an input port as shown in table 5.1.
5.4.6 Read Generated Input XML File

InputXMLFileReader.java class is concerned with reading XML file, new instance of DocumentBuilder class is used for obtaining XML 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 input XML file.

```java
//Parsing the xml File
DocumentBuilderFactory builder = DocumentBuilderFactory.newInstance();
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. The getElementsByTagName method is used for retrieving root element. Information enclosed in attributes of composite component is retrieved by using getAttribute method as shown in below code:
Input ports of the composite component are fetched by reading InputPorts tag. Moreover, attributes of each input port are retrieved by using for loop (such as Name and inputValues). After reading input port(s), output port(s) of the composite component are retrieved by using similar approach.

```java
NodeList component = rootElement.getElementsByTagName("Components");
for (int i = 0; i < component.getLength(); i++) {
    Node componentGraphNode = component.item(i);
    Element componentGraphElement = (Element) componentGraphNode;
    String compositeName = componentGraphElement.getAttribute("compositeName");
}
```

Information about inner child component(s), their input and output data port(s) is also retrieved in a similar way. Child component(s) of composite component have transformation rule(s). Transformation rule(s) are read by getElementsByTagName method. Each expression contains one transformation rule, which is further divided into left-hand-side (LHS) and right-hand-side (RHS). Information enclosed in each expression is obtained by using for loop as shown in source code below:

```java
// read main input values
NodeList inputEntries = componentGraphElement.getElementsByTagName("InputPorts");
if (inputEntries.getLength() > 0) {
    Node inputEntryNode = inputEntries.item(0);
    Element inputEntryElement = (Element) inputEntryNode;
    NodeList inputs = inputEntryElement.getElementsByTagName("inputPort");
    for (int i = 0; i < inputs.getLength(); i++) {
        DataPort dataPort = new DataPort(_composite);
        Node inputNode = inputs.item(0);
        Element inputElement = (Element) inputNode;
        String inputName = inputElement.getAttribute("name");
        String inputValuesStr = inputElement.getAttribute("inputValues");
    }
}
```

```java
// Reads Transformation Rules
NodeList transformationRules = componentGraphElement.getElementsByTagName("TransformationRules");
for (int k = 0; k < transformationRules.getLength(); k++) {
    Node constraintEntryNode = transformationRules.item(0);
    Element constraintEntryElement = (Element) constraintEntryNode;
    NodeList constraints = constraintEntryElement.getElementsByTagName("Expression");
    for (int i = 0; i < constraints.getLength(); i++) {
        TransformationRule rule = new TransformationRule();
        Node constraintNode = constraints.item(0);
        Element constraintElement = (Element) constraintNode;
        String lhsPattern = constraintElement.getAttribute("lhsPattern");
        if (lhsPattern.equals("")) {
            String[] lhsDataPorts = lhsPattern.split(",");
            for (String inputPort : lhsDataPorts) {
                int endIndex = inputPort.indexOf(",");
                String portName = inputPort.substring(0, endIndex);
                for (DataPort inputDataPort : softcomp.getInputDataPorts()) {
                    if (inputDataPort.getName().matches(portName))
                        rule.parse_lhsPattern(portName, inputDataPort.getVarName());
                }
            }
        } else {
            rule.setLhsPattern(rule.parsedLhsPattern());
        }
    }
```
For obtaining corresponding port name(s), LHS or RHS of expression is break down into multiple pieces based on comma (,); each retrieved string is further break down based on the index of dot (.)

A connection between two ports is obtained by reading Connection tag. For loop is used for getting each connection. In connection, srcName attribute represents start end of connection and destName is used for indicating destination end of connection. Information enclosed in srcOwnerName and destOwnerName attributes is useful for identifying unique port(s) in presence of multiple same instances.

