LOUGHBOROUGH UNIVERSITY

Formal Transformation Methods for Automated Fault Tree Generation from UML Diagrams

A Doctoral Thesis
Submitted in partial fulfilment of the requirements for the award of
Doctor of Philosophy of Loughborough University

March 2019

by
Rosmira Roslan

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Professor Charles E. Dickerson and Doctor Siyuan Ji, my research supervisors, for their patient guidance, enthusiastic encouragement, valuable and constructive suggestions during the planning and development of this research work. I came to know about so many new things and I am really thankful to them. A special note for Charles Jackson, a manager at BAE Systems, for his help in proofread some part of this thesis. His willingness to give his time so generously has been much appreciated.

I would also like to thank Professor Ron Summers, my internal examiner for his advice through the viva voce in the first two years before he resigned from the Loughborough University. A special acknowledgement goes out to Doctor David Mulvaney, my new internal examiner, for his advice for my progression into the final year.

Besides my academic supports, I must express my very profound gratitude to Haji Roslan Mohd Noor and Hajjah Nor Hayati Ramlan, my parents, and family for their support and encouragement from the beginning and throughout my study. A special thanks to Mohd Fadhil Arshad, my husband, for his support and understanding. His presence in my final year gave me positive pressure for finishing this thesis. Words cannot express how grateful I am for all their continuous prayers and thought they have made on my behalf.

My journey could not have started without the financial support from Majlis Amanah Rakyat, my sponsor, and throughout the four years for the tuition fees and living allowance to survive alone with the new environment far from home.

Finally, I would also like to expand my gratitude to all those who have directly and indirectly guided throughout this journey, especially my colleagues have made valuable comment suggestions on my work which gave me an inspiration to improve the quality of the work.

PUBLISHED WORKS

Two publications have been published throughout the completion of this doctoral thesis.

1. Conference paper: Formal methods for a system of systems analysis framework applied to traffic management.

   Dickerson, C.E., Ji, S. and Roslan, R., 2016, June. In 11th System of Systems Engineering Conference (SoSE), pp. 1-6.

2. Journal paper: A formal transformation method for automated fault tree generation from a UML Activity model.

   Dickerson, C.E., Roslan, R. and Ji, S., 2018. IEEE Transactions on Reliability, 67(3), pp.1219-1236.

GLOSSARY OF TERMS

AM-FPC-FT Activity model-Fault Propagation Chain-Fault Tree: An overarching metamodel presented in Subchapter 4.2.7.

ARP Aerospace Recommended Practice: An international guideline for aircraft development by the Society of Automotive Engineers.

Attribute An observable characteristic or property in the context of system or system element (International Council on Systems Engineering 2015).

Behaviour The manner in which a system acts under specified conditions (Umeda et al. 1990).

CFT Component Fault Tree: A metamodel developed by (Adler et al. 2011) in which its development is based on the hierarchical decomposition of a system.

CM-FT Class model-Fault Tree: A metamodel presented in Subchapter 4.3.8.

CMF Conjunctive Material Form: Adopted from Conjunctive Normal Form for conjunctions of material implications.

COMPASS Comprehensive Modelling for Advanced Systems of Systems: A group of researchers and companies to collaborative research on model-based-techniques for developing and maintaining systems of systems from October 2011 to September 2014.

CoO Concept of operation: Characteristics of a proposed system that describe the capabilities of the system to achieve the desired purpose.

CS Constituent system: A system of system of systems that contributes to the performance of system of systems. It is also known as system element or subsystem.

CPS Cyber physical system: System that integrated with computation and physical system.

Error

1. An occurrence arising as a result of an incorrect action or decision by personnel operating or maintaining a system (International Society of Automotive Engineers 1996).

2. A mistake in specification, design, or implementation (International Society of Automotive Engineers 1996).

3. An occurrence that triggering fault and failure of a system.

Event

1. An occurrence which has its origin distinct from a system, i.e. external event (International Society of Automotive Engineers 1996).

2. An occurrence of an action that progress from one state to another state of each process in a system (Waldecker & Garg 1991).

Facility A terminology used in this thesis to describe physical object (presented by Class), subsystem and system (presented by Class), and system component.

Failure	A loss of function or a malfunction of a system or a part thereof (International Society of Automotive Engineers 1996).
Fault	An undesired anomaly in a system (International Society of Automotive Engineers 1996).
Formal Methods	A specification written in a formal notation, often for use in proof of correctness
FPC	Fault Propagation Chain: Graphical representation of faults presented in Subchapter 4.2.
FTA	Fault Tree Analysis: A safety assessment analysis method.
FTM	Fault Tree Metamodel: A metamodel for Fault Tree presented in Subchapter 3.2.3.
fUML	Foundational UML Subset: An executable subset of standard Unified Modeling Language.
Function	The special kind of proper activity or the mode of action by which is fulfils its purpose (Umeda et al. 1990).
Graphical Language	Information or knowledge expresses in graphical type rather than textual, i.e. Unified Modeling Language and Systems Modeling Language.

INCOSE International Council on Systems Engineering: An organisation that focuses to develop and disseminate systems engineering principles and practices.

MBSE Model-based systems engineering: A methodology that uses models in the processes of SE such as to define and design rather than documented-based.

Modelling Language Information or knowledge expresses in a structure that is defined by a consistent set of rules, e.g. graphical language and textual language.

Modelling Technique Method of expressing information or knowledge, e.g. Unified Modeling Language, Systems Modeling Language, simulation, and flowchart.

OMG Object Management Group: An organisation that drives the development of technology standards for industry such as Unified Modeling Language and Systems Modeling Language.

QVT Query/View/Transformation: A model transformation language defined by Object Management Group.

RAMS Reliability, Availability, Maintainability, Safety: A principle that use as to measure in the design, implementation, and maintenance phase.

RMS	Ramp Meter System: A constituent system of Traffic Management System of Systems that has been used as a case study in this thesis (c.f. Chapter 3, Chapter 4, and Chapter 5).
Safety Assessment	Activity taken to produce evidence of safety.
Safety Assessment Analysis Method	1. Method that used in the safety assessment, e.g. Fault Tree Analysis and Failure Mode and Effects Analysis. 2. Hazard analysis technique (Leveson 2012).
Safety Critical System	System whose failure may harm the user, environment, and the system itself.
SE	Systems Engineering: An interdisciplinary field and approach for covering all aspects in engineering such as design, tools, resources, documents, installation, processes, and communication to enable the realisation of a system.
SoS	System of systems: A system whose constituent systems (subsystems) are managerially and/ or operationally independent systems (International Council on Systems Engineering 2015).
SoSE	System of Systems Engineering: It is emerging as an attempt to address integrating complex metasystems (Keating et al. 2003).

SSE System Safety Engineering: It is a compilation of engineering analyses and management practices that control dangerous situation (Bahr 2014).

SysML System Modelling Language: A modelling language for model-based systems engineering to visualise a system customised from Unified Modeling Language by Object Management Group.

System A combination of inter-related elements to perform a specific function(s).

System Element Terminology used in (International Council on Systems Engineering 2015) as to present subsystem or constituent system.

TCC Traffic Control Centre: A constituent system of Traffic Management System of Systems that has been used as a case study in this thesis (c.f. Chapter 3, Chapter 4, and Chapter 5).

TMSoS Traffic Management System of Systems: A case study that has been used in this thesis (c.f. Chapter 3, Chapter 4, and chapter 5).

UML Unified Modelling Language: A modelling language for model-based systems engineering to visualise a system adopted by Object Management Group.

UML Diagram The standard of diagrams used for representing system adopted by Object Management Group.

UML System Model	The terminology is defined in this thesis for representing system models that modelled and represented in Unified Modeling Language.

ABSTRACT

With a growing complexity in safety critical systems, engaging Systems Engineering with System Safety Engineering as early as possible in the system life cycle becomes ever more important to ensure system safety during system development. Assessing the safety and reliability of system architectural design at the early stage of the system life cycle can bring value to system design by identifying safety issues earlier and maintaining safety traceability throughout the design phase. However, this is not a trivial task and can require upfront investment. Automated transformation from system architecture models to system safety and reliability models offers a potential solution. However, existing methods lack of formal basis. This can potentially lead to unreliable results. Without a formal basis, Fault Tree Analysis of a system, for example, even if performed concurrently with system design may not ensure all safety critical aspects of the design.

Therefore, motivated by the above challenge, this thesis develops transformation methods that promises automated Fault Tree generation from system models that are presented in the Unified Modelling Language (UML). These methods are staged into three parts: (i) a transformation method that generates a functional Fault Tree from UML Activities to capture the behavioural failure aspect of a system, (ii) a transformation method that generates a component Fault Tree from UML Classes to capture the structural failure aspect of a system, and (iii) a transformation method that integrates the previous two methods to enable generation of hierarchical fault events in a Fault Tree that adhere to system functional allocation and hierarchical decomposition. The key features of the transformation methods include: (i) the use of propositional calculus and probability theory to establish a mathematical foundation that ensures the generated Fault Trees are semantically equivalent to their source models, (ii) the development of the Fault Propagation Chain which serves as a bridge to connect the system architectural viewpoint and system failure viewpoint, and (iii) a set of overarching metamodels that unifies relevant UML metamodels and Fault Tree metamodel to facilitate tool development.

Unlike existing transformation methods, this research exploits the relational structure embedded in the system models in addition to system model elements. As demonstrated through the application of the transformation methods to a Traffic Management System, it is observed that derived Fault Trees preserve the relational structure of the system model. This key finding therefore provides a means to assess the safety and reliability of system architecture at the early stage of a system life cycle. To verify the methods and the finding, these transformation methods are also applied to a Railway System case study in which the system architecture is evaluated through Fault Trees generated from the available architecture models.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Current Situation and Challenges of System Development

The Systems Engineering (SE) life cycle is divided into stages that cover end-to-end processes of system development. Two examples of established standards are ISO/IEC/IEEE 15288:2015 (ISO/IEC 2015) and United States (US) Department of Defense (DoD) Systems Engineering. These standards have different viewpoints on the SE life cycle structure for ensuring the system developed meets the required functionality (International Council on Systems Engineering 2015). Additionally, Waterfall (Royce 1970), Spiral (Boehm 1988), and Vee (Forsberg & Mooz 1991), are the three most well-known models that illustrate SE life cycle. The first two models, Waterfall and Spiral, were developed specifically to address software development processes only of large-scale systems (Estefan 2007). The Vee was developed to address the involvement of SE in a project life that is wider than just software development including cross-functional between engineering disciplines and non-engineering disciplines. In this thesis, four generic stages of SE life cycle that consist of specification, design, implementation, and testing are structured by using Vee model.

System Safety Engineering (SSE), another engineering domain, is embedded into SE for analysing the system under development to ensure it is safe for the end users and environment. Similar to SE, SSE has end-to-end processes that can be grouped into three stages; system safety requirement, system safety assessment, and verification and validation. In IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems standard, 16 phases of system safety life cycle are stated (International Electrotechnical Commission 2010). The phases are structured into three large groups: i) Analysis; ii) Realisation; and iii) Operation (Redmill 1999). Whilst US DoD System Safety standard, MIL-STD-882, has 25 safety tasks that are organised into four groups: i) Management; ii) Analysis; iii) Evaluation; and iv) Verification (Department of Defense United States of America 2012). The major overlap of Analysis is principally in system requirement analysis.

SE and SSE are interdependent in development of any system as depicted in Figure 1.1 with light purple and bright grey respectively (read from left to right). The stages of SE and SSE life cycles run on the same project timeline. In reality, the life cycle is not linear as it shows. In order to ensure effective and efficient communication that accounts for ongoing learning and decisions, iteration and recursion are applied to the life cycle processes with appropriate feedback loops (International Council on Systems Engineering 2015). Nevertheless, their processes are performed separately as SE and SSE establish different views and different levels of details (Mhenni et al. 2018). During the design phase, the detailed design of a system under development should undergo safety analysis before it is locked for the Implementation Stage. This is to ensure that the designed system has satisfactory safety levels (Mhenni et al. 2018). However, the detailed design of a system can be analysed for reliability characteristics as soon as they are available (Pai & Dugan 2002). This may occur late in the design phase (Mhenni et al. 2014).

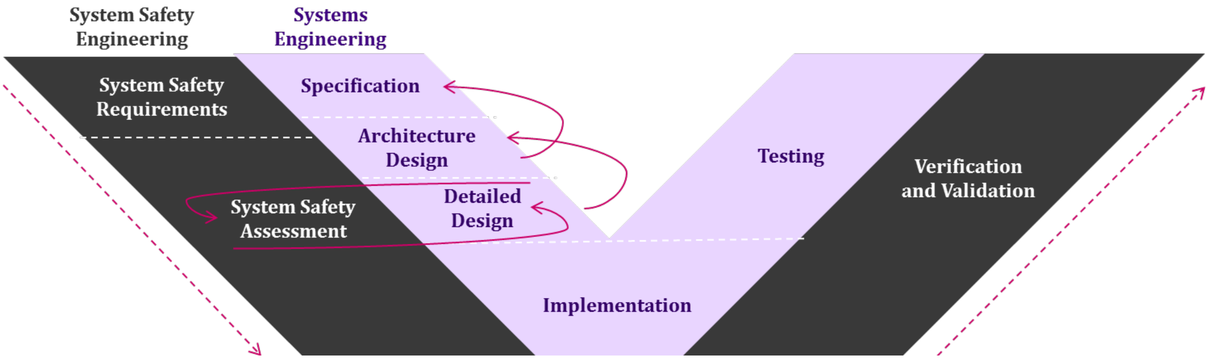


Figure 1.1 General Processes of System Development

System Safety Assessment (SSA) is a stage in the SSE life cycle that is associated with the detailed design of a system. During the Detailed Design Stage, a comprehensive evaluation is undertaken to show safety requirements of the developing system are met.

A range of SSA analysis methods are recommended to be used within the safety evaluation process. The system design will be verified at multiple levels including decomposition of the system qualitatively and quantitatively in order to meet the system safety requirements. Fault Tree Analysis (FTA), Dependence Diagram (DD), and Markov Analysis (MA) are three model-based methods that are recommended in analysing safety aspects of system design (Dai et al. 2018). These methods facilitate subdivision of system level events into lower level events for ease of analysis. The events are failure events of the system which are identified either being individually or collectively leading to the occurrence of undesired system failure. Amongst the three methods, FTA is the most popular one. It supports formal analysis as it uses Boolean logic gates in the tree. Compared to the other two, FTA can evaluate the specific situation of the failure events to the occurrence of the undesired system failure.

More recently, from the 1990s to the 2000s, researchers and practitioners have been automating Fault Tree generation to reduce the time required by producing Fault Trees manually and to avoid human errors in manual Fault Tree constructions (Majdara & Wakabayashi 2009), (Bhagavatula et al. 2016). The key challenge in the automation process, as identified by researchers in the field, is actually not the automation itself. Rather the challenge is to do with how the system should be modelled (Majdara & Wakabayashi 2009). For instance, creating a comprehensive model for a complex system while maintaining sufficient level of detail can be challenging. In addition, a good modelling technique in one domain may not be general enough to have a wide applicability to cover other engineering domains. To tackle this challenge, model based approaches such as the use of diagraphs (Wang et al. 2003), (Vemuri 1999), state diagrams (Rauzy 2002), (Liggesmeyer & Rothfelder 1998), component-based modelling (Majdara & Wakabayashi 2009), (Bhagavatula et al. 2016), and knowledge-based approaches as reviewed in (G 2002) are included in modelling techniques developed for automated Fault Tree generation in the past. Despite the various level of success in the adoption of these methods in industrial settings, model availability has been one of the key issues. For instance, a substantial effort may be required to create the sophisticated system models, which is in contradiction to this idea of reducing time required in producing Fault Trees.

Much more recently, model-based SE (MBSE) a rapidly growing field originated from defence and aerospace, has attracted attention from the reliability and safety community (Zeller et al. 2016), (Thoma et al. 2012), (Lazǎr et al. 2010). In MBSE, modelling techniques and languages have been developed to model complex systems and system of systems (SoS). Many of modelling languages in current practice do not have well-founded semantics to support MBSE (Fitzgerald et al. 2013) and formal methods for MBSE are needed. Of the various modelling languages, Unified Modelling Language (UML) (Jacobson et al. 1999), which was originally developed for software engineering, became popular in SE due to its general applicability and extendibility when it was first introduced by Object Management Group (OMG) in 1997 (Liebel et al. 2018). In UML Specification 2.5.1, the latest version of UML, as presented in Figure 1.2, 14 diagrams of two major kinds of diagram types are established to support conceptual design of complex system (Object Management Group 2017a). UML diagrams are divided into two major categories; behaviour diagram and structure diagram (Object Management Group 2017d). Respectively, the diagrams classification is regarded as 'how' is the performance of the system and 'what' is the appearance of the system (Holt 2001). The functions and behaviours of a system that describe a series of changes over time is showed by behaviour diagrams, i.e. performance of the system. Element specifications that are irrespective of time and represent the static structure of objects in a system are showed by structure diagrams, i.e. appearance of the system. UML has been elaborated into domain-specific languages such as the System Modelling Language (SysML) for general purpose system modelling (Object Management Group 2007) and the UML profile for modelling and analysis of real-time embedded systems for embedded systems (Selic & Gerard 2014). As compared to UML, a smaller number of visual modelling language is offered by SysML with only eight diagram types (Amissah et al. 2018). The software specific components of UML elements that are unnecessary in SE design are explicitly omitted. The Activity Diagram construct is also modified to support activity extensions and continuous functions, i.e. allocation table, and action and object in a diagram. Two structure diagrams from UML, Class Diagram and Composite Structure Diagram, are replaced with Block Definition Diagram and Internal Block Diagram respectively. Technically, the selection of diagram for modelling a system is dependent to the perception of the user, i.e. designer.