```java
// Read connections between inputs and output ports
NodeList connectionsList = rootElement.getElementsByTagName("Connection");
for (int i = 0; i < connectionsList.getLength(); i++)
{
    // get a connection at a time
    Node connectionNode = connectionsList.item(i);
    Element connectionElement = (Element) connectionNode;
    String src = connectionElement.getAttribute("srcName");
    String srcOwner = connectionElement.getAttribute("srcOwnerName");
    DataPort srcPort = _composite.findDataPort(src, srcOwner);
    String dest = connectionElement.getAttribute("destName");
    String destOwner = connectionElement.getAttribute("destOwnerName");
    DataPort destPort = _composite.findDataPort(dest, destOwner);
    srcPort.addConnectedPort(destPort);
    destPort.addConnectedPort(srcPort);
}
```

Information read from input XML file is stored in separate java classes (SoftwareComponents, DataPorts and TransformationRules) by using data structures; this information is used during apply propagation process.

5.4.7 Apply Propagation

This class is concerned with calculating input combinations, check applicability of transformation rule, identifying more specific transformation rule and apply transformation rule; implementation details of these methods is presented below:

- **Compute input combinations:**

  Each component must contain at-least one input and output port for performing FPTC analysis. If component contains more than one input ports or more than one failure on single input port, then possible combinations of failures are computed through permutation to be compliant with left-hand-side of transformation rule(s). Source code written for computing input combinations is written below:
Check Applicability of Transformation Rule:

Each component must contain at least one transformation rule and left-hand-side of at least one transformation rule should match with computed input combination. Each computed combination is matched with left-hand-side of transformation rule(s). If combination is not matched with left-hand-side of any transformation rule then this combination is discarded. Source code for checking applicability of transformation rule is presented 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.addInputFault(new InputFault(inputPort, fault));
        }
        if (curCombinations.isEmpty())
            return new ArrayList<InputFaultCombination>();
        combinations = computeCombinations(combinations, curCombinations);
    }
    return combinations;
}
```

Selecting More Specific Transformation Rule

Each component must have at least one matching pattern. Possibility exists that component may have more than one matching patterns. In case of more than one matching patterns, more specific transformation rule(s) is selected as shown in below code:

```java
private boolean matchTransformationRule(TransformationRule rule,
    SoftwareComponent comp, InputFaultCombination combination) {
    // check that it is applicable
    Map<String, 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;
}
```

If none of the combination is matched with left-hand-side of any transformation rule then input failures are propagated to the output port(s) of the component.

Selecting More Specific Transformation Rule

Each component must have at least one matching pattern. Possibility exists that component may have more than one matching patterns. In case of more than one matching patterns, more specific transformation rule(s) is selected as shown in below code:
Apply transformation rule

Once matched transformation rule(s) is found, failure(s) at right-hand-side of transformation rule are moved towards corresponding output port(s) of a component as shown in below code:

```java
private boolean applyTransformationRule(TransformRule rule, SoftwareComponent comp, InputFaultCombination combination) {
    // 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 = varMapping.get(variable);
            if (fault != null)
                outputDataPort.addFault(fault);
            continue;
        }
    }
    return true;
}
```

5.4.8 Propagation of Analysis Results Back into Model

Results of the FPTC analysis are propagated back into the initial model. For this purpose, each output port (of composite component as well as it’s inner child component(s)) is annotated with the comment stereotyped with FPTCSpecification. In FPTCSpecification stereotype, failure attribute is used for specifying failure(s) on port and partWithPort attribute is used for specifying instance of a port.

After performing FPTC analysis, output failure(s) of a component are added into the failure attribute. In CHESS model, Software Component(s) are modeled in ComponentView. So, initial step is to get the ComponentView by using getPackagedElement method. Later, all elements inside ComponentView are fetched and stored in a list and a loop is used for finding the corresponding composite component from the list as shown in below code.
Once, a composite component is found, another method named PropagateBackComp is called for adding the resulting failure(s) on output port(s). PropagateBackComp method requires parameter value for SoftwareComponent and is called for composite component as well as its inner child component(s). In PropagateBackComp method, GetComment method is called for getting the comment (annotated with output data port of a component). Once comment is found, previous failure(s) on failure attribute are erased and new failure(s) are added as shown in below code:

```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 failurertype : outputDataPort.getFaults()) {
                failure.add(failurertype);
            }
        }
    }
}
```

In PropagateBackComp method, GetComment method is called for every DataPort of a component for getting the corresponding comment. For selecting right comment, three checks are performed: in first check, it is identified whether comment is stereotyped with FPTCSpecification, in next check DataPort name is matched with annotated element name and finally, instance name is matched with partWithPort attribute as shown in below code:

```java
public static Comment GetComment(SoftwareComponent component, DataPort outputDataPort, CHESSEditor editor) {
    for (Comment comm : component.getOwnedComments()) {
        if (comm.getAnnotatedElements().tostring().contains("name: "+outputDataPort.getName()))
            if (component.getComponentName().matches(component.getName())){
                fptcspecification = comm.getApplicableStereotype(FPTCSpecification);
            return comment = comm;
        }
        else if (comm.getValue(fptcspecification, "partWithPort").tostring().contains
            (component.getComponentName().replaceAll(comp.getName()+"_", "name: "))
            return comment = comm;
    }
    return CreateComment(component, outputDataPort, editor);
}
```

If comment is not found, then it is programmatically created by using CreateComment method. For making, created comment compliant with corresponding output DataPort, comment is stereotyped with FPTCSpecification by using comment.applyStereotype method. Once FPTCSpecification stereotype is applied, attribute partWithPort is set by using comment.setValue method and annotation link between comment and output DataPort is created by using comment.getAnnotatedElements.add method as shown in code below:
After programmatically creating comment and making it compliant with corresponding output DataPort, comment is dropped in composite component diagram by using below code.

```java
protected void doExecute() {
    comment = comp.createComponentComment();
    Stereotype fptcSpec = comment.getApplicableStereotype(FPTCSpecification);
    comment.applyStereotype(fptcSpec);
    Object partWithPort = null;
    Port annotation = null;
    for(Property property : comp.getAllAttributes()){
        if(component.getComponentName().contains(property.getName())){
            partWithPort = property;
            Component property_Type = (Component) property.getPropertyType().
            for(Port port : property_Type.getOwnedPorts()){
                if(port.getName().equals(outputDataPort.getName()))
                    annotation = port;
            }
        }
    }
    comment.setValue(fptcSpec, "partWithPort", partWithPort);
    comment.setAnnotatedElements().add(annotation);
}
```

After programmatically creating comment and making it compliant with corresponding output DataPort, comment is dropped in composite component diagram by using below code.

```java
public static void DropCommentToDiagram(CHESSEditor editor, Object pageManager, SoftwareComponent component){
    // Drop comments to the editor
    DropObjectsRequest dropObjectsRequest = new DropObjectsRequest();
    ArrayList<DropObjectsRequest> list = new ArrayList<DropObjectsRequest>(){
        list.add((org.eclipse.uml2.uml.Comment)comment);
        dropObjectsRequest.setObjects(list);
        dropObjectsRequest.setLocation(new Point(700,100));
        diagramEditor.getGraphicalViewer().getEditPartRegistry().put(component.getCompVarName(),
        pageManager);
        Command commandDrop = editor.getDiagramEditorPart().getCommand(dropObjectsRequest);
        diagramEditor.getDiagramEditorDomain().getDiagramCommandStack().execute(commandDrop);
    }
```

After performing FPTC analysis and back propagation of analysis results into the user model, failure(s) are displayed in comment (annotated with every output port of a component as shown in Figure 5.14).
Figure 5.14: The analysis results are propagated back on the model
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 [33, 34, 35 and 36].

The AAL2 is an adaptation layer type on top of an Asynchronous Transfer Mode (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 [33].

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 into 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 (Protocol Data Unit). STC services also provide the assurance of error free transformation of data and delivery of data in a sequence [34].

- 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

Figure 6.2 summarizes the experiment’s activities though an activity diagram. The first activity consists in modeling the AAL2 signaling protocol in CHESS FunctionalView. The second activity consists in decorating the components to add the information required for the analysis (CHESS Extra-FunctionalView). In the CHESS DependabilityAnalysisView, the analysis is invoked (third activity) on the desired platform. The FPTC analysis result is back-propagated and then it can be interpreted.
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.

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 modeled. Moreover, relations 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 (ServedUser_Impl) component implementation type:**
  - *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.
    
    *noFailure* $\rightarrow$ *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.
    
    *noFailure* $\rightarrow$ *omission*

  - *ServedUser* gets correct information concerning an AAL2 on time on the input port and behaves correctly.
    
    *noFailure* $\rightarrow$ *noFailure*
- 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.

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

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

\[ \text{noFailure} \rightarrow \{\text{omission, valueCoarse, late, noFailure}\} \]

- 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.

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

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

![Figure 6.4: Served User at component Implementation level](image)

- Failure behavior of SignalingStackMultiplexer (SignalingStack_Mux_Impl) component implementation type:
  - SignalingStack_Mux behaves properly.

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

- 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.

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

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

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

- 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.

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

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

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

- \( \text{SignalingStack\_Mux} \) 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.