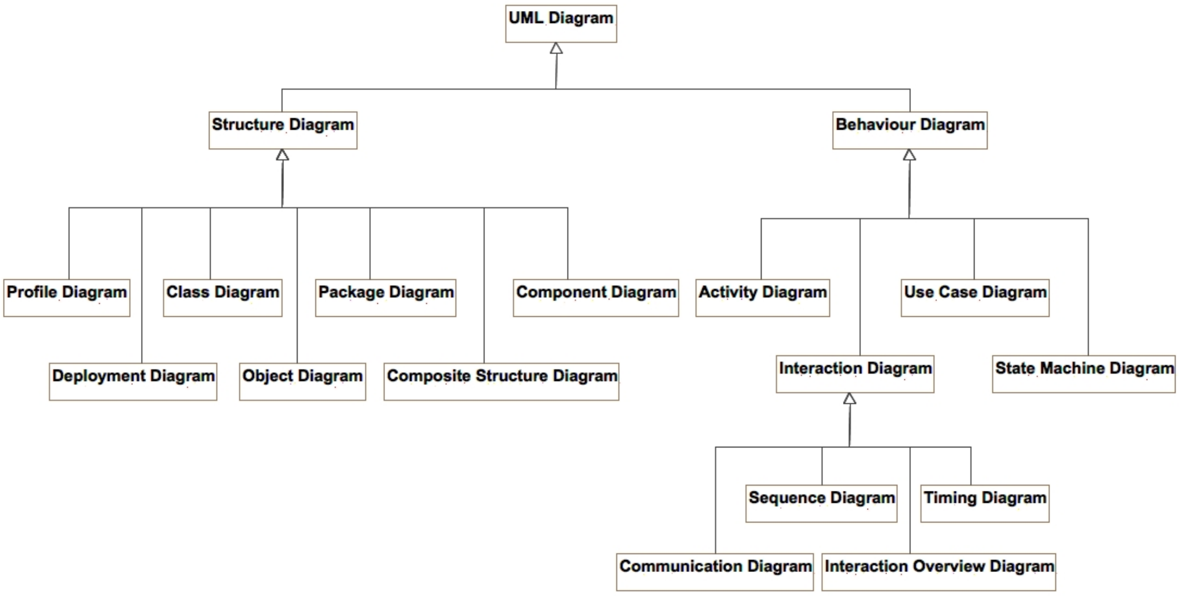


Figure 1.2: Unified Modelling Language Taxonomy (Object Management Group 2017d)

Unlike the previous specific modelling techniques developed for automatic Fault Tree generation, the modelling of systems across various domains can be facilitated by UML and its extensions. Nowadays, domains have started adopting MBSE approaches using UML; and its extensions are becoming widely adopted. Moreover, researchers and practitioners have also started using UML and its extensions for modelling and analysis from a system safety and reliability perspective (Thoma et al. 2012), (Lazǎr et al. 2010), (Holt 2001), (Friedenthal et al. 2014), (Selic & Gerard 2014), (Zoughbi et al. 2011). As such, for the purpose of automated Fault Tree generation, model availability is becoming less of an issue. However, efficient and reliable transformation techniques between models created using these languages and Fault Trees remain a challenge.

Thanks to the development of transformation languages, such as ATLAS Transformation Language developed by French Institute for Research in Computer Science and Automation (Jouault et al. 2008) conforming to the Object Management Group (OMG) standards, Query/View/Transformation (Guduric et al. 2009), transformations between models developed in different languages and in different domains can be more easily achieved technically.

Researchers and practitioners have developed various ways for transforming UML models (Kim et al. 2012), (Hu et al. 2011), (Zhao & Petriu 2015), (Kim et al. 2010) and SysML models to Fault Trees (Mhenni et al. 2018), (Xiang & Yanoo 2010), (Yakymets et al. 2013). These transformations share the following commonalities: (i) they define entity-to-entity types of mapping. For instance, a Use Case in UML is mapped onto an intermediate event in a Fault Tree (Zhao & Petriu 2015), (ii) the transformations developed lack formality; hence they lack provable rationales to support the defined mapping. Without a formal basis, Fault Tree Analysis of a system, for example, even if performed concurrently with system design may not ensure all safety critical aspects of the design. Further precision can be brought to modelling through semantic transformation between models.

Given that Fault Trees are meant to be used to assess system reliability and safety, trustworthiness of an automatically generated Fault Tree from system models can become questionable if the transformation does not have a formal basis. To bring formality into the development of system models and Fault Trees, attempts in formalising UML models (Lazӑr et al. 2010), (Craciun et al. 2013) and Fault Tree models (Xiang & Yanoo 2010), (Xiang et al. 2004) have been made independently. Furthermore, to enhance consistencies between models developed in different domains, major projects such as Comprehensive Modelling for Advanced Systems of Systems (COMPASS) (Ingram 2014) have developed tools and methods for engineers from all range of disciplines to provide support in cross-domain collaborations (Andrews, Ingram, et al. 2014), (Bryans et al. 2014). Nonetheless, a formalised transformation method between UML models and Fault Tree models is still missing.

1.2 Aim and Objectives of the Research

The aim of this thesis is to develop mathematically meaningful transformation methods for generating Fault Trees automatically from system models that are modelled in UML. The generated Fault Trees can then be used for analysing system safety and reliability at the system architecture design stage. As briefed in Subchapter 1.1, UML system models consist of 14 types. For this thesis, of the 14 types, Activity and Class are selected as representing behaviour and structure of a system respectively. By using the formal transformation methods, the undesired event of the system derived from the system models are presented in the generated Fault Tree that can be used to assess system architecture design that was created in UML.

To achieve the aim, three major ideas are to be explored: (i) definition of models and mathematical representation of the models, (ii) development of transformation methods, and (iii) verification of the transformation methods, which the details are further elaborated as follows:

1. To develop metamodels – As the system models in this thesis are concerned with UML Activity and Class, two holistic metamodels of Activity and Class are developed based on UML Specification by the OMG. A metamodel of Fault Tree is also developed based on ARP 4761. The development of these metamodels concerns the elements that graphically presented in the respective models.

2. To define a mathematical representation of system models – The nodes in the respective system models are defined in a mathematical basis. In particular, propositional calculus is used to define Actions (function) and Facilities (component). Hence, a precise semantic of the system models can be offered. The approach of using mathematical basis to represent the models also assists later process such as verifying the models in a systematic way. The application of mathematic is also used to define the fault viewpoint of the system models through the actions and facilities.

3. To develop formal transformation methods from single perspectives of a system to Fault Trees – Based on the defined propositions and fault, logical models of system behaviour (Activities) and system structure (Classes) together with their fault viewpoints are identified for developing semantic mapping rules. In addition, intermediate steps are designed to support the transformation methods: (i) Fault

Propagation Chain is introduced to represent fault viewpoint of Activities, and (ii) Complementary Class is introduced for a complete structure of Classes.

4. To develop transformation method from the integration of two system models to Fault Tree – The concepts of facilitation and ownership are introduced to integrate system behaviour and system structure from the fault viewpoint for the Fault Tree transformation.

5. To develop overarching metamodels – Overarching metamodels are developed for abstracting the domain-specific metamodels. The development of the overarching metamodels is concerned with metamodels that developed earlier and transformation methods including the introduced steps. The overarching metamodels is used to assist automated Fault Tree generation from system models that modelled in UML.

6. To apply the developed methods – The transformation methods are applied to case studies to demonstrate automated Fault Tree generation from system models that created in UML. Furthermore, the generated Fault Tree can be used to analyse the system models.

1.3 Scope of the Research

The coverage of this thesis is limited to general concepts and generic description of processes, tools, and techniques. Formal testing of system safety, verification, and validation from the viewpoint of regulators are not being considered in this thesis. The demonstration of the methods developed in this thesis is via modelling and analysis without the use of physical testing.

To normalise the use of languages from different domains in this thesis, definitions of terminologies used in the following two standards follows are adopted and listed as in the Glossary of Terms.

1. **UML Specification 2.5.1** - The information for defining system models in Unified Modelling Language (UML) is provided by an international technology standards consortium, Object Management Group (OMG) (Object Management Group 2017d).

2. **ARP 4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment** – There are several international safety standards that provide guidelines on safety analysis and assessment with the use of Fault Tree Analysis (FTA) across different domains of application. To name a few, there is ARP 4761 (International Society of Automotive Engineers 1996) which is a guideline for conducting safety assessment process in aircraft domain, DO 178C (Radio Technical Commission for Aeronautics 2011) which provides recommendations for the production of software for airborne system and equipment, and IEC 61025 (International Electrotechnical Commission 2006) which describes FTA and provides guidance on its application. The intention of this research is not to normalise these standards for one unified Fault Tree methodology and metamodel. Rather, one particular standard, which ARP 4761 will be followed in this thesis due to the following reasons. Firstly, it is an authoritative standard that provide extensive knowledge on the use of Fault Tree in practice. It is not only used by aerospace organisations, but also by practitioners and academia in other domains (Joshi et al. 2005). Secondly, there is an alignment between ARP 4761 and ARP 4754, a guideline for development of aircraft from systems requirements through systems verification. The alignment allows safety assessment process in the systems engineering (SE) methodology through model-based approach. This offers a good idea of model-based systems engineering (MBSE) and model-based safety analysis (MBSA) practices in the industry. Lastly, it has relationship to other international standards which constitute materials of safety analysis such as ARP 4754 and DO 178 (Xiaoxun et al. 2011). The adoption of ARP 4761 in this thesis provides harmonisation and consistency with other international standards. In this thesis, the ARP 4761 is used as to refer the fundamental of Fault Tree.

The development and demonstration of Fault Tree in this thesis are supported by published works. There are different ways of Fault Tree development applied by practitioners and academia that can be referred to develop formal transformation methods for generating Fault Trees in this thesis.

1.4 Structure of Thesis

This thesis is structured in seven chapters as follows:

Chapter 1: Introduction

Brief overview of the current situations and the challenges of system development in industry are discussed. SE and SSE life cycles for the system development are provided. Methods and tools that are commonly used in the system development are also included.

Chapter 2: Literature Review

In this chapter, detailed information on issues discussed in Chapter 1 is presented. Review of literatures for the research area are also included. In addition, the standard processes used in SE and SSE are presented. An alternative way for integrating SE and SSE to close the gap between processes and implementations of formal methods in the integration part are reviewed and discussed.

Chapter 3: Foundational Research Knowledge

The formal transformation methods of this thesis that concern model-based approach and formal methods are discussed. An authorised case study for demonstrating developed methods is introduced. In this chapter, foundational knowledge of technical segment in Chapter 4 and 5 is presented.

Chapter 4: A Formal Transformation Method for Automated Fault Tree Generation from Single UML System Model

Development of formal transformation methods for automated Fault Tree generation from system models that modelled in UML are presented. The formal transformation methods are developed separately to the correspond of the system models. In particular, Fault Trees are shown to be generated formally and automatically from system models modelled in UML Activity and UML Class. To assist formality of the methods, supporting methods are developed based on the foundational knowledge discussed in Chapter 3. Two overarching metamodels are developed of unifying each of the system models and Fault Tree together with the formal transformation method to describe the formal transformation methods between models in the abstract level.

Chapter 5: Automated Fault Tree Generation from Integrated UML System Models

The formal transformation methods developed in Chapter 4 are expanded for integrating system behaviour and system structure failures in a single Fault Tree. The concepts of allocation is reviewed which then introducing the concepts of facilitation and ownership as to connect system behaviour and system structure. An overarching metamodel is developed for abstracting the transformation from the integrated system models to Fault Tree.

Chapter 6: Verification of the Formal Transformation Methods

The proposed formal transformation methods in Chapter 4 and Chapter 5 are applied to an authorised case study in appropriate order. The derivation of system failure states from a holistic viewpoint rather identify isolated failure states based on ad hoc reasoning is discovered. Comparative analysis is done to compare result of the proposed methods with related researches.

Chapter 7: Conclusion and Future Work

Finally, the development of the formal transformation methods is concluded and limitations of proposed methods are discussed. At the end of this chapter, research direction for future work is also proposed.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

The transformation methods from a system architecture modelled in UML to a safety analysis model in this thesis are supported by literature on model-based approaches, system safety assessment methods, and formal methods of system architecture design. Since the formal transformation methods involve transformation of models across engineering disciplines, the thesis presents the implementation of model-based approach in SE and SSE. Nevertheless, since the precise semantics in the transformation from model to model is a concern, formal methods are also applied to the transformation methods.

2.2 Key Concept of Systems

The review starts by setting out the general understanding of systems, based on the definitions of two independent international organisations and one collaboration between three other international organisations specialising in standards related to SE; namely, the International Council on Systems Engineering (INCOSE), the National Aeronautics and Space Administration (NASA), and the International Organisation for Standardisation/ International Electrotechnical Commission/ Institute of Electrical and Electronics Engineers (ISO/IEC/IEEE). According to these organisations, a system is a collection of two or more integrated elements, which can also can be atomics, that can be constructed in a vertical range of contexts such as hardware, software, human, and environment (International Council on Systems Engineering 2015), (National Aeronautics and Space Administration 2007), (ISO/IEC 2015). These elements are organised and interact with each other to achieve and serve stated purposes defined by the stakeholders' requirements. On a large scale system, a system may include a number of different elements, which are called subsystems, each with their own set of elements.

In such a scenario, where the operations of the subsystems can be recognised individually, the system as a whole is referred to as system of systems (SoS) (Boardman & Sauser 2006).

System of systems (SoS) is defined as a collection or decomposition of independent complex operational system(s) that interact among themselves to achieve a common goal in various applications including aerospace, military, space, and manufacturing (Jamshidi 2008). Although for more than a decade, the concept of SoS as a presentation of a higher-level viewpoint of systems interactions has remained at a development stage (Jamshidi 2008). Recently, SoS has started to be used widely even in earth observation, and this means that many fields of applications are emerging to address the common problems of integrating many independent, autonomous, and often large, systems that operate together in order to satisfy a global goal (Nativi et al. 2015). Also, the concept of SoS is accepted in the renewable and non-renewable energy areas which deal with the supply of environmental, water, land, and economic including trade-off between resources, i.e. SoS composed of independent but interacting systems dealing with factors such as water, land, the climate, and economy (Hadian & Madani 2015).

2.2.1 The Relationship between Systems and Systems Engineering

The fact that systems engineering (SE) has been adopted in various interdisciplinary fields means that there are a number of definitions of SE (Keating et al. 2003), (Blanchard & Fabrycky 2013), (International Council on Systems Engineering 2015). Generally, SE is an interdisciplinary engineering management, process, or approach that seek to transform descriptive needs into a successful and desirable system by means of a complete system life cycle (Keating et al. 2003). Here, the system life cycle encompasses requirement specification, design, verification, and the retirement of the system in a balanced set of people and process solutions including tools and concepts. SE should not be organised in a similar manner to fields in speciality engineering disciplines such as civil engineering and mechanical engineering, however, since a well-planned and highly disciplined approach must be followed (Blanchard & Fabrycky 2013). At the level of SoS, system of systems engineering (SoSE) has been introduced that departs from SE to integrate multiple complex systems that address specific problems or need.

Blanchard and Fabrycky have emphasised four areas of SE, namely: (i) A top-down approach which views a system as a whole from integrated subsystems to the components; (ii) A life cycle structure which address all phases of system development and problem solving; (iii) A definition of system requirements which marks the baseline of decision making for the system design process; and (iv) An interdisciplinary approach which addresses design objectives, methods, techniques, and tools to facilitate the implementation of the SE process. These four areas can be related to the four strengths of SE identified by Keating. The first strength is systems theory and principles for design, analysis, and execution of a system which address the phases of the system life cycle. Second, the interdisciplinary focus in system development and problem solving. In essence, since the system development is addressed through the system life cycle, the problem solving can also be considered on the basis of the life cycle of the system. The third strength of SE is the emphasis on a disciplined and structured process to achieve results. The definition of system requirements is used to define specific criteria for the design and can be used as a basis for the development of a successful system. The fourth strength is an iterative approach to developing systems to meet expectations for problem resolution. The life cycle orientation can be the benchmark throughout the development of a system which enables a traceable iterative process for the achievement of a desirable system.

2.3 Safety-critical Systems

Safety can be defined as the condition of being free from undergoing or causing hurt, injury, loss, or potential harm. Systems whose failure could result in loss of life, significant property damage, or damage to the environment such as medical devices, aircraft flight controls, weapons, and nuclear systems are viewed as safety-critical systems (Knight 2002). For example, an aircraft is a complex system and it is also a safety-critical system, consisting of more than ten integrated main systems (Moir & Seabridge

2008). These main systems consists of elements or subsystems, some of which are safety-critical system, e.g. unlike the radar system, the failure of flight control system could cause the aircraft to crash. Moreover, four years of search and rescue activities after Malaysia Airlines Flight 370 was reported missing on 8th of March 2014 (Ashton et al. 2015), has brought safety recommendations and regulation revision in designing safety-critical system into the spotlight.

Safety-critical systems and SoS can be found in a variety of commercial domains such as aerospace and commercial nuclear facilities, to name just two. Failures in these types of systems can result in injury or even death. The increasing use of embedded systems underscores the importance of understanding the behavioural aspect of systems and SoS. Safety is normally governed by international safety standards for systems and software. For example, in the aerospace industry, compliance with RTCA DO-178C standard is recognised as an acceptable means for verifying airworthiness and for the certification of the quality assurance of the airborne software systems. Nonetheless, such certification has been plagued by challenges such as miscommunication between engineers and certification authorities (Zoughbi et al. 2011). Thus, ensuring close and continuous monitoring during the development process is an important element in tackling these challenges.

2.3.1 Relationship between System Safety Engineering and Safety-critical System

SSE was first established in 1962 for the development of United States Air Force ballistic missiles systems. SSE, which is closely related to SE, seeks to design out the potential causes of accidents at an early stage in the design process and whilst systems are under development (i.e. SE life cycle). Where hazards cannot be designed out of systems SSE seeks to minimise the probability of safety-critical failures (Roland & Moriarty 1990). Safety-critical system failure occurs when there is a transition from correct service to incorrect service delivered by a system (Avižienis et al. 2004). Safety-critical systems such as in nuclear and aerospace applications are complex and need rigorous safety verification (Ruijters et al. 2017).

2.4 The Relationship between System and Safety

Systems and safety are observed as two important but independent domains in engineering. Safety in SE was introduced in the Institute of Aeronautical Sciences for more than a decade before it was formally applied in the early 1960s (Roland & Moriarty 1990). According to International Council on Systems Engineering (INCOSE), safety must be designed into developing system (International Council on Systems Engineering 2015). This means the faults and failures of a system must be understood during the system design. System safety has to ensure that faults and failures are prevented before they happen when the system operates. From the development of a system until its disposition, the condition of the system and environment must be assured as safe. The difficulty of safety analysis corresponds to the level of complexity of a system. Safety and reliability are part of system design. They can influence the subsequent design decisions requiring trade-offs and affecting system cost (Pai & Dugan 2002).

2.4.1 Technical Practices of System Design and Safety Analysis

According to Nancy Leveson, system design is an independent process from system development with safety analysis (Leveson 2012). Since SE and SSE are two different domains, the developments of system design and system safety are assumed to occur independently. Both (system design and system safety) have their separate activities in their own 'time zone'. Although they are treated as independent, their activities are interrelated at some point. System design is conducted with some regard for safety issues but, in most of the cases, a safety analysis is typically done after the detailed design has been produced (Leveson 2012). The system design is then handed over to the safety engineers who analyse the safety of the design. It will be handed back over to the system engineers with comments on the safety design and often requests for change (Fenelon et al. 1994). Though this practice, the potential causes of accidents, that is scenarios that can lead to losses can be eliminated or controlled in the design of overall system before damage occur. Safety assessment analysis has traditionally been performed manually by the safety engineers (Joshi & Heimdahl 2005). This manual practice has been upgraded to automatic with the support of a model-based approach to system design.

2.4.2 Safety Analysis at the End of System Design Stage

Traditionally, safety analysis techniques rely solely on skill and expertise and experience of the safety engineer. In contrast to the traditional method, and now that the model-based approaches have become more prominent, the safety-related causal relationships can be derived from a system model. If safety analysis is performed at the early stage of system development, safety engineers could only be provided with a broad system architecture design by system engineers. At this point, by using conventional Fault Tree Analysis (FTA), the high level details of a system are somehow not visible in the analysis. The details of the system remains unclear and hard to analyse, making it difficult for safety engineers to provide system engineers with feedback about safety consequences and potential improvements of the design (Olmo et al. 2018). In order for a system design engineers to provide sufficient details of the system under development to system safety engineers, it has become a practice for safety assessment analysis to be performed at the end of detailed design stage.

FTA is based on a detailed system, operation diagram, and therefore needs to be done after the system design stage. In ARP 4761, however, system functional analysis is also considered in generating an FTA and the development of safety analyses is recommended as early as the system design stage (International Society of Automotive Engineers 1996).

2.4.3 Challenges of the Practices

The design and safety departments in an organisation work as separate entities which are populated by engineers with different skills (Fenelon et al. 1994). Although it is inevitable that engineers complete their tasks according to the objectives of their departments, poor cooperation between system design and safety analysis processes does not easily accommodate the development of systems, especially large-scale and safety-critical system. This becomes worse when there is a limitation in safety assessment's applicability to today's system.

2.4.4 Proposed Ideas to Tackle the Challenges

Practitioners and researchers have proposed a number of approach to tackle the challenges in respect to the integration of system design and safety analysis within system development. Safety must be built into the design of a system, it cannot simply be an add on or measured afterwards, but must be considered in the context of the system as a whole and not just as part of the individual components of the system such as hardware and software (Leveson 2012).

Safety should be accommodated at an early stage of system development, when the design decisions are made. The implementation of safety into system design is one of keys to having a cost-effective safety effort (Leveson 2012). It is less expensive and far more effective to build safety early than try to tack it on later. This supports the 'make it right' concept during the design stage, including then thorough testing.

Fenelon et. al. propose a new way of organising and structuring system development and safety assessment processes (Fenelon et al. 1994). Their aim is to facilitate and support change management by stipulating that design and safety outputs must be agreed by both design and safety engineers during the early stages of system development to ensure that the needs for the subsequent stages are considered. This often entails appropriate trades and decisions to accommodate the needs of later stages in an affordable and effective manner (International Council on Systems Engineering 2015).

2.4.5 International Standards, Guidelines, and Recommendations

Standards, guidelines, and recommendations have been established as a benchmark in the SE life cycle. Most safety critical system in domains such as aerospace, railway, and automotive are subjected to international standards enforced by third parties (e.g. certification authority) as a way of ensuring safety and reliability (i.e. that they do not pose undue risks to people, property, and the environment) (De La Vara et al. 2016). According to the International Organisation for Standardisation (ISO), a leading standardiser, 22242 international standards have been published by the organisation to address ways of doing things (i.e. products) in various areas. Of these, ISO 15288 was introduced to provide a generic SE life cycle framework of processes and life cycle stages. Furthermore, in the automotive arena, ISO 26262 and EN 61508 (European standard) were established as a generic functional safety standards for electrical and electronic systems.

A general view of civil aircraft development given by Benjamin Gorry, a lead engineer for product safety at BAE Systems, as depicted in Figure 2.1, a collective of international standards and guidelines are involved in the development of an aircraft (Gorry 2015). Generally, SAE ARP 4754A is a specific guideline that was established for the development of civil aircraft and its systems. Furthermore, there are specific guidelines for fixed wing and rotary aircraft for specific components and elements in an aircraft system such as software, hardware, and integration of the modular avionics for original equipment manufacturer (OEM) to follow. The establishment of guidelines and standards is recommended and mandatory for the OEM to follow as a proof of the safety and quality of the product.

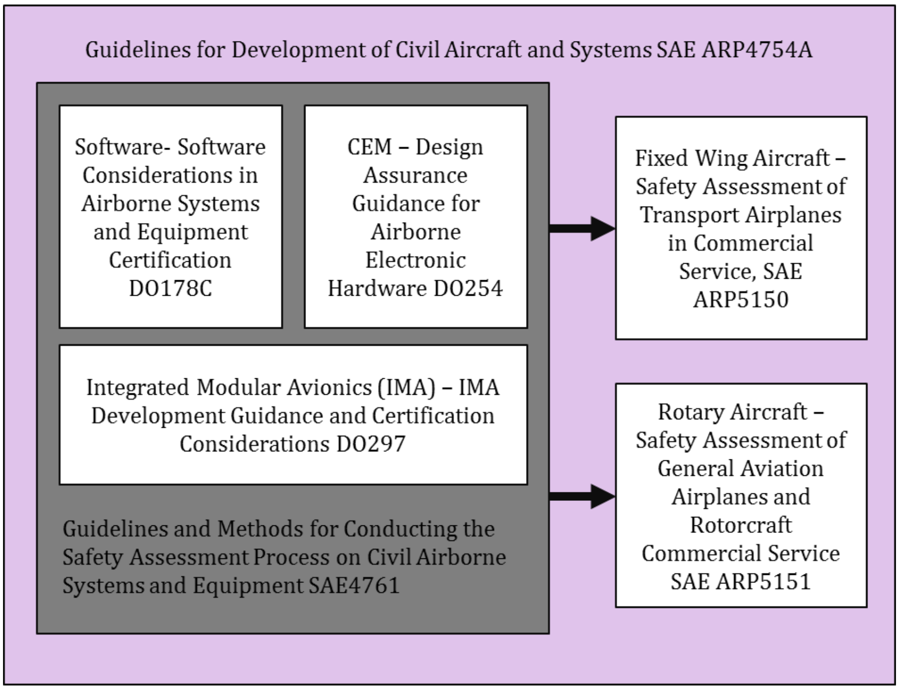

Guidelines for Development of Civil Aircraft and Systems SAE ARP4754A
Software- Software Considerations in Airborne Systems and Equipment Certification DO178C
CEM – Design Assurance Guidance for Airborne Electronic Hardware DO254
Integrated Modular Avionics (IMA) – IMA Development Guidance and Certification Considerations DO297
Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment SAE4761
Fixed Wing Aircraft – Safety Assessment of Transport Airplanes in Commercial Service, SAE ARP5150
Rotary Aircraft – Safety Assessment of General Aviation Airplanes and Rotorcraft Commercial Service SAE ARP5151

Figure 2.1: A Part of International Standards and Guidelines for an Aircraft

Although the international standards, guidelines, and recommendations offer a valuable support (i.e. products), they tend to address quite general issues (Pietrantuono & Russo 2013). The users might not find the right techniques to be used for their executing project directly from the international standards, guidelines, and recommendations. Furthermore, in the international standards, guidelines, and recommendations might not provide cost justification for the users. As a consequence, the effectiveness of these documents is questionable.

2.5 Solution to the Challenge of Coordinating Systems Engineering and System Safety Engineering Processes

In the early 1990s, researchers began to focus on safety as an important property to address in combination with design. Over the years, researchers have proposed approaches to harmonise activities within the design and safety disciplines. Despite the academic efforts to identify interdependencies and to propose combined approaches for design and safety, there is still lack of integration between them in the industrial context, as they have separate standards and independent processes and are often addressed and assessed by different organisational teams and authorities. Specifically, safety concerns are generally not covered in any detail in design specification.

One process for integrating SSE into SE is deriving detailed designs from more abstract designs to measure safety properties (Fenelon et al. 1994). This occurs one step before a direct safety assessment of the existing design (Leveson 2012). With the opportunities afforded by technology, this analysis can be done using a range of classical and computer-based techniques. The process must be observed carefully, however, since there is poor integration between system design and safety analysis, especially for software-based systems (Fenelon et al. 1994). The integration processes have to be documented and agreed in a company since communication is critical in handling any upcoming property in a complex system. Today's systems are designed and built by at least hundreds of engineers with different skills and therefore decision making has to be available to the right people at the right time, especially during the development process.

2.5.1 Positive Insights Regarding Integration from Industry

Practically, the integration of SE and SSE is done at an early stage of SE cycle. The information supported by SSE is significant to ranges of SE stages especially for safety related systems. System reliability resulted from SSE can be used to influence subsequent design decision at the conceptual design stage (Pai & Dugan 2002). For example according to Cepin and Markov research, requirement specifications can be improved by FTA developed for particular system (Cepin & Mavko 1999). Consequently, the estimation of reliability requirements estimation during the conceptual design stage is very important for system critical computer-based systems.

Trade-off are most effective when key attributes of the system such as performance and reliability are measured during the early critical design stages (Pai & Dugan 2002). Furthermore, consistency between SE and SSE can be ensured and a common understanding can be reached about the optimal architecture from both perspectives. These are the keys to avoiding the late detection of errors and thus reducing the time needed to develop complex systems (Pai & Dugan 2002). When designing a complex system, time and cost are very crucial. Design time and associated resources can be reduced when system design and its reliability characteristics are analysed as soon as they are available (Pai & Dugan 2002). This allows design engineers to decide if redesign is required.

2.5.2 Framework, Life Cycle Models, and Integration at Early Stage

According to Jon de Olmo, there are two objectives of safety analysis during the design stage (Olmo et al. 2018). First, the drafting of a safety case document that allows manufacturer to obtain a corresponding safety certificate. Second, analysis of the architecture of the system under development to ensure that it meets availability, reliability, and maintainability requirements. The latest objective is the most important for bridging the gap between SE and SSE. In some cases and complex system, analysing system architecture with safety analysis is an iterative process between SE and SSE.

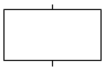
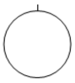
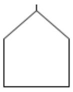
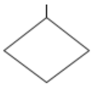
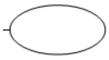
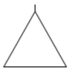
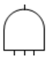
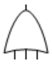
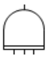
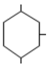
A methodology for the effective and efficient integration of SE and SSE, called safety integration in SE (SafeSysE), has been introduced by Faida Mhenni et. al. (Mhenni et al. 2018). SafeSysE uses a system modelling language (SysML)-based approach across three scopes. First, a formalisation of SysML-based is observed in SafeSysE to support safety analyses. Second, SysML is extended to enable integration of specific safety concepts in a system model. Lastly, an automated exploration of the SysML models is performed to generate the information needed to elaborate safety artefacts such as FTA and Failure Mode and Effects Analysis (FMEA).

2.6 Fault Tree Analysis as a Safety Assessment Method

FTA is a conventional system analysis technique for assessing safety in the development of a system. It is a deductive failure analysis method for determining the causes of undesired events. An event in a Fault Tree is a fault event. Fault events can be classified as internal or external (Joshi et al. 2005). Fault modelling by means of FTA begins by identifying an undesired top-level event, before then determining possible intermediate events until reaching down to a sufficient level. FTA has therefore been recommended to be constructed once the functioning of the entire system is fully understood. FTA facilitates both qualitative and quantitative analyses. A list of basic FTA notations used in the research based on ARP 4761 is presented as in Table 2.1. Events and logic gates, and the branches that connect them, comprise the standard structure of FTA. The detailed description of the construction of a Fault Tree is specified in the international standards such as IEC 61025 which is intended for cross-industry (International Electrotechnical Commission 2006), use including network data-loss prevention (Dirksen et al. 2009). Two types of FTA are defined in IEC 61025: static and dynamic (International Electrotechnical Commission 2006). Unlike ARP 4761, however, the IEC 61025 does not integrate into a lifecycle process for system safety engineering. Other types such as dynamic FTA (i.e. time related) (Čepin & Mavko 2002), formal FTA, and fuzzy FTA (Yuhua & Datao 2005) will not be considered in the research.

Fault Tree minimal cut set is the main concern in the qualitative evaluation of FTA. The cut set defines the minimum set of basic events that must occur in order for the top level undesired event to occur. This consideration straightens the independence of events to avoid significant error in analysis. When a Fault Tree is organised using minimal cut sets, the original Fault Tree structure that reflects the hierarchical deduction is often lost, however. Quantitative FTA uses Boolean algebra and probability theory to evaluate mathematically the probability of faults events occurring. In brief, in conventional FTA, qualitative evaluation is concerned with identifying information such as failure paths with the use of minimal cut sets, while quantitative evaluation is concerned with determining the probability of the top event.

Table 2.1: Fault Tree Symbols

Symbol	Description
	**Output Event** – output event (referred to as top event or intermediate events)
	**Basic Event** – event which is internal to a system without further development
	**External Event –** event which is external to a system
	**Undeveloped Event** – event which has little impact on the top event without further development
	**Conditional Event** – a necessary condition for a failure mode to occur
	**Transfer Event** – where fault tree information is transferred out to another fault tree or transferred into a fault tree
	**AND-Gate** – Boolean logic gate – output event occur when all intermediate events occur
	**OR-Gate -** Boolean logic gate – output event occur when at least one intermediate event occurs
	**Priority AND-Gate -** Boolean logic gate – output event occurs when all intermediate events occurred in a specific sequence (sequence usually represented by a conditional event)
	**Inhibit-Gate** – require an input and conditional event for the output event to occur

2.6.1 The Significance of Fault Tree Analysis

FTA has been a recommended method for system safety analysis for more than five decades, based on its focus on defining faults and structuring fault processes (Vesely et al. 1981). It is the most prominent technique for analysing complex and safety-critical systems (Ruijters et al. 2017). Fault analysis and resolution of faults should be part of any end-to-end system development process. For example, FTA also has been used to test the functionality and reliability of a system (Paiboonkasemsut & Limpiyakorn 2016).