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

- \( \text{SignalingStack\_Mux} \) receives correct information in input but it does not maintain correctly the status of the AAL2 connections.

The behavior \( \text{SignalingStack\_Mux} \) component is defined at component implementation level \( \text{SignalingStack\_Mux\_Imp} \) as shown in Figure 6.5.

![Figure 6.5: Signaling Stack Multiplexer Implementation](image)

- Behavior of \( \text{SignalingTransportConverter} \) (STC):
  - \( \text{SignalingTransportConverter} \) receives correct information in input but waits to send signals to underlying transport network when network is crowded which causes the delay of delivery.

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

- \( \text{SignalingTransportConverter} \) behaves properly.

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

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

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

- \( \text{SignalingTransportConverter} \) receives correct but late information in input and being unable to reduce the delay it propagates a late failure.
late \rightarrow \text{late} \\
\begin{itemize}
  \item \textit{SignalingTransportConverter} receives commission sequence failure and tries to manage the sequence numbering and behaves like a sink (it transforms commission into noFailure).
  \end{itemize}
commission \rightarrow \text{noFailure} \\
\begin{itemize}
  \item \textit{SignalingTransportConverter} receives omission sequence failure and tries to manage the sequence numbering and succeeds in getting the missing information, however not in time. \textit{SignalingTransportConverter} succeeds in mitigating the failure.
  \end{itemize}
\text{omission} \rightarrow \text{late}

The behavior \textit{Signaling Transport Converter} component is defined at component implementation level \textit{SignalingTransportConverter _Impl} as shown in Figure 6.6.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6_6.png}
\caption{Figure 6.6: Signaling Transport Converter Implementation}
\end{figure}

\begin{itemize}
  \item The behavior of \textit{SignalingStack_DeMux}:
    \begin{itemize}
      \item \textit{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.
      \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textit{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 \textit{noFailure} into a \textit{valueCoarse} failure.
  \end{itemize}

\begin{itemize}
  \item \textit{SignalingStack_DeMux} behaves properly.
  \end{itemize}

\text{noFailure} \rightarrow \text{noFailure}

The above two rules must be written as:
\[\text{noFailure} \rightarrow \{\text{valueCoarse, noFailure}\}\]

- \text{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.

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

- \text{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 \text{SignalingStack\_DeMux} component is defined at component implementation level \text{SignalingStack\_DeMux\_Impl} as shown in Figure 6.7.

\[\begin{array}{|c|}
\hline
\text{«Component»} \\
\text{SignallingStack\_DeMux\_Impl} \\
\hline
\text{«fPTC»} \\
\{ fptc=SSD\_Impl\_R1.omission \rightarrow SSD\_Impl\_P1.omission \}
SSD\_Impl\_R1.noFailure \rightarrow SSD\_Impl\_P1.valueCoarse, SSD\_Impl\_P1.noFailure
\text{SSD\_Impl\_R1.late} \rightarrow SSD\_Impl\_P1.late
\hline
\end{array}\]

\[\begin{array}{|c|c|}
\hline
+ SSD\_Impl\_R1: \text{SignallingStack\_DeMux} \\
+ SSD\_Impl\_P1: \text{SignallingStack\_DeMux} \\
\hline
\end{array}\]

**Figure 6.7: Signaling Stack De-Multiplexer Implementation**

- The Behavior of \text{UnderlyingSignalingTransport}:

- \text{UnderlyingSignalingTransport} behaves properly.

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

- \text{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:

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

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

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

- \text{UnderlyingSignalingTransport} receives correct but late information in input and being unable to reduce the delay it propagates a late failure.

\[\text{late} \rightarrow \text{late}\]
- 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).

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

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

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

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

---

### Figure 6.8: Underlying Signaling Transport Implementation

- **Behavior of SignalingStack:**
  - SignalingStack behaves properly.
    - \[ \text{noFailure} \rightarrow \text{noFailure} \]
  - 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.
    - \[ \text{noFailure} \rightarrow \text{late} \]
  - 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.
    - \[ \text{noFailure} \rightarrow \text{valueCoarse} \]

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

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

- 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.

\[ \text{omission} \rightarrow \text{commission} \]
• **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**

---

### 6.2.3 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:** this sub-system, which corresponds to the AAL2 sender end point (shown in Figure 6.1), is responsible of the 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 input port of **ServedUser_Impl** is connected to the input port of **OriginatingServiceEnd**. Output port of **ServedUser_Impl** component is connected to the input ports of **SignalingStack_Mux_Impl** component, output port of **SignalingStack_Mux_Impl** component is attached with the input port of **SignalingTransportConverter_Impl**.