Since the architecture of systems is widely designed using model-based approaches, failure of a system to function as intended is also evaluated through a model-based approach. In SSE, FTA is a prominent model-based approach methods (Volk et al. 2018) for determining various combinations of hardware and software failures and human error that could cause undesirable events (i.e. top level events) at the system level (Simha Pilot 2002). The construction of a Fault Tree is based on a deductive analysis technique and a top-down approach from the top level event to the basic level event (Dixon 2017). The connection between the top level event and basic level event, through intermediate level events, is demonstrated using Boolean logic gates (International Society of Automotive Engineers 1996). Consequently, finding of the original common Fault Tree development for analysing system hardware is supported (Xiang et al. 2005).

2.6.2 Model-based Safety Analysis

One approach to utilising systems models to analyse the safety of a system is called model-based safety analysis (MBSA) (Joshi et al. 2006). The safety analysis of a system is a process of evaluation to ensure that a mishap does not occur as the system performs its mission (Leveson & Stolzy 1987). According to Leveson and Stolzy, the first step in safety analysis is to identify hazards (path to a mishap) within the system followed by eliminating (or minimising) faults or failures leading to the mishap. These processes are aligned to ARP 4761.

Fault and failure are precisely defined by ARP 4761. An undesired anomaly in a system can lead to the loss of function or a malfunction of the system which means a failure can caused by a fault (International Society of Automotive Engineers 1996) and (Andrews, Bryans, Payne & Kristensen 2014). Faults can be modelled and analysed based on the functionalities of the system under development (Olmo et al. 2018). In the process of analysing faults, probability of occurrence and severity of the system functionalities are identified using systematic approaches that can include model-based approaches. This approach has been recognised amongst safety community as capable of overcoming the limitations of the conventional techniques when analysing complex systems during the system design stage (Olmo et al. 2018).

MBSA methodologies can be divided into two main groups: (i) failure logic modelling (FLM) and (ii) behavioural fault simulation (BFS) which is also known as fault injection (Olmo et al. 2018). These methodologies are used in different ways and result in different outcomes. Examples of MBSA tools with respect to their methodologies are presented in Table 2.2.

Table 2.2: Model-based Safety Analysis Methodologies and Tools

Failure Logic Modelling	Behavioural Fault Simulation
HiP-HOPS (Associate with Matlab Simulink (Lisagor et al. 2011) and (Olmo et al. 2018)) (Associate with AADL (Lisagor et al. 2011)) (Associate with NuSMV (Sharvia & Papadopoulos 2015)) (Associate with SPL (Oliveira et al. 2016))	SCADE (Lisagor et al. 2011) (Olmo et al. 2018) (Associate with PROVER plug-in (Bozzano et al. 2003))
FPTN	Matlab Simulink

(Lisagor et al. 2011)	(Lisagor et al. 2011) (Bozzano et al. 2003) (Associate with HIL platform (Olmo et al. 2018)) (Associate with ErrorSim (Saraoğlu et al. 2017))
AltaRica (Lisagor et al. 2011)	Statement (Associate with VIS model checker (Bozzano et al. 2003)) (Olmo et al. 2018)
	Cecilia-OCAS (Associate with AltaRica (Bozzano et al. 2003)) (Associate with IC3 algorithm (Bozzano et al. 2015)) (Associate with AltaRica 3.0 (Prosvirnova et al. 2013))
	FaultTree+ (Bozzano et al. 2003)

FLM is based on the automatic generation of safety analysis through safety assessment methods (i.e. Fault Tree and Failure Modes and Effects Analysis) using information stored in a component model of a system. Hierarchically Performed Hazard Origin & Propagation Studies (HiP-HOPS), Failure Propagation and Transformation Notation (FTPN), and AltaRica are three examples of FLM method of MBSA. The safety assessment models that are generated through FLM capture failure modes exhibited by other components of the system (Lisagor et al. 2011). The failure modes are the dependencies of the components in terms of deviation of their behaviour from design intent, which are typically defined at an abstract level in the system models (Lisagor et al.

2011). The application of FLM methods requires comprehensive knowledge of the system since the safety analyses are done based on design intent or what has been modelled (Lisagor et al. 2011). The FLM has been linked with other applications to enhance its performance. For example, its association with software product lines allows the systematic reuse of safety related information within the MBSA tools (Oliveira et al. 2016). Safety analysis of the dynamic behaviour of systems is a challenge for FLM tools, however (Olmo et al. 2018).

BFS is based on the injection of faults using formal models into modelled system (i.e. executable models) of the system so as to define their effects. The BFS methodology is therefore fundamentally based on formality and the extension of models (Bozzano et al. 2003). Safety assessment models that are developed using BFS methods can be defined through extension of the system models in the system design stage which are specified in languages such as SCADE or Matlab Simulink (Lisagor et al. 2011). The BFS method is therefore usually applied after designs and models are developed in the later stages of the development process (Olmo et al. 2018). The system configuration of different tools may have different input languages such as PROVER plug-in, VIS model checker, and AltaRica (Bozzano et al. 2003). The system models are extended with failure mode models so as to insert failure mode in the flow (Lisagor et al. 2011). Unlike failure modes in the FLM method, in BFS they are defined by components of the system itself through the failure mode model (Lisagor et al. 2011). This means that the BFS method delivers consistency between the safety analyses and the design model of the system (Lisagor et al. 2011). Failure mode models may provide unnecessary constraints which lead to unintended and incomplete analysis results, however (Lisagor et al. 2011). Furthermore, more failure fault models are required to analyse complex systems (Olmo et al. 2018). Linking FLMs with a tool for error propagation analysis of Simulink models such as ErrorSim can tackle the above challenge. ErrorSim allows this method to inject different types of faults and analyse error propagation to critical system parts and output (Saraoğlu et al. 2017). This method tightens the gap between system and safety development stages by sharing a common modelling environment, languages, and tools (Lisagor et al. 2011).In addition, more accurate analyses of dynamic system behaviour can be obtained (Olmo et al. 2018).

The MBSA techniques seek tighter integration between safety assessment and design artefacts. Research has been done to address the challenge so as to provide a high level of intellectual engagement on the system by associating FLM with BFS approaches such as Matlab Simulink (Olmo et al. 2018) and Architecture Analysis and Design Language (AADL) (Lisagor et al. 2011) which can limit safety assessment to considering only intentional interactions between components. Furthermore, combining FLM and model checking, such as NuSMV, works to assess how well safety requirements are satisfied (Sharvia & Papadopoulos 2015). Similar to FLM, research has been done to associate BFS with other applications such as OPAL-Real-Time Simulator (Bozzano et al. 2003). This allows the development and validation of a system based on hardware-in-the-loop (HIL) platforms to improve the results of the fault analysis with quantitative information about the effects of each fault mode (Olmo et al. 2018). The MBSA tools are capable to generate safety assessment such as Fault Tree and FMEA for different failure modes. In this thesis, both methodologies are used to generate Fault Tree results from total behavioural failure of a system through transformation methods.

2.6.3 Modelling Methods for System Safety Analysis

Modelling methods for SSA are comprised of qualitative and quantitative analysis techniques. For example, Fault Tree Analysis (FTA), Dependence Diagram (DD), and Markov Analysis (MA) are used to determine failures or combinations of failures at the lower level that might affect safety objectives (i.e. assurance level in probability evaluation) in graphical presentation.

Of all the available SSA methods, FTA is recognised as the main conventional technique to perform fault analysis (Olmo et al. 2018). FTA is also used together with other SSA methods such as FMEA, as an input and output. The safety of systems is analysed qualitatively and quantitatively in FTA. FTA is used for dependability analysis (Kabir 2017) and has been accepted as a reliable model by practitioners. The popularity of FTA as a reliable SSA method has led to the expansion of conventional FTA through research and discussion (i.e. formal and fuzzy).

2.7 Fault Tree Construction in Common Practice

The capability of FTA being used for modelling and analysing failures of components of a system has brought FTA to originally developed for hardware system analysis (Xiang et al. 2005). Generally, a Fault Tree is constructed based on individual component failures within a system. This means, the details of hardware components that comprise a system are used to develop the tree. System design engineers have the most information of the physical components of the developing system. This means that Fault Trees were often constructed after a detailed system operation diagram had been provided by system design engineers, i.e. after the system design stage (Xiang et al. 2005).

Events and logic gates are considered as the main elements of Fault Trees. Furthermore, events are associated with a change of state, i.e. a change of value of an attribute or a change of action of a function. Failure events in Fault Trees should therefore also be associated with a change of state. The relationship denoting the formation of a higher event is supplied by intermediate events through a logic gate (Kim et al. 2010).

From the perspective of Boolean algebra on a Fault Tree, a true (universal) formula is sought by mathematicians, whilst, predicted system behaviour is sought by engineers. The system behaviour is the change of attribute or action (International Council on Systems Engineering 2015). This change of action is starts with an event and ends with a state. For example, consider a simple system of a light bulb series circuit which consists of a light bulb, a power supply, switch, and wires to connect these physical components. As depicted in Table 2.3, the design intent of the system is when the light bulb and switch are in the same event. This means the light bulb follows the switch if it is switch ‘ON’ or ‘OFF’. In this case, an exclusive NOR logic gate is applied for the Boolean algebra truth values.

Table 2.3: Events of Design Intent

Light bulb	Switch	Design Intent
False	False	True
False	True	False
True	False	False
True	True	True

Formality in constructing a Fault Tree has brought to the attention of practitioners. A formal Fault Tree can be constructed based on states combination of components in a system (Xiang et al. 2005). The construction is grounded to the individual component failures and conditions that cause system failures. For instance, a system failure can caused by a combination of component states which is not a failure. A lot of research has been done to formalise Fault Trees and to generate Fault Trees from system models. Formal Fault Trees are constructed in various ways including using formal specifications to define the entities of Fault Tree and logic to form the structure of Fault Tree.

2.7.1 Fault Tree Support Tools

Nowadays, in this technology dependent era, systems are becoming increasingly complex and analysing the safety of the systems which integrate elements from diverse disciplines such as electrical, mechanical, and software has become more challenging. As this becomes a matter of concern, significant efforts are being made by industry and academia to developed computerised support tool that can help to reduce the difficulties in safety analysis arising from generating Fault Trees manually and from traditionally based on requirements and informal design which requires knowledge and experiences. Since FTA is the prominent system safety assessment method, the support tools are very useful to generate Fault Trees, i.e. from static Fault Tree and its extension. Some of the support tools come with packages of qualitative and quantitative analysis, while others have system safety assessment method generators.

There are various open source tools for Fault Tree generation, such as OpenFTA (Auvation n.d.), Fault Tree+ (Isograph n.d.), and Fault Tree Analyser (ALD n.d.). These tools allow users to generate Fault Tree manually and easily by dragging components from the provided list and dropping them into a worksheet area. This free concept allows the users to apply their knowledge to generate Fault Trees. Normally, the users need to do the safety analysis of the system in a separate session using the best of their knowledge prior to the drag and drop stage. Another open source tool which requires user knowledge of programming language is FaultCAT (Dehlinger & Lutz 2006). The Fault Tree generated through FaultCAT needs to be written in Java. By using FaultCAT, the generated Fault Tree can be applied to analyse a system for faults (Dehlinger & Lutz 2006). A more automatic support tool is Hierarchically Performed Hazard Origin & Propagation Studies (HiP-HOPS). This is able to generate Fault Trees automatically but requires data from architectural models specifically on system structure to achieve this (Papadopoulos n.d.). Furthermore, HiP-HOPS enables design optimisation alongside assessing failures of the system (Papadopoulos et al. 2011). The structure of the system can be redesigned and reanalysed to meet safety requirement, thus simplifying both SE and SSE aspects.

2.8 Model-based Systems Engineering and Modelling Techniques

A model is an abstraction of a context which includes systems, business context, and relationships (De La Vara et al. 2016). The elements of a system such as boundary, organisation, structure, and interaction are abstracted into a model. A model that represents a system is simplified to the stage of promoting understanding of the real system. In a model, essential details that contain all the elements needed to describe the system are emphasised and irrelevant details are omitted. Multiple perspectives of the system that include all parts of the entire system can be demonstrated through a model. This distinguishes a model from the concept of a diagram, which presents a single perspective and specific information of a part of a whole system. There are two uses of models in industrial practice (Zeller et al. 2016). First, the efficiency of safety engineering can be assessed as a standalone sub-task of system development. Second, the gap between functional development and safety assessment can be bridged. The latter use has a significant impact on the systems engineering life cycle is focused using model-based approach in this thesis.

Models have an important role in system development through their ability abstract the complexity of the system (Liebel et al. 2018). The benefit of models opens up new possibilities for creating, analysing, manipulating, and formally reasoning about the system at a high level. Model-based approaches have gained popularity in SE (David Long 2016). Model-based systems engineering (MBSE) is about elevating models in the engineering process to a governing role in the specification, design, integration, validation, operation, and safety of a system (Estefan 2007). In spite of its model centric approach for documenting information, MBSE is employed in the development of safety critical systems such as NASA's aeronautics research projects (Gough & Phojanamongkolkij 2018). Generally, the focal point of MBSE is in its methodology that covers processes and tools (Estefan 2007).

An industrial survey on MBSE has revealed positive feedback regarding the approach in various of domains such as providing abstractions of complex systems, simulation and testing, and support for performance-related decisions (Liebel et al. 2018). With implementation of MBSE, the processes in the system development life cycle are enhanced in terms of time, cost, and quality of the products. The success of MBSE in an organisation can be seen through the enhanced understanding and communication among the team members, and their ability to control the project, (Hutchinson et al. 2011) arising from the abstraction of the design of the project and the removal of unnecessary details (Espinoza et al. 2009). The productivity of the organisation and software quality can be improved (Baker et al. 2005) (Mohagheghi & Dehlen 2008) and errors can be detected at the early stage of development (Kirstan & Zimmermann 2010). This leads to a reduction of defects (Baker et al. 2005)(Mohagheghi & Dehlen 2008) and the potential for easy validation and verification (Espinoza et al. 2009). The MBSE approach can deliver a higher degree of automation, and cost savings by reducing accessible time and defect products (Kirstan & Zimmermann 2010).

Model-based approaches can be used to present the requirements and specification of a system during the conceptual design phase. This happens in the very first phase of the SE life cycle. The information for the requirement models is gathered from the stakeholders verbally or through documentation. Based on what has been specified in the requirement models, behavioural models are generated for the behaviour and function of the system. These behavioural models can have multiple presentations depending on the required level of detail. For example, the highest level of system behaviour is presented in informally by the Use Case. In the UML Use Case, the interaction between a system under development and its environment can be analysed. On the other hand, in UML Class Diagram, the components of the system can be presented using Classes. System models developed in the conceptual design phase can be reused in the verification and validation phase.

The flexibility of model-based technique means that it can be implemented as a tool for organising and managing requirements including specifications, standards, and guidelines at the early stage of system development. According to Gregory et. al., UML modelling language can be used to model safety-related concepts for aerospace system software in order to achieve software quality assurance requirements of the DO-178 standard (Zoughbi et al. 2011). The achievement of this assurance level supports system certification through RTCA DO-178 by improving communication and collaboration among stakeholders.

Seminal work regarding the mathematical foundation for SE and MBSE has been done by one of the founders of SE, Albert Wayne Wymore, (Wymore 1993), (Oren & Zeigler 2012), (Oren et al. 2018). The relationship between mathematics and MBSE has been applied to develop real systems. As defined in INCOSE's SE technical operation, MBSE is the formalised application of modelling to support activities throughout the SE life cycle phases (International Council on Systems Engineering 2007).

The application of model-based technique is to acknowledge the heritage of traditional, document-driven, programmatic reviews, and the challenge organisations face when attempting to adopt mode advanced, electronic or model-driven techniques (Estefan 2007). Although a model provides an abstraction of a system (or context), it considers relevant aspects of the system. The interpretation of the system is not affected by restriction of documentation script.

2.8.1 Modelling Language and Modelling Language Tools

There are two types of modelling techniques entailing graphical or textual presentation. Diagrams with notations and lines are used in the graphical modelling language. In this form, information is presented in structured shapes with less emphasis on text. In the textual modelling language, meanwhile, wording with standardised keywords and phrases are used in the presentation. A modelling language is a semi-formal language (Bondavalli et al. 2001) as the syntax of the presented models is well-defined although the semantics attached to the individual models is less formal (Fraser et al. 1994). Modelling languages, therefore, cannot be regarded as a complete formal modelling technique.

Flowcharts, simulations, and UML are a few of the techniques available for modelling. UML together with SysML are the products developed by OMG and have become the most popular standard of modelling language. They offer a practical way to present knowledge of a system. Various modelling language tools are available in the market such as ATLAS Transformation Language, Papyrus, Eclipse, Microsoft Visio, Enterprise Architect, Rational Rhapsody, and MagicDraw. The listed tools are software applications that support UML functions. Some of the modelling language tools are open-source software. Graph theory has been used in modelling complex architectures including data threads regarding the mission environment using MBSE tools i.e. UML and SysML, to validate viable complex architectures quantitatively early in the life cycle (Marvin & Garrett 2014).

One of the applications of MagicDraw, Class Diagram, can automatically be generated using the swimlane designed in Activity Diagram as the basis. The built-in application reduces the time needed to generate multiple system models by reusing information stored in the database. System architect engineers need to dig functional details to present attributes and operations as a complete Class, however. Class Diagram had been withdrawn from Enterprise Architect, however.

2.8.2 Modelling Language Tool with Verification Support

Modelling language tools provide dependable information at an early stage of system design. In 2008, the first precise operational and base semantics for a subset of UML object-oriented activity modelling was provided by the adoption of Foundational Subset for Executable UML (fUML), and it has been applied to state machine and class modelling constructs. The semantics defined by fUML specify a virtual machine for executing models compliant to the subset of UML for object-oriented modelling. fUML Action Language (Alf) was developed in order to enhance the practical viability of fUML (Seidewitz 2014). Alf notation can be attached to a UML model any place that behaviour can be. Thus, this becomes the success in philosophy attribution. The mature initial results of base semantics for a subset of UML models play an important role in the model's formal verification. Recently, users of SysML can breathe easily due to the establishment of Executable Systems Modelling Language (ESysML). Similar to fUML, precise language semantics is offered by ESysML by retaining SysML as the primary modelling construct (Amissah et al. 2018). fUML and ESysML are not required for a modelling language but semantically define and verify behaviour of UML and SysML models respectively.

2.8.3 Metamodel as a Models Managing Agent

Metamodela are a higher level of a model that represents a system, basically a model of models. Just as a model is an abstraction of a system, a metamodel is an abstraction of a model (Saeki & Kaiya 2006). A standardisation and concise definition of model elements are provided by a metamodel. The data flow diagrams and entity-relationship diagrams specified in a model are contained in the respective metamodel (Nakatani et al. 2001). The presentation of a model including semantics, structure, and the relationship between components in the model confors to the respective metamodel.

The most prominent metamodel of UML is described in Meta-Object Facility (MOF) by OMG. The MOF specifies a standard for metamodels that represent object-oriented concepts and systems. Models that are compliant to MOF can be exported and imported into different format, stored in a repository or transported across a network, and generated into codes. Another metamodel standard for describing models is Ecore by Eclipse research group. Models that use Ecore are supported with change notification and Java-based implementation.

Metamodeling is one of the important concepts of a systematic use of models as primary engineering artefacts throughout the engineering life cycle (refer model transformation in Subchapter 2.8.1) (Berramla et al. 2016). Metamodels have been used for many purposes across horizontal business domains. Surprisingly, metamodels have been used to resolve issues of safety compliance by holistically specifying the safety compliance needs when using models for safety critical systems (De La Vara et al. 2016). The implementation of metamodels in a project gives an advantage in modelling and managing specific information in the process of executing models to create a product compliant with standards and where key decisions are justified.

The benefits of metamodels have been acknowledged. The absence of a mature foundation for specifying transformations has led K. Czarnecki et. al. to propose a model transformation framework using metamodels to enable transformation between models and transformation between models to code (Czarnecki & Helsen 2006). The development of metamodels has been extended to integrating multiple models. According to (Banhesse et al. 2012), metamodels can be used for dynamic integration of elements from multiple models during the process improvement cycle. Although the developed metamodel is to address the challenges in software application, for generic purposes, metamodels have proved to be advantageous for tasks like supporting design changes with automated model development and model transformation, solving specific problem, improving development processes, and managing the architecture of models with intrinsic characteristic, as well as model transformation.

Modelling in multiple perspectives can be supported by using an overarching metamodel. For example, requirements descriptive model with capturing relationship between resources can be defined by using integrated Use Case Metamodel and Activities Metamodel (Nakatani et al. 2001). Furthermore, the developed requirements descriptive model based on the overarching metamodel of Use Case and Activities become more manageable. A metamodel that has encompassed both of safety and security analysis tools has been presented in (Ruijters et al. 2017).

2.9 Model Transformation of Different Domains

Model transformation is an important concepts for the systematic use of models as primary engineering artefacts throughout the engineering life cycle (Berramla et al. 2016). Model transformation enables translation of different models expressed in different modelling languages. It is increasingly used in software design and development, especially for synthesising two or more models, improving the development of model, verifying models, and simulating model. Model transformation has also been expanded for use as a medium to unify analysis tools to ensure that the developed model is the most reliable (Ruijters et al. 2017).

Nevertheless, practitioners have to deal with the problem of debugging and correctness testing in the transformation process (Burgueño et al. 2015). The quality of the resulting system is therefore highly influenced by the quality of the model transformations that are employed to produce the system. To support the capability of model transformation, languages, formalism, techniques (Jilani et al. 2010), processes, tools, and standards are needed.

Frameworks such as Relational Oriented Systems Engineering and Technology Tradeoff Analysis (ROSETTA) have been introduced to capture the relationship between entities of complex system such as SoS. This framework can be employed for model specification and relational transformation (Holden & Dickerson 2013) as it uses a mathematical concept to support traceability across models. Research into fault discovery from system functions and behaviour modelling (Zwolinski et al. 2000), fault modelling (Ingram et al. 2014), formal Fault Trees based on system state transition (Xiang et al. 2005), and structural relationships (Zhang et al. 2017) can be meaningful with the implementation of semantic transformation. For example, the Petri net model that uses a mathematical basis for node transitions can be generated from Activity Diagram by using XML transformation (Latsou et al. 2017). The flows of actions in the behaviour diagram are mapped to the transition in a graphical representation of the Petri net model. The nodes in the Petri net model are extracted from the actions and behaviour description of the Activity Diagram.

2.9.1 Transformation of Information between System Models and Safety Analysis Models

Model transformation can be used to bridge the gap between models of different domains such as between SE and SSE. Since SE and SSE have a common approach of using model-based design and analysis, model transformation is one the best approaches for taking the details from one model and transforming them to another model. The transformation method in the perspective of integration between SE and SSE helps engineers to produce system models and safety analysis models more quickly.

One of the purposes of the implementation of model transformation between domains in a system development is for analysing models within a certain timeframe. For example, in the COMPASS project, a tool that has been developed using COMPASS technology to conduct analysis on the architectural modelling of SoS at constituent system (CS) level (Andrews, Bryans, Payne, Dider, et al. 2014). An abstraction of a system model is needed for the tool to generate a temporal Fault Tree which is an intermediate model of a Fault Tree. The process continues by analysing the system model with the composition of fault events in the generated temporal Fault Tree.

In addition, with model transformation, safety assessments can be done concurrently while architecting a system. For example, Fault Tree and Failure Model and Effect Analysis (FMEA) can be generated from Use Case and Internal Block Diagrams. The procedures and steps developed to carry on the transformation process cover the physical aspects of the system. Similarly, a graphical block diagram of a system designed in MATLAB-Simulink Model can also be used to generate a Fault Tree. The MATLAB-Simulink model allows the ability to capture the dynamic and embedded environment of system, static and dynamic types of Fault Tree (Tajarrod & Latif-Shabgahi 2008). This is useful for a process combining multiple systems, such as with power delivery systems (Volkanovski et al. 2009).

Transformation from System Models to Fault Tree

As recognised by practitioners, the information underpinning in system models can be used as an input to generate Fault Trees. Hence, safety analysis can be performed in conjunction with system design and can be established at the earlier stage of conceptual design. Many studies have processes for model transformation between OMG's standard system models and Fault Tree models.

Practically, UML and SysML are a standard commercial modelling technique in SE. The establishment of these modelling techniques, however, requires knowledges and skills from various industries and offer different types of diagram to model a system graphically. For further development, models produced through these techniques can be connected to a variety of object-oriented programming languages such as C++ and Java, as well as to architectural description languages such as VHSIC Hardware Description Language (VHDL). In SSE, meanwhile, Fault Tree is the most well-known safety assessment method and the model best suited for the analysis of system development.

Frequently, model transformation between domains involves intermediate tasks. There is no Fault Tree generation done directly from the SysML system model (Mhenni et al. 2014). For example, the connectors and ports between the subsystem modelled in the SysML Internal Block Diagram (IBD) illustrate the internal structure of a subsystem that can be manipulated fto explore fault and failure propagation through that subsystem (Mhenni et al. 2014). Logic gates and fault events, which are the main elements in Fault Trees, can be derived from the IBD by tracing a graph traversal algorithm and identifying entry and exit patterns within the diagram. Another method for transforming IBD to Fault Tree is by combining with Sequence Diagram and using a reliability configuration model (RCM) (Xiang et al. 2011). The structure of a system, and functional dependencies between the system components, are specified in RCM with Maude, an executable algebraic formal specification language, before generating a static type of Fault Tree.

From a UML Activity Diagram perspective, Activity Diagram can also be used to generate Fault Trees. One study uses Activity Diagram to generate Fault Trees as test cases in respect to the process flow illustrated in the Activity Diagram (Paiboonkasemsut & Limpiyakorn 2016). The process flow is used to generate a condition-classification tree model which is ultimately used to derive a table of test cases. The test cases generated from Activity Diagram cover the functionality and reliability of the developing system. One of the advantages of using Activity Diagram for generating test cases is the early detection of faults and a reduction in development time (Patel & Patil 2013). This proposed Fault Tree development technique is incompatible for a large number of decision nodes, however.

Transformation can also be done between multiple UML system models and Fault Trees. For example, an algorithm used to synthesise dynamic Fault Trees automatically is logically developed based on multiple structure models (Pai & Dugan 2002). The algorithm that encodes structural components and the associations between components are extracted from Class, Object, and Deployment Diagrams. All the information comprised in those models are compiled in Rational Rose before generating the algorithm. In the implementation of the method, Class Diagram plays an important role as each class in the diagram is simply taken as a basic event for the Fault Tree. The application of the algorithm is also known for transforming details from Fault Tree to generate State Machine in pseudo code (Kim et al. 2010).

In one study, ATLAS Transformation Language was used to transform a number of UML system models to Fault Trees using entity mapping from the UML system models. The basic components of the UML Composite Structure Diagram, Sequence Diagram, and Use Case Diagram with the extension of Modelling and Analysis of Real-Time Embedded systems (MARTE) and Dependability Modelling and Analysis (DAM) profiles, are taken as the inputs to generate the Fault Tree by means of the ATLAS Transformation Language to generate Fault Tree (Zhao & Petriu 2015). In the research, entities mapping is applied to generate Fault Tree from the UML system models.

2.10 Implementation of Formal Methods on Models and Model Transformation

Formal methods are a particular kind of mathematically-based approach in which a statement can be formal with mathematically correct. It is used extensively in formal language to pattern arguments (statement). For example, proposition and predicate calculus are used to design and pattern a complete statement with premises, conclusion, and a relationship between premises. In software and computing, formal language has been extensively used for software programming such as Alloy, Z, B, and OCL.

The formal method has advantages in terms of analysing and verifying development in any part of the SE life cycle. The implementation of formal method helps engineers in a project team to receive the same standard information about the project. The formal method has been implemented in various scopes such as for centralising details to a primary context in safety cases (Denney, Pai, & Whiteside, 2015). This gives a strong basis for identifying and ordering safety cases, which is useful for safety specification, safety analysis, and system testing. Other than safety, formal methods contribute to the reliability and robustness of a system design with respect to its requirements. For example, in the aerospace industry, formal methoda are used for integrating the design and safety analysis of the system (Bozzano & Villafiorita 2003). The integration work provides a better working environment in that design engineers can formally verify a system and safety engineers can automate safety assessment together in one platform.

According to Daniel M. Berry, there are three main groups of formal method for software-intensive computer-based systems, namely verification, intensive study of key problems, and refutation (Berry 2002). The first group encompasses the partial or complete proof of a system. The second group delivers the intensive mathematical study of an aspect of the whole system. The third group verifies that the requirements of the system are correct. Benjamin Gorry, a lead engineer for product safety at BAE Systems, categorises formal methods into three different categories, namely theorem proving, model checking, and formal testing (Gorry 2015). These types of formal method verify, respectively, the reasoning of programs, system models, and system testing.

2.10.1 Application of Formal Methods

Formal methods can be applied at any point in the SE life cycle. For example, in the development of software-intensive computer-based systems, formal methods are best applied during the specification stage (Berry 2002). Usually, specification of the system and requirements from stakeholders is articulated in natural language. Here, formal methods help to minimise the associated difficulties and ambiguity for determining the actual specification of the developing system. With the finest specification, full traceability to test cases can be accommodated (Pietrantuono & Russo 2013). Through this it is possible to reduce the incidence of major errors only coming to light at the end of the development process. Formal methods also support users when handling engineering's problem domain (Berry 2002). For example, fixing accident of software-based system increases productivity and makes coding easier and less error prone.

2.10.2 Industry Feedback on Formal Methods

Formal methods are no longer a strange to the average engineers, especially in design teams. A survey has been carried out to compare the practice of formal methods between 1990 and 2009 (Woodcock et al. 2009). According to this survey, the implementation of formal method technology as a part of industrial development processes was discussed in the early 1990s, leading to improvements in practice by the following decade. Based on an article of a working group that supports the application of formal methods application, it is recommended for implementation in industry (Clarke & Wing 2002). The positive feedback has also led to National Aeronautics and Space Administration (NASA) publishing a guidebook on the use of formal methods for the specification and verification of software and computer systems (Kelly et al. 1998). The positive feedback of formal method from industry can be seen by the adaptation of formal method within the most critical parts of a system and in safety-critical systems such as automotive, aerospace, and control system domains.

2.10.3 Challenges Facing the Application of Formal Method in Industries

Regulation is one of the challenges that industry has to accept. For example, in the development of systems with software dependability, industry is required to comply with certification standards such as DO-178 (Pietrantuono & Russo 2013). DO-178 is a safety standard for software in airborne systems, and has to be complied with by relevant manufacturing industries. In the standard, formal methods are highlighted as a verification technique for the software development (Brosgol 2011). Engineers with specific skills are required to implement formal methods. Industry, especially for manufacturing large-scale systems, has to bear the costs associated with acquiring the necessary skills and technology.

2.10.4 The Need for Formality in Respect to Models

Models are a medium of communication and documentation for reflecting important information about a given context. For example, Activity models can be used to understand the behaviour of a system (Object Management Group 2017d). The behaviour of system is modelled from user perspective (Felderer & Herrmann 2018). A challenge with Activity models, however, is that if two system architects were asked to produce an Activity model for the same system behaviour specification, it is quite likely that they would be different. This contrasts with formal analysis: it is probable for example that two different safety engineers would produce the same unique Fault Tree representation can be produced by two safety engineers.

Since UML modelling language has been widely used in industry, multiple users are able to model and amend UML system models at the same time with the power of web technology (Kurniawan et al. 2014). This approach helps the users to easily follow up on the system being modelled and spot any changes done on the model. The application of formality in the system models could bring better interpretation, however. In order to help modelling language users, many applications and tools for complementing informal modelling technique with formal methods have been established in the market. For example, the construction of Use Case applies informal techniques to define and analyse system behaviour at the early stage. The UC-B plug-in can be used to perform formal assurance on the Use Case model based on set theory of Event-B (Murali et al. 2016). Furthermore, the application of fUML can supports high level conceptual models for the design of the architecture of more complex systems (Amissah et al. 2018). By using fUML, behaviour models such as Activity and State Machine are developed with precise semantics.

Formal methods can be used for proving any development outcomes, be it statement or model, with specifications and requirements. Formal methods support a consistent form of specification and requirement to the end of system development. With a precise system architecture and design, a rigorous analysis can be made. This makes verification and validation processes easier and faster.

2.11 Summary of the Literature Review

Systems engineering and systems safety engineering are two different engineering domains. In a development of any large system, which is complex and safety related, both domains have individual line processes but are inherently inseparable. The system design and safety communities are aware of each other's needs understand that design is not only for system functionality but for safety as well. Since model-based approaches are being implemented for the realisation of some of the processes, there is a need for easy-to-use tools that are able to generate reliable models automatically.

Model-based Safety Analysis (MBSA) tools such as HiP-HOPS and SCADE are developed to adequate the demand from the communities. These tools are well established and used during design of safety critical system for generating reliable models. The tools help safety engineers to generate Fault Trees using alternative method particularly by using system model as input. By setting the Fault Tree generation as a common goal, these tools have also been used as a starting point for an expanding body of research work by looking at different viewpoints of system models.

The information of Fault Tree can be transformed from multiple types of system models. For instance, Zhao and Petriu take inputs from UML Composite Structure, Interactions, and Use Case to generate Fault Tree by using ATL (Zhao & Petriu 2015). The transformation from multiple types of system model can represent different viewpoint of a system. Furthermore, the inputs from different types of models give different level of information in the Fault Tree. For instance, the input from Composite can be traced to basic events and the gathered inputs from Use case and Interactions can be traced to intermediate events in the Fault Tree. There are also, in the research areas, where the generation of Fault Tree is transformed from a single Use Case of a system (Hu et al. 2011). However, the information provided by the Use Case just to support top event and some of intermediate events in the tree. In the research conducted by (Paiboonkasemsut & Limpiyakorn 2016), a Fault Tree that generated from an Activity Diagram is an approach to support validation test case for system functionality. The single type of diagram transformation had become one of motivations of this research to generate fault Tree from behavioural aspect of a system specifically Activity Diagram.