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 behavior (see the FTPC 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, OriginatingServiceEnd sub-system produced \{noFailure, late\} failures on the output port.

This result is motivated as follows: ServedUser_Impl gets noFailure in input 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 failures \{noFailure, omission, late, valueCoarse\} so, instead of looking for a rule with a matching left-hand-side, we look for all the rules that match the elements of the set (i.e. we check if there are rules that match noFailure, omission, etc.) \[7\].
A noFailure behavior matches with the left-hand-side of the first rule of
SignalingStack_Mux_Impl, therefore it transforms noFailure into {valueCoarse,
noFailure}.

An omission failure matches with the left-hand-side of the second rule; therefore
SignalingStack_Mux_Impl transforms omission into {omission, commission}.

A late failure matches with the third rule and therefore SignalingStack_Mux_Impl
propagates late.

A valueCoarse failure is discarded because; it is not matched with the left-hand-side of
any transformation rules in SignalingStack_Mux_Impl.

As a result of the combination, SignalingStack_Mux_Impl produces the set of failures
{commission, late, valueCoarse, noFailure, omission} on the output port. It should be noted that
the user interested in observing the reaction of the system to specific failure propagation, instead
of considering the rule:

\[
\text{noFailure} \rightarrow \{\text{omission, valueCoarse, late, noFailure}\}
\]

In above case, user could perform the analysis 4 times by considering single failure behavior each
time.

SignalingTransportConverter_Impl receives the set of failures coming from
SignalingStack_Mux_Impl and it reacts in the following way:

- A noFailure matches with the left-hand-side of first rule of
  SignalingTransportConverter_Impl and according to that rule noFailure is transformed
  into {noFailure, late}.
- A late failure matches with the left-hand-side of its second rule and according to that rule
  late is propagated.
- A commission failure matches with the left-hand-side of its third rule and according to
  that rule commission is transformed into noFailure.
- An omission matches to left-hand-side SignalingTransportConverter_Impl and transforms
  omission into late.
- A valueCoarse failure is discarded because; it is not matched with the left-hand-side of
  any transformation rules in SignalingTransportConverter_Impl.

As a result, \{late, noFailure\} is received 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: this sub-system, which corresponds to the AAL2 switch (as shown in
Figure 6.1), 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 the input port of
SignalingTransportConverter_Impl component and the output port of
SignalingTransportConverter_Impl is connected with the input port of SignalingStack_Impl.
Similarly, output port of other SignalingTransportConverter_Impl is connected with the output
port of SignalingSwitch sub-system.

Similarly to what was done with the previous sub-system, to evaluate the failure behavior of this
composite component, we calculate its behavior in response to a set of failures. More specifically
we use the set: \{noFailure, late\}. The model of the composite component before the analysis is illustrated in Figure 6.12.

SignalingSwitch receives the input failure \{noFailure, late\} at SSH_R1 input port and provides these failures to its inner component SignalingTransportConverter_Impl as shown in Figure 6.12.

**Figure 6.12: SignalingSwitch sub-system before analysis**

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 \{late, noFailure\} as it is on output port. The explanation of the result is similar to the one which was given for the OriginatingServiceEnd sub-system.

**Figure 6.13: SignalingSwitch sub-system after analysis**
**DestinationServiceEnd Sub-System/Composite Component**: this sub-system, which corresponds to the AAL2 receiver end point (shown in Figure 6.1), contains three components `ServedUser_Impl`, `SignalingTransportConverter_Impl` and `SignalingStack_DeMux_Impl` (used for managing de-multiplexing) as shown in Figure 6.14.

To perform the analysis, DestinationServiceEnd sub-system, as Figure 6.14 shows, is fed with a set `{noFailure, late}` behavior.

![Figure 6.14: DestinationServiceEnd sub-system before analysis](image)

![Figure 6.15: Analysis result on DestinationServiceEnd sub-system](image)
The analysis result of DestinationServiceEnd sub-system is shown in Figure 6.15. DestinationServiceEnd sub-system receives \{noFailure, late\} on its input port and produces \{noFailure, valueCoarse, omission, late\} on the output port. The explanation of the result is similar to the one which was given for the OriginatingServiceEnd sub-system.