This thesis is concerned about formality. Most of the published research which regard to the concern present the ways for generating formal Fault Tree. For instance, as presented in (Xiang et al. 2005) and (Ortmeier & Schellhorn 2007), disjunctive normal form is applied to develop formal Fault Tree. The analyses of the developed formal Fault Tree are done based on the states transition concept of an operating system. This concept underlies cause-consequence relation in the developed formal Fault Tree. The generation of fault Trees from model-based systems engineering (MBSE) lack of semi-formal transformation. Many of the published research apply transformation language such as AltaRica (Yakymets et al. 2013) and ATL (Zhao & Petriu 2015) to get it formal. However, using transformation language puts formality from input to output which maps entity-to-entity. As a result, the generated Fault Tree is absent with relational structure of system models.

Methods for transformation between UML system models and Fault Tree models are proposed in this thesis. This is to support the conventional way of constructing Fault Trees which do not support structured analysis. Mathematics is employed in the development of the methods in order to achieve semantic precision. By using these methods, the relational structure of UML system model is preserved in the Fault Tree. The methods are suggested to be used at the early stage of system design since the elimination of fault at the earliest time is a key to improving the productivity of the development process.

CHAPTER 3

FOUNDATIONAL RESEARCH KNOWLEDGE

3.1 Introduction

A set of foundational knowledge that are necessary for this thesis is presented in this chapter. The foundational knowledge provides a technical basis for formal transformation methods later in Chapter 4 and Chapter 5. Generally, the technical basis is divided into two segments: (i) model-based approach, and (ii) mathematical representation. In this thesis, the model-based approach is used as representing system at the architecture level. There are two aspects of the system architecture to be focused which are system function and system component. These aspects will be presented by using UML Activity and UML Class respectively. Nevertheless, the model-based approach is also used in assessing safety of the system. In particular, Fault Tree models faults and errors that lead to the system failure. These models are the focal points in the formal transformation methods. Each of UML Activity, UML Class, and Fault Tree model which can be presented by a higher model called metamodel are also described in this chapter. Despite, these models apply graphical presentation, if not more formal. This came across the second segment of the technical basis which is mathematical representation. In this segment, mathematics that can be performed by the models are discovered to support the formal transformation methods later in Chapter 4 and Chapter 5. For the UML models, propositional calculus is applied to the models for a constructive proof of the model elements and the structure. For the Fault Tree, probability theory that lies behind the presentation of the Fault Tree utilise Boolean algebra for estimating failure of the system is demonstrated. The mathematic representations are the key features of the transformation methods. Therefore, a precise presentation of the models can be offered for formal transformation from system architecture to system safety and reliability models. The application of the developed methods is demonstrated on an authorised Traffic Management System of System (TMSoS) case study which is reviewed in this chapter.

This remainder of this chapter is structured as follows:

Subchapter 3.2: Model-based Approach for System and Safety

As applicable for this research, specific characteristics of the system models such as model elements and structure of the models are discussed. Two types of models in UML which are Activity and Class are considered as for modelling architecture of system in this thesis. Nevertheless, one model of safety assessment method, Fault Tree, is also discussed. Metamodel that described each model are also presented.

Subchapter 3.3: Mathematical Representation of Models

As one of the key features of formal transformation methods, mathematical basis is provided to support the representation of the model-based approach. First, the propositional calculus is applied to system architecture models that presented in UML. Logic operations are also applied along with the propositional calculus that has been determined for the system models. This involves the negation as a fault representation of the models. Then, the probability theory that has been implemented in Fault Tree is explained. Furthermore, unlike the mathematics provided for the structure of the system models, the symbols in the tree used as the mathematics operation are also explained.

Subchapter 3.4: Traffic Management System of Systems Case Study Review

An authorised TMSoS case study is briefly explained in this subchapter. The TMSoS case study is used to demonstrate the model-based approach for presenting system architecture and assessing system safety and reliability. Nevertheless, the case study is also used to demonstrate formal transformation methods developed in Chapter 4 and Chapter 5.

Subchapter 3.5: Summary of Foundational Research Knowledge

In this subchapter, a summarisation on the two key features of formal transformation methods and a case study for demonstrating the application of the formal transformation methods is presented.

3.2 Model-based Approach for System and Safety

In this thesis, formal transformation methods will be developed for automated system safety assessment generation from system architecture design. The system architecture design and system safety assessment are defined by using model-based approach. The system architecture design used in the development of formal transformation methods is modelled in two types of UML diagrams for representing behaviour and structure of a system. The behaviour and structure of the system are modelled in UML Activity and UML Class, respectively. The scope of UML Activity and UML Class Diagrams is the basic notations and relationships that often used in modelling. This means a subset of Activity and Class Diagrams is selected for the formal transformation. For generating system safety assessment, the modelling of failure will be defined based on the transformation of the subset of the diagrams. The transformation suggests relational structure mapping with formal approach to the methods. The work proposes to start with the basic structure that identifiable in these models. Fault Tree is one of the safety assessment methods that presented in a graphical model. Fault Tree is selected for designing safety assessment for the system.

In the rest of this subchapter, the model-based approach to the behaviour and structure modelling, and safety analysis with the basic notations of the models are described. Metamodels that emphasise the fundamental structure and basic notations of the subset of each models are developed and are also described.

3.2.1 Model-based Approach to Behaviour Modelling

According to International Council on Systems Engineering (INCOSE) (International Council on Systems Engineering 2015), behaviour of a system can be classified into two types: dynamic and emergent. Dynamic behaviour of a system is based on the time evolution of the system state, whilst emergent behaviour can be seen collectively in a large scale as it cannot be understood in terms of individual system elements (International Council on Systems Engineering 2015). This thesis is primarily concerned with dynamic behaviour.

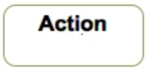
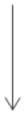
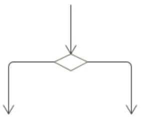
The execution of system functions is directly associated with the behaviours of a system. And a change of system states is often led by the execution of a system function. Hence, the modelling of the behaviours of a system can hold a functional viewpoint or a state viewpoint. In model-based approach with UML, the two viewpoints of system behaviours are led to four different model representations namely Use Cases, Activities, Sequences, and State Machines. First semantic transformation is demonstrated on Use Cases and Activities to analyse SoS design using the graphical language. Activities is further analysed for the development of an overarching metamodel in Chapter 4.

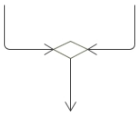
Activity Diagram as a Behaviour Model

The sequencing of actions of a system or SoS is specified by UML Activities. Here, *Action* is the technical term and a metaclass used in UML to represent the fundamental unit of behaviour specification (Object Management Group 2017d). For the rest of this thesis, for clarity, Pascal case format is adopted, i.e. concatenating capitalised words, in the naming of metaclasses, e.g. ActivityNode. In the case of modelling system behaviour with Activities, an action is represented by a node (ExecutableNode), refers to an elementary step to be executed by the system. To show the sequencing of the steps, the nodes are then connected via directed edges (ActivityEdge). A list of graphical notations available to the modelling of system behaviour in Activities within the scope of the research is provided in Table 3.1. In addition to executable node, the use of other types of nodes such as control nodes and objects are involved in Activity modelling. As objects do not represent system functions, they will not be considered in the research for the purpose of functional fault analysis. The graphical complexity of an Activity model is also reduced by only include control flows.

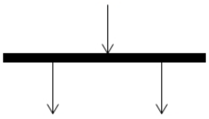
To describe the execution of an action and the flow of controls, the concepts of tokens (which are not explicitly modelled in Activities) and guards are used in UML. After a function completes its execution, a control token will be offered to the next node via the edge that connects them. Furthermore, edges may have guards on them. A token can only pass through an edge with a guard if the guard evaluates the tokens to true for the offered token. Guard is commonly used on the outgoing edges of a decision node (DecisionNode). The foundation for formalising control flows by using propositional calculus is provided by the concept of passing a token after the successful execution of an action. In brief, for two connected actions via an edge, if the execution of the second action is completed, the first action is supposedly completed its execution as the token must have been offered and accepted by the second action for the second action to execute.

Table 3.1: Activity Model Symbols

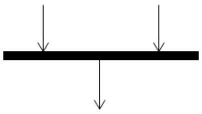
Symbol	Description
	**Action** – action state of system behaviour
	**Control Flow –** directional activity flow of control nodes and action states
	Initial Node – initial state of activity flow
	Activity Final Node – final state of activities completion
	**Decision Node** – point of alternate paths decision



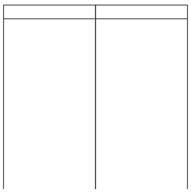
Merge Node – point of multiple flows merge to a single flow without synchronization



Fork Node – point of single flow splits to multiple flows



Join Node – point of multiple flows synchronize to a single flow



Swimlane – classify and hold activity flows according to systems in partitions

Reduced Activity Metamodel

Activities Metamodel is served as a reference to Activity modelling by defining the abstract syntax and the interrelationship between model elements in the standard Activity model (Liu 2010). As depicted in Figure 3.1, within the scope of this research, only a subset of the UML Activities metamodel, such as ActivityNode and ActivityEdge, is considered for the development of the overarching metamodel. Metaclasses, such as Object and ObjectFlow are neglected due to their irrelevance to fault modelling from a functional viewpoint. These metaclasses hold object (facility) during the course of the execution of an activity. For instance, a facility has a potential to be modelled by using variables. This means that a facility possibly has multiple states. Therefore, in defining the fault modelling of the component by using proposition would be more than just True or false. Nevertheless, the propositions could not systematically capture all of those variables. In addition to this subset, Action metaclass is also included based on previous discussions. This metamodel will be referred to a Reduced Activity Metamodel (RAM) in the rest of the thesis.

In the RAM, at the top level, ActivityNode and ActivityEdge are associated with each other through two relations. In the modelling of control flows, two cases are represented by these relations: (i) an edge comes after a node, and (ii) a node comes after an edge. Detailed descriptions of the lower level metaclasses are provided as follows:

1. Two types of nodes are generalised by the ActivityNode: ExecutableNode and ControlNode. These activity nodes are points of intersection where respective operation takes place in the Activity model.

2. The ExecutableNode is the generalisation of Action which specifies the actions to be executed in the Activity model. The proposition that describes an action is captured within the Action symbol (Table 3.2).

3. A set of paired nodes that are used to manage different types of control flows is generalised by the ControlNode. These pair nodes are: (i) InitialNode and FinalNode which are used to indicate the starting and ending point of a flow respectively; (ii) ForkNode and JoinNode which are used to specify concurrent flows; and (iii) DecisionNode and MergeNode which are used to specify alternative flows. Although the pairings are not reflected in the RAM, paired usage has been regarded as reflected in the RAM, paired usage has been regarded as necessary practice for semantic consistency. As many of these control nodes allow multiple coexisting (concurrent and alternative) flows to be modelled, it is therefore possible to have situations where a single activity node is associated to multiple activity edges. These situations are covered by multiplicity on the association line where an asterisk symbol is depicted toward the ActivityEdge end.

4. In the UML 2.5.1 specification, guard is not explicitly captured by a metaclass. Instead, the concept of ValueSpecification is used by the metamodel.

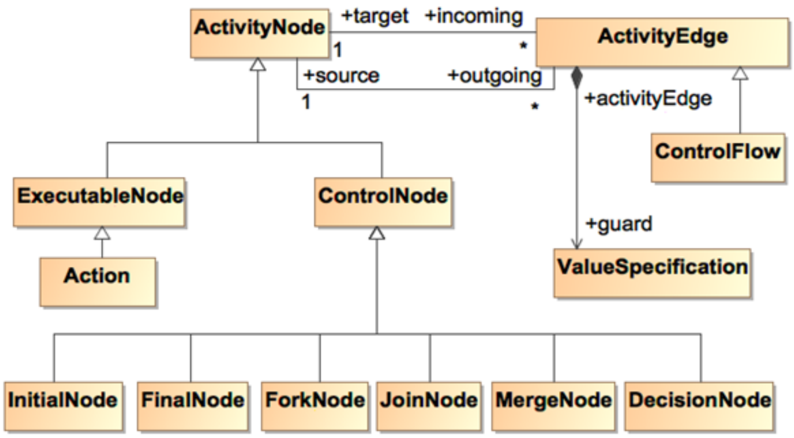


Figure 3.1: Reduced Activity Metamodel extracted from the OMG UML Specification (Object Management Group 2017d)**.**

3.2.2 Model-based Approach to Structure Modelling

The organisational representation of system structure is defined by elements of a system which also called components in this thesis and their interrelationship (International Council on Systems Engineering 2015). The structure of a system shows the static organisational of components and the relation to each other in the achievement of the stated purpose of a system.

There are seven types of diagrams that have been categorised for UML structure diagram by the OMG (c.f. Figure **1.2**). Each type of structure diagram presents a different view of the system structure. One of the structure diagrams of UML is Class Diagram that shows structure of the designed components as classes with features and their relationships. The Class Diagram is the most widely used structure diagram, as it is the richest diagram in terms of the amount of syntax available to the modeller. As suggested by (Kurniawan et al. 2014) using Class Diagram as the only structure diagram for describing system structure in UML tools collaboration web-based project of more than one user at a time is sufficient. The capability of Class Diagram of providing much information of a system has brought to the establishment of Requirements Analysis and Class Diagram Extraction (RACE). Class Diagram can be extracted directly from large volumes of textual requirements including interview excerpts, documents, and notes by using RACE (Ibrahim & Ahmad 2010). Despite, being a semi-formal modelling tool, an approach to verify Class Diagram in providing syntactic correctness with respect to requirements has to be introduced (Chanda et al. 2009).

Class Diagram as a Structure Model

The connection of elements of a system or SoS through the features of the objects is specified by UML Classes. A Class symbol is the primary unit in a Class Diagram that specify classification of an object or a set of objects in a system through structural and behavioural features. Respectively, these features are named as properties that also known as attributes (Property) and operations (Operation). These features are also presented in the Class. Features of each Class are unapproachable by other Class. However, Class which has ancestor is allowed to approach its ancestor's features as the Class presents its ancestor. The modelled Classes may have semantic relationship with each other represented by the embedded properties. This influence the configuration of collection of Classes. In some cases, from a viewpoint of system and subsystem, a multiple of Classes are supressed into a single Class. A list of symbols of the Class Diagram discussed above is provided in Table 3.2.

For the diagrammatic presentation, without starting and ending points like Activities, the configuration of Classes is formed by the collection of Classes and connection lines called relationship. The relationship is normally drawn as a solid line connecting two Classes. In this thesis, four common types of relationships are focused. The types of relationships as specified by the OMG are Association, Generalisation, Aggregation, and Composition. The Association is used to indicate a direct relationship between Class and associated Class. In a way to read the modelled Classes, a solid pointing triangle can be attached on the connection line to indicate the reading direction.

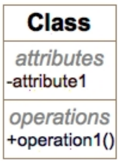
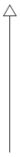
The Generalisation is a type of relationship that uses to organise the hierarchy of Classes by a common feature. For simplicity, the concept of child and parent can be applied of showing one or more child Classes attached to a parent Class by a relationship. The parent Classis always modelled on top of the set of Classes. In some literatures, Generalisation is called Inheritance as the set of child Classes seem to inherit the parent Class. The Classes at the lower hierarchy presents the ancestor.

The Aggregation relationship is used to model circumstance of container (also called whole) Class and part Class their properties. The part Class can stand alone if the container deleted. Furthermore, one part Class could have more than one container Classes. For example, University and Professor are connected by Aggregation as a container Class and part Class respectively. The professor is at the university to teach and exist whenever the university does no longer exist. The professor also can teach other than university which shows Professor can has Aggregation relationship to another container Class.

The Composition relationship presents stronger relationship as owner and part Classes. The part Class is owned by the owner Class. For example, the Composition relationship can be used to model House Class and Room Class as owner and part Classes respectively. When the owner Class does no longer exist be it deleted, the part Class could never be existed.

In the UML Classes, multiplicity is used to indicate the cardinality of objects (Class) that have relationship to other Class. The multiplicity can be as lowest as zero, i.e. multiplicity is not applicable. The multiplicity can be placed near the end of the line of the particular Class. The presentation of multiplicity is not a necessary. If no multiplicity is shown on the diagram, no conclusion may be drawn about the multiplicity in the model.

Table 3.2: Class Model Symbols

Symbol	Description
 Class attributes -attribute1 operations +operation1()	**Class** – represents a classification of an element or a set of elements in a system
	Association – relationship that shows link between Classes
	**Generalisation** – relationship that generalises features of one or more child Class by the parent Class (the triangle attaches to parent Class)
	Aggregation – relationship that describes one or more Class as a part of the container Class (the white diamond attaches to container Class)
	Composition – relationship that describes one or more part Class as the composite of the owner Class (the black diamond attaches to owner Class)

Reduced Class Metamodel

Similar to the purpose of Activities Metamodel to serve as a reference of a specific type of modelling, Structured Classifiers Metamodel is also served as a reference to three types of structure modelling by defining the abstract syntax and the interrelationship between model elements in the standard structure models (Object Management Group 2017b). As UML Class has been selected for presenting structure model in this thesis, the relevance metaclasses of the Structured Classifier Metamodel, such as Class and Relationship, are considered for the development of the overarching metamodel. Metaclasses such as EncapsulatedClassifiers and Collaborations are neglected due to their purpose of specifying interaction between Classes. Furthermore, metaclass such as Component is neglected as it specified modular unit which require more than True and False propositions for modelling the fault. For instance, Component covers 'black-box' and 'white-box' views of an element of a system. The proposition should address both views of what causes the faulty of modelled elements. In addition to the relevance metaclasses, Generalisation is also included based on previous discussions. As depicted in Figure 3.2, This metamodel will be referred to a Reduced Class Metamodel (RCM) in the rest of this thesis.

The RCM can be presented according to the metaclasses at the top level, Class and Relationship. These metaclasses marked the symbols presented for Class Diagram in previous subchapter. Detailed descriptions of metaclasses are provided as follows:

1. Class owns Operation and Property. Operation and property are required for a Class as a Class classifies object or a set of object through the structural and behavioural features.

2. Association has member ends that represented by Property. The Association relationship indicates semantic relationship of Classes that represented by properties.

3. Relationship generalises Association and Generalisation. Association and Generalisation are type of relationship of Classes. Association may also represent Aggregation and Composition relationships according to group Classes.

4. MultiplicityElements specify the collection of value that includes the value of property. Multiplicity is presented near the end of the line of particular Class to indicates the cardinality of Class that presented by property of the Class.

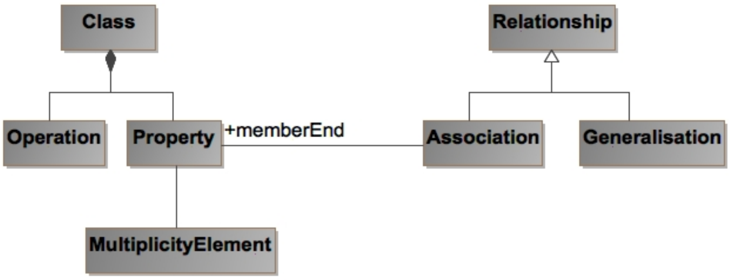


Figure 3.2: Reduced Class Metamodel extracted from the OMG UML Specification (Object Management Group 2017d)

3.2.3 Model-based Approach to Fault Analysis

As mentioned in earlier, model-based approach to fault analysis, such as FTA, Reliability Block Diagrams, Binary Decision Diagrams, Dependency Diagram, and Markov Analysis are widely used for system safety assessment. These model-based techniques are often used by Safety Analysts to evaluate system architecture to quantify probabilities of occurrence of system failures, and to identify potential related risks. In this subchapter, essential background knowledge in FTA is provided and a Fault Tree Metamodel (FTM) is constructed based on safety standard ARP 4761 (International Society of Automotive Engineers 1996).

Fault Tree Metamodel

A standardised metamodel for Fault Tree currently remains unspecified (Zhao & Petriu 2015). Several research efforts have contributed to the construction of a generic metamodel for Fault Tree. For instance, the generic Component Fault Tree (CFT)

Metamodel (Adler et al. 2011) and a Fault Tree Metamodel (Zhao & Petriu 2015) are developed based on different viewpoints for particular research. CFT Metamodel is constructed based on the hierarchical decomposition of a system. It emphasises on the concepts of Component and Component Proxies based on component-based software development. Whilst, the latter metamodel is constructed based on the FaultCat analysis tool. Despite the different viewpoints, the two metamodels share a common set of metaclasses, e.g. Event and Gate (logic), which represent the backbone structure of a Fault Tree.

Metamodels of Fault Tree developed in the research are further extended and applied in other areas. For instance, the CFT Metamodel has been further integrated with Architecture Domain Specific Modelling Language to reduce efforts and times needed for safety analysis; and the integration also leads to potential reusable models. The metamodel developed in (Zhao & Petriu 2015) is being used for dependability analysis between UML model and Fault Tree. In this paper, metamodel of Fault Tree is developed for the purpose of unifying system functional architecture and system failure analysis.

The FTM developed in this paper, as depicted in Figure 3.3 is constructed based on ARP 4761 (International Society of Automotive Engineers 1996). Similar to the construction of the RAM, Pascal case format is used to denote a metaclass name to distinguish it from the name of an actual model element. Individual metaclass is explained as follows:

1. The main elements in Fault Tree model are events, branches, and logic gates. Hence, they are abstracted into metaclasses at the highest-level as Event, Branch, and Logic metaclasses respectively in the FTM. Similar to how ActivityEdge and ActivityNode are connected in the RAM, Branch is associated to both Event and Logic also by two-way relations. The multiplicity also reflects situations where an event (or logic gate) can be associated with one or more multiple branches going into the event (or logic gate) and out of the event (or logic gate).

2. Event metaclass consists of OutputEvent, PrimaryEvent, TransferEvent, and ConditionalEvent, each representing the corresponding type of event seen in Fault

Trees (c.f. Table 2.1). The PrimaryEvent further generalises BasicEvent, UndevelopedEvent, and ExternalEvent. The TransferEvent is a generalisation of TransferredIn and TransferredOut which represent the existence of external branch of Fault Trees. For the purpose of the research, BasicEvent can be further classified into two; FunctionalBasicEvent and ComponentBasicEvent.

3. Logic metaclass consists of ANDGate, PriorityGate, ORGate, and InhibitGate which is used to evaluate input branches and tie the branches together.

4. ConditionalEvent is associated with PriorityANDGate or InhibitGate. This reflects the use of conditional event in Fault Tree construction where a priority AND-gate and inhibit-gate is always accompanied with a conditional event specified in an oval shape (c.f. Table 2.1).

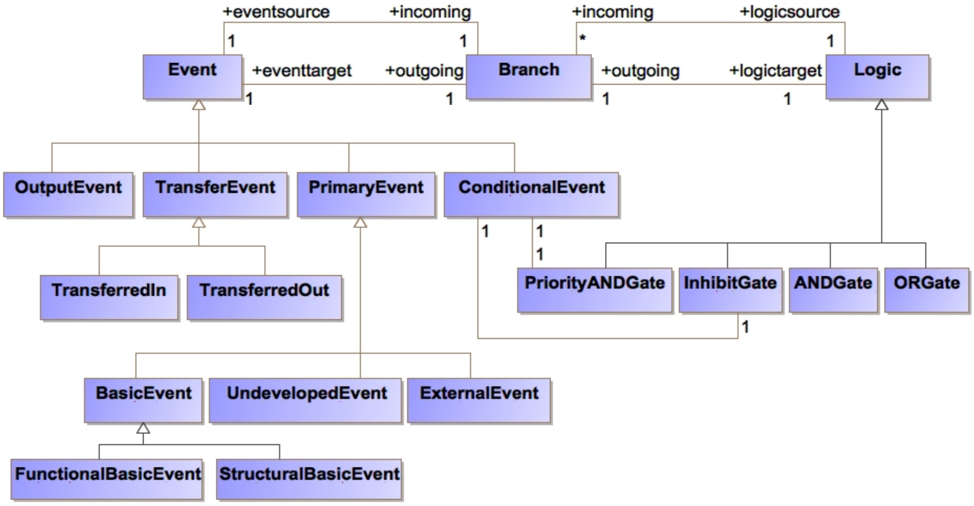


Figure 3.3: Fault Tree Metamodel developed based on ARP4761

3.3 Mathematical Representation on Models

Model-based approach is widely being used for designing conceptual of a system. It has become a powerful design technique especially in designing safety-critical cyber physical system (Jensen et al. 2011). In one hand, by using model-based approach, least documentation is produced in the abstracting complex system. On the other hand, rich information that is difficult to capture through documented source can be provided by using the approach (Wylie et al. 2016). The information stored in the mode-based document is the baseline decision lock before the progression of preliminary and detailed design (Woodward 2018). Being a critical document for designing safety-critical and complex system, a better reasoning is the vital interest in modelling.

As formality in models transformation method is the concern in this thesis, mathematics on the correspond UML models in the scope of this research is emphasised. The logical models of UML are precisely presented by using mathematic. The presentation of the logical models uses the basic pattern of arguments and conditions: Propositional Calculus. This includes logical operations such as conjunction and negation. The precise semantics of UML models will be the basis of Fault Tree transformation. The formal transformation method will be supported with probability analysis performed by Boolean logic gate in the Fault Tree. In this subchapter, the mathematic applications for the formal transformation methods are discussed for UML models and Fault Tree model.

3.3.1 Mathematics on Unified Modelling Language Model

In this subchapter, mathematical representation on UML system models that related to this research is discovered. As mentioned earlier, the graphical language is not a formal language. It is less formal but still has formality, i.e. semi-formal language. This is because, the construction of UML system models is formalised within the same specification logic based on predicates over arrow diagrams that correspond to Category Theory (Diskin 2003). In the context of this thesis, the predicates is related to the concept of relation between constructed nodes such as normally done in structural and behavioural of UML. Correspondingly, sketch procedure used in Category Theory is implemented in UML on the abstraction of a system in visual presentation that consist of directed multigraph of nodes and arrows, and marked predicate labels (Diskin 2003). As discussed in (Diskin 2003) on mathematics of UML, the sketch nodes and arrows with set and mapping including ordered pair is proven in Category Theory.

Application of Propositional Calculus

In this thesis, the system behaviour and system structure are modelled by using UML Activity and Class respectively. Generally, each particular node in the models is uniquely specified for designing a system. Thus, formalising the nodes by using propositional calculus will distinguish the modelled nodes and facilitate transformation from UML system models to Fault Tree.

Proposition is an assertion that expresses premise conclusion or argument which can be expressed in symbol or variable (Lemmon n.d.). The propositional calculus is the part of mathematical logic where the validity of an argument depends only on how the propositional sentences are formed and not on the internal structure of the propositions. This is sufficient for the modelling and analysis of functional and structural faults. Specifically, the behavioural proposition of interest will be of the following form:

$$p_i\text{: The Action } A_i\text{completed execution.} \qquad (3.1)$$

Note that this is a decidable declarative statement which has a yes-no answer; and therefore adheres Boolean calculus of evaluation of truth. The Action A_i can then be associated with {1,0} or {True,False} values. The truth values of the proposition will be regarded as outcomes of a designed experiment.

It is useful to view a coin tossing experiment from the behavioural viewpoint. The Action A_i can be stated as: the coin was flipped. Specifying the outcome is part of the design of the experiment. This could be as simple as specifying the coin began in one state {Head,Tail}, underwent a random change of state, and resulted in the coin coming to rest in a state, and resulted in the coin coming to rest in a state determined by the side of the coin facing up when the process completed execution. This process could fail to complete execution if, for example, either change was not random or the coin somehow came to rest on its edge. The outcome of this behaviour is silent on the end state of the coin. A second proposition could be introduced to complete the experiment; but this would not be a behavioural proposition.

In the relation between functions and components of a system, an expected system function is served by at least a system component. In recent development of industrial equipment, reliability, availability, maintainability, and safety (the latter is also referred to as supportability) (RAMS) (Eti et al. 2007) are defined as four key features that have to be analysed and managed throughout complex systems life cycle (Olmo et al. 2017). The practicality of the key features are dependent on systematic efforts as they can also be partly applied in many ways convenient to the industry (Saraswat & Yadava 2008). According to the stated key features in the development of industrial equipment, availability (which depends on reliability and maintainability) is concerned with describing system component (i.e. facility, element, or object). In this thesis, considering structural diagrams for system modelling, facility will be used to define a system component. The structural proposition of interest for the modelling and analysis of structural faults will be of the following form:

$$c_i\text{: The Facility } F_i \text{ is available.} \tag{3.2}$$

The availability of facility is not objectively means the physical availability of the facility. However, the availability of facility is based on duration of uptime and downtime for operation. The availability of facility is related to the service period of time (Eti et al. 2007).

From structural viewpoint, using the same coin tossing experiment as an example, the Facility F_i can be stated as: the coin and tosser were existed. The achievement of the coin flipping process is facilitated by the coin and tosser. It is important to note in this simple example, how the specification of the experiment immediately led to details associated with the internal structure of the propositions. Further analysis of system and faults involving non-functional properties will need the predicate calculus, which is a separate concern than what is presented in this thesis.

Representation of Faults

The sentences of interest will be well-formed formulae consisting of the types of propositions in (3.1) and (3.2). This will be referred to as behavioural and structural propositions. For any two propositions, taking p_i and p_j of behavioural propositions as an example, the sentences are declarations in one of the following forms or a negation of the form:

$$\neg p_i \qquad p_i \wedge p_j \qquad p_i \vee p_j \qquad p_i \rightarrow p_j \qquad p_i \leftrightarrow p_j. \tag{3.3}$$

where $\neg p_i$ is the negation of p_i, $p_i \wedge p_j$ is a logical conjunction between two propositions (behavioural); $p_i \vee p_j$ is a logical disjunction between two propositions (behavioural); $p_i \rightarrow p_j$ is material implication which is read as p_i implies p_j; and $p_i \leftrightarrow p_j$ is material equivalence which is read as p_i if and only if p_j. The proposition p_s will also be used and reserved for the completion of the overall execution of the system functions within a given Activity model. In addition, the proposition c_c will be used and reserved for the complementary system component within Composition relation Classes.

The analysts has choices as to how faults should be presented. From a logical viewpoint it can be useful to consider collections of propositions in disjunctive normal form (Xiang et al. 2005), for example:

$$(p_1 \wedge p_2) \vee (p_3 \wedge p_4). \tag{3.4}$$

These might be associated with a logical transition to a system behaviour p_i or its failure $\neg p_i$. For the rest of the thesis, the negated proposition is also defined as a fault event,

$$a_i \overset{\text{def}}{=} \neg p_i : \text{The Action } A_i \text{ failed to complete execution,}$$

$$f_i \overset{\text{def}}{=} \neg c_i : \text{The Facility } F_i \text{ is not available.} \tag{3.5}$$

Correspondingly, the proposition, a_s, is reserved for overall system failure. In the language of Fault Tree, this will be regarded as the top event within the scope of this thesis.

3.3.2 Mathematics on Fault Tree Model

FTA offers quantitative analysis for analysing the safety and reliability of a system. The quantitative analysis as discussed by (Nieuwhof 1975) includes probability evaluation and failure rate evaluation. Both evaluations are analytical techniques. The probability evaluation is used to calculate the probability of occurrence of each combination of events that can cause the top event. The failure rate evaluation is used to calculate the failure rate over some interval to further calculate system reliability. In this thesis, since the model transformation concerns with success and failure of events, probability theory is discussed in further detail.

The quantitative analysis of Fault Tree is supported by employing Boolean algebra for describing logical fault of a system in numeric relations. In Fault Tree model, the logic function performed by Boolean logic gate is employed in the construction of Fault Tree to describe the relationships between events, i.e. fault events. From the bottom to the top of a Fault Tree structure, commonly, events are tied together as the inputs to the Boolean logic gate that lead to an output event. The two most basic Boolean logic gates commonly used in a conventional Fault Tree are the AND-gate and the OR-gate.

This subchapter is divided into two parts. In the first part, probability theory on failure of events (fault events) in a Fault Tree is discussed. For the quantitative analysis of Fault Tree, system safety is determined by calculation of probability of failure based on constructed fault events in the Fault Tree. In the second part, the two most common logic gates, AND-gate and OR-gate, that describe the relationship between the fault events (input and output events) in a Fault Tree are discussed. The gates that derived from Boolean symbols are related to the set of operation of Boolean logic which use to calculate the probability of failure of the fault events.

Probability Theory

The mathematical description of undesired events in Fault Tree is performed based on probability theory (Vesely et al. 1981). Probability is treated as a measure taking values between 0 and 1 of a probability space of a specific situation or an experiment. In the probability space, a set of all possible outcomes is called a sample space and any specified subset of these outcomes is called an event. Probability assigned to each event is the measure of likelihood the event will occur.

In this thesis, every event has probability of success, P_s, and probability of failure, P_f, of its occurrence. Therefore, each event has two possible outcomes that are mutually exclusive, i.e. success and failure outcome. The sum of the probabilities, i.e., success and failure, gives a value of 1 as expressed follows,

$$P_s + P_f = 1. \tag{3.6}$$

An event in a Fault Tree is an undesired event in which the probability assigned to the undesired event is failure probability. Similar to any other event, the probability of an undesired event of a Fault Tree is a non-negative real number. The failure probability of an event in Fault Tree is the value of unreliability. This value is calculated by one minus the probability of success,

$$P_f = 1 - P_s. \tag{3.7}$$

The probability of success is the probability that the event will not fail over a specified time. The probability of success can be defined based on failure rate, λ, and the specified time, t, $P_s = e^{-\lambda t}$. Although probability in Fault Tree has a relation with failure rate, but this research is focused on probability. Instead of using failure rate, the failure probability can be derived by considering following a thought experiment. This experiment is designed for two possibility situations of two actions, Action A_i and Action A_j, using a simple block diagram. In the first situation, As depicted in Figure 3.4, both actions are observed to be connected in series, i.e. Action A_i completed execution before Action A_j executes. Given the probability of the respective actions fail to execute are $P(a_i)$ and $P(a_j)$, i.e. a_i and a_j are the fault events of the respective actions.