### 6.2.4 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 identified behavior of all sub-systems. Failure behavior of sub-systems can be identified by running FPTC analysis on sub-system level. Behaviors of all sub-system are as follow:

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

As seen previously, by providing noFailure in input port at the OriginatingServiceEnd sub-system, we obtain \{noFailure, late\} on the output. To retrieve further rules, we feed the OriginatingServiceEnd sub-system with valueCoarse failure at its input port and it propagates valueCoarse on the output port. The analysis result is motivated as follows:

- *ServedUser_Impl* gets valueCoarse in input port and propagates valueCoarse failure on its output port.
- *SignalingStack_Mux_Impl* gets a valueCoarse failure but, it is not matched with the left-hand-side of any transformation rules and propagates as it is on the output port.
- *SignalingTransportConverter_Impl* receives a valueCoarse failure but, it is not matched with the left-hand-side of any transformation rules and propagates as it is on the output port.

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

<table>
<thead>
<tr>
<th>Behavior of <strong>OriginatingServiceEnd</strong> 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

To retrieve further rules, we feed the subsystem with different failure information. The explanation of the result is similar to the one which was given for the OriginatingServiceEnd sub-system. After having performed the analysis we obtain the behavior shown in Table 6.2.

<table>
<thead>
<tr>
<th>Behavior of <strong>SignalingSwitch</strong> 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 6.3: Behavior of DSE Sub-System**

<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>

After getting the behavior of all three sub-systems, we create the corresponding component implementation diagrams and at implementation level we define the rules of these components. We create the composite structure diagram of AAL2 signaling system as shown in Figure 6.16. In AAL2 system/composite component, the output port of OriginatingServiceEnd is attached with input port of UnderlyingSignalingTransport. Similarly output port of UnderlyingSignalingTransport is attached with the input port of SignalingSwitch. We reuse UnderlyingSignalingTransport, so, its input port is attached with the output port of SignalingSwitch and so on.

The AAL2 system is fed in input with a normal behavior (noFailure) on its input port (AAL2_R1). Then the analysis is invoked.

![Figure 6.16: AAL2_System before analysis](image-url)
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\} to \textit{SignalingSwitch} and \textit{SignalingSwitch} transforms them into \{noFailure, late\}.

But \textit{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.

![Figure 6.17: FPTC Analysis Result on AAL2_System](image)

### 6.2.5 Discussion

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 analyze 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 \{noFailure, late\} 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 [37].
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, system’s dependability threats and failures analysis techniques.

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, to avoid misconception, only term failure is considered. 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.

A survey is conducted 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 do not allow 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, ambiguities are discovered in FPTC syntax and rules presented by different authors. Therefore, modified syntax of FPTC is presented after removing the ambiguities.

In the context of CHESS project, this thesis aims to investigate a tool-supported framework to engineer high integrity real-time component-based dependable embedded systems to evaluate the system’s dependability.

According to MDE principles, analysis models are automatically obtained from high level model of system’s architecture. This approach prevents the user for providing details at analysis level of the system. FPTC analysis results are back propagated at initial model(s). Model transformations are the means by which models can be manipulated. M2T transformation technique is used for getting 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 be compliant with CHESS tool-set.

The CHESS-FPTC plug-in is provided to evaluate the dependability in real-time component-based embedded systems. To illustrate and evaluate the CHESS-FPTC framework, a case study on AAL2 signaling protocol is provided. By using CHESS-FPTC plug-in user can identify failure behavior of the entire system as well as behavior of its building components. In case the system’s output is a failure, user should evaluate failure’s criticality and establish if mitigation is needed. In case of a critical failure, user should identify source(s) and introduce adequate mitigation means (i.e. additional components implementing fault tolerance strategies). In case system’s output is normal, the system can be considered robust with respect to the assumed failure in input. Possible extension in CHESS-FPTC plug-in 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 plug-in. 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 FI\textsuperscript{4}FA

A useful and small extension of the CHESS-FPTC plug-in could be to support the FI\textsuperscript{4}FA analysis technique [10]. This extension would be possible by considering additional failure types. FI\textsuperscript{4}FA focuses on failures avoidable through transaction-based mitigations.
REFERENCES


17. Nikolaos Spanoudakis, Pavlos Moraitis, “Model-Driven Agents Development with ASEME”, Technical University of Crete,