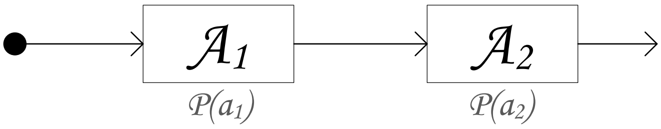


Figure 3.4: Two Actions connected in Series

A system is run for n times and the number of system failures is observed by an observer, i.e. failed to complete the intended system behaviour as modelled in UML Activity. Each action will have probability of failure and success for n times of experiment. The expected number of failure of Action A_i at the nth time of experiment is defined as,

$$nP(a_i), \tag{3.8}$$

and by using (3.6), the expected number of success of the Action A_i at the nth time of experiment can be defined as,

$$n(1-P(a_i)). \tag{3.9}$$

As the actions modelled in series, the experiment proceed to evaluate probability of the next event after Action A_i, Action A_j. At this point, one might observe the expected number of Action A_i executes in success is then proceeds in the experiment. Taking the expected number of success of the Action A_i in (3.9) as the number of experiment that going through Action A_j, the expected number of failure of failure of Action A_j can be defined as,

$$n(1-P(a_i))P(a_j). \tag{3.10}$$

Up to this point, the probability of failure of the series actions, from Action A_i to after Action A_j can be calculated as,

$$Probability\ of\ failure\ of\ Action\ A_i and\ Action\ A_j\ in\ series$$
$$= \frac{nP(a_i) + n\big(1-P(a_i)\big)P\big(a_j\big)}{n}. \tag{3.11}$$

The sum of probability of failure of Action A_i and Action A_j in the n times of experiment has to be divided with the total number of experiment, n, to obtain probability value.

In the second situation, again, the two actions are used and modelled in parallel as depicted in Figure 3.5. The probability of actions in parallel could also be calculated by taking probability of failure of Action A_i and Action A_j of n times of experiments as $nP(a_i)$ and $nP(a_j)$ respectively.

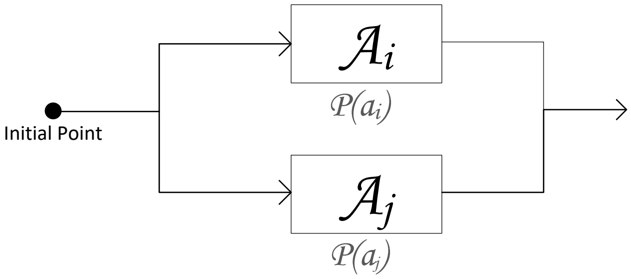


Figure 3.5: Two Actions connected in Parallel

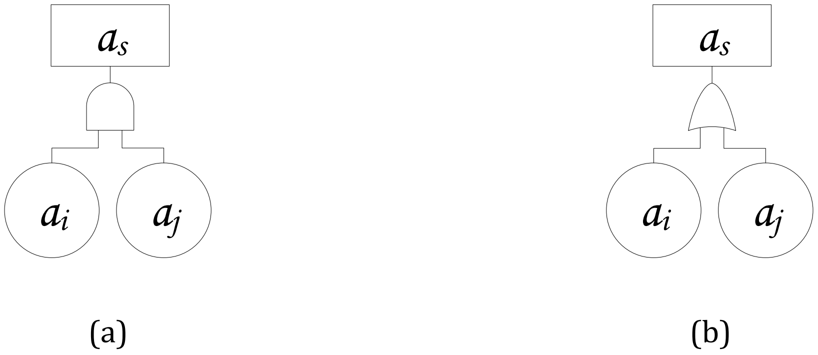
In this case, the probability of failure after the execution of both actions that occurs at the same time is defined as,

$$Probability\ of\ failure\ of\ Action\ A_i\ and\ Action\ A_j\ in\ parallel\ = \frac{nP(a_i)P(a_j)}{n}. \tag{3.12}$$

The experiment can be carried on to calculate required actions, i.e. more than two actions. With the technology and computerised system, the probability of obtaining outcome fault event for the whole Fault Tree can be calculated using computer programs (Purba et al. 2015).

Boolean Logic Gates

As mentioned earlier, the construction of Fault Trees is consisted of Boolean logic gates. The most commonly used Boolean logic gates in a Fault Tree are AND-gates and OR-gates as depicted in Figure 3.6. The Boolean logic in Fault Tree does not only calculate truth value, {True,False} or {1,0}, of an event but the Boolean logic is associated in the probability of failure value of the event.



(a) (b)

Figure 3.6: Elementary Structure of Fault Tree with (a) AND-gate, and (b) OR-gate

By referring Fault Tree elements in Figure 3.6, at the top of each figure, a_s is referred as the overall system failure. It is an output fault event from two input fault events, a_i and a_j. These two input fault events are tied together with Boolean logic gate, i.e. an AND-gate as in (a), and an OR-gate as in (b). The AND-gate represents the two input fault events, a_i and a_j, both have to occur to lead to the output fault event, a_s. The probability of the occurrence of a_s is defined as in (3.12). The equation can be simplified as,

$$P(a_s) = P(a_i)P(a_j). \tag{3.13}$$

The attachment of two input fault events to an OR-gate represents the occurrence of any or both of the input fault events can lead the output fault event. This can be referred to the derived probability in (3.11) and simplified as,

$$P(a_s) = P(a_i) + P(a_j) - P(a_i)P(a_j). \tag{3.14}$$

The subtraction of the product of the probability of the input fault events is considered as to eliminate the overlap calculation of the probability. As the failure probability in practical systems is rather very small, e.g. less than 0.01%, the product of the probability yields an even smaller value that is often negligible. This gives approximation of the sum of the inputs fault events to,

$$P(a_s) \approx P(a_i) + P(a_j), \tag{3.15}$$

which is often a conservative estimation for the output fault event.

3.4 Traffic Management System of Systems Case Study

The transformation methods and overarching metamodels that will be developed in Chapter 4 and Chapter 5 are evaluated through an application to Ramp Meter System (RMS) studied in (Ingram et al. 2014). The RMS case study is extracted from Traffic Management System of Systems (TMSoS) case study which was presented by (Ingram et al. 2014) at the IEEE SoSE in 2014 conference. The TMSoS case study that studies the inter-urban road network in the Netherlands is supplied by West Consulting. Fault Modelling Architecture Framework (FMAF) is proposed in the paper to provide a systematic approach to capture fault tolerance aspects of SoS. The FMAF study was done by collaborations between a research group from Newcastle University and West Consulting. The study discovered that quality tools and methods are needed to support reasoning of SoS fault and fault-tolerant design at the architectural level. The reasoning of SoS fault and fault-tolerant design at the architectural level is required for fault recovery strategies which lead to affect SoS service quality. Extended FMAF views in SysML were produced to support reasoning of degraded level of SoS service (which faults contribute) and the representation of failures. The fault tolerant functions of RMS situated on the access inter-urban highway was described using FMAF.

By using the Fault Tolerant Structure View (FTSV), redundancy of similar services and effects from CSs were identified. From the identified redundancy, architectural engineers are permitted to reason faults/failures at the SoS level by the negative and positive influences of CSs towards the SoS goal. Exactly five SoS-level faults were cited that could arise within the RMS. The faults were then stored in a Fault/Error/Failure View Diagram to allow the relevant CSs failures to be stated.

3.4.1 System Models: Ramp Meter System

The RMS is a CS of a TMSoS. The RMS is situated on the access ramp used to access inter-urban highways. An informal presentation of an RMS is presented in Figure 3.7.

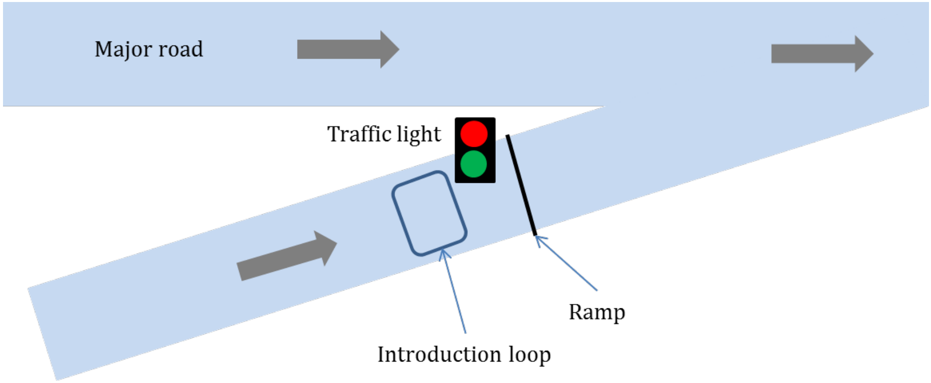


Figure 3.7: Informal Layout of a Single Ramp Meter System (Ingram et al. 2014)

The RMS employs two-phase (red and green) traffic lights to control the rate at which vehicles join the highway. The RMS has access to data about traffic in its own immediate vicinity as opposed to the Traffic Control Centre (TCC) which has access to region wide traffic data. To model the behaviour of the RMS, an operational requirement Use Case as known as high level Use Case as depicted in Figure 3.8, is firstly constructed to determine the stated purpose of RMS. The stated purpose of RMS is to control traffic flow on major road and slip road. The RMS can operate in three different modes to control the traffic light. These modes are: (i) Fixed-time Mode with fixed length of red/green phases; (ii) Responsive Mode in which a varying phase is adopted to respond to logical traffic; and (iii) Collaborative Mode in which the control strategy is provided by the TCC.

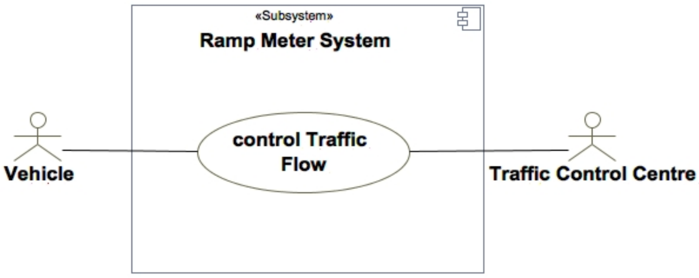


Figure 3.8: High Level Use Case Model of the Ramp Meter System

The concept of operation of the RMS is slightly modified to allow both concurrent control flows and alternative control flows to be presented in the Activity model, and to avoid control loops that are not within the scope of the research. The Activity model will be transformed into a Fault Tree and its structure will be preserved in the tree. The loops are used for specifying iteration of control flows in Activity model and are not compatible with Fault Tree. A Fault Tree has top-down of construction without iterative or 'loop' structure. The Fault Tree structure allows bottom-up calculation to measure the probability of a system to fail as top event from the root causes defined by basic events through intermediate events. The suggestion of using loops in the Fault Tree construction will allow top event or intermediate events cause lower fault event to happen which break down the tree structure. In addition, the calculation with Boolean algebra would not work, as the calculation would add up infinitely as loop continues. The avoidance os using control loops is to find the solution by applying the developed semantic mapping rules in Chapter 4. However, loops are not completely being ignored in the safety analysis method as loops are convenient for the state transition which are allowed by using

Markov Analysis. The compatibility between loops and Markov Analysis could be the subject of the future work as to harmonise the use of loops as control flows in modelling Activity Diagram.

The RMS begins each traffic light control cycle with collecting local data (vehicle flow rate). Then, the RMS starts to analyse the data, and concurrently, send the data to the TCC. Based on the data received, the TCC will decide whether to instruct the RMS to operate in Collaborative Mode. The analysis result by the RMS and the instruction from the TCC together form the basis on which operational mode, i.e. Fixed-time Mode, Responsive Mode, and Collaborative Mode, to be selected. The decision logic for mode selection follows that TCC instruction has higher priority over RMS analysis result (note that this logic will not be modelled explicitly as they are treated as a detailed design at lower level). The RMS will then implement the mode being selected. The successful execution of a control cycle will lead to the traffic light to operate in a pre-defined mode. The execution of the next control cycle may or may not change the mode being previously operated.

The following requirements of TMSoS, including RMS and TCC, derived from (Ingram et al. 2014) are grounded for system models development to execute model transformation:

1. The RMS is *situated on the access ramp* of a slip road. The ramp is used to access inter-urban highways.

2. **Two-phase (red and green) traffic lights** are employed by RMS to **control the rate** at which vehicles join the highway.

3. At the time when there are too many vehicles join a major road, the *bottlenecks being formed can be prevented* by RMS. *Vehicle distribution can be improved* by breaking up platoons, and *accidents that are caused by high speed can be reduced*.

4. Typically, *data about traffic in immediate vicinity can be accessed* by RMS. The function is opposed to region wide traffic data can be accessed by TCC.

5. One of several modes follows is operated by RMS:

   5.1 A *fixed time mode*, with fixed length red or green phases;

   5.2 an *adaptive mode*, which responds to current traffic conditions;

   5.3 a *collaborative mode*, where RMS decision is overridden by <u>TCC</u>.

6. **Local data might be gathered** by RMS and **decision in isolation will be made** by the RMS in isolation. Otherwise, an instruction from <u>TCC</u> is received to override the RMS.

7. The **data of local area is regularly collected and analysed** by the RMS. A **responsive mode of the adaptive mode will be selected** when necessary to RMS operates in making ramp-metering decisions using local data only.

In the narrative above, RMS is identified as the system of interest. The actors of the system of interest are <u>underlined</u>, the functional requirements of the RMS are in **bold**, and the non-functional requirements of the RMS are in *italic*. A lower level Use Case Diagram, as depicted in Figure 3.9, is constructed to capture all of the system functions given in the concept of operation (CoO).

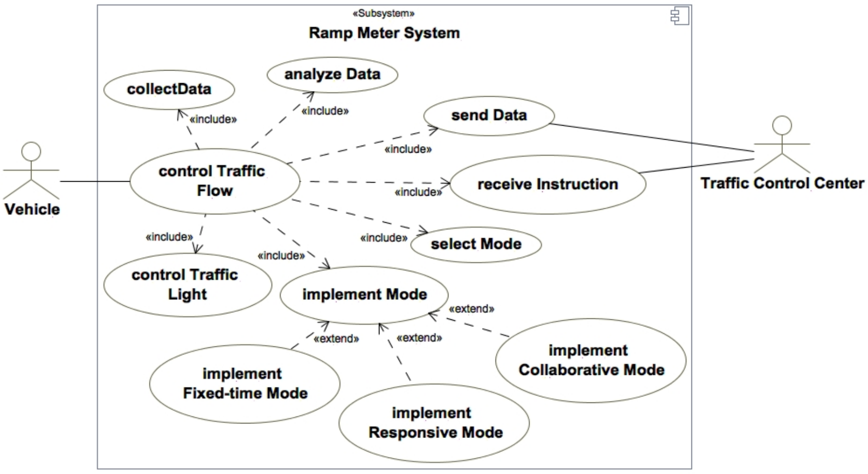


Figure 3.9: Use Case model of the Ramp Meter System

The Use Cases is then expressed in a table as a list of activities with conditions and extension points as in Table 3.3. The Use Case is elaborated by adding detail of interaction between the RMS and Actors. Then, the functions are ordered into control flows based on the details of the CoO.

Table 3.3: Use Case Description of Ramp Meter System

Use Case name	controls Traffic Flow
Description	RMS control vehicles flow using one of the operation modes at a time.
Actors	Vehicle (including Pedestrian) and TCC
Pre-conditions	Vehicles that are approaching ramp follow the traffic light signal controlled by RMS.
Post-conditions	Vehicles leave the ramp according to the traffic light signal and the RMS operates according to the selected mode.
List of activities for basic flow	1. RMS collects data of vehicles at the ramp. 2. RMS analyses the data. 3. RMS sends the data to TCC. 4. TCC sends instruction to RMS. 5. RMS selects one of the three modes. 6. RMS controls the traffic light according to the mode operation.
Alternative flow	5a. RMS implements Fixed-time Mode. 5b. RMS implements Responsive Mode of Adaptive Mode. 5c. RMS implements Collaborative Mode of Adaptive Mode.

The corresponding Activity model is presented in Figure 3.10. In the Activity model, a system function (action) is modelled by a verb-noun phrase with the first letters of the nouns capitalised. In addition, a proposition denoted by p_i is used to model the completion of the execution of the corresponding system action, A_i (note that i is integer). The RMS Activity model is started with "collect Data" (A_1) action and flows into a pair of fork node and join node. In between the fork and join node, two concurrent series of actions are modelled. One series on the left is modelled with a single action, "analyse Data" (A_2); and the other series on the right is started with "send data" (A_3) action that leads to an external function, "send Instruction" (A_4), owned by the TCC. And then, the series flows back to the RMS where RMS shall "receive Instruction" (A_5). Immediately after the join node, the control flow is continued with a flow into “select Mode” (A_6), where the logic of mode selection shall be modelled at the next level of detail. Once a decision is made, the control flow continues into a pair of decision and merge nodes in which three alternatives paths are modelled. Each path involves an action of a designated mode implementation, i.e. “implement Fixed-time Mode” (A_7) with “Fixed-time Mode selected” (c_7), “implement Responsive Mode” (A_8) with “Responsive Mode selected” (c_8), and “implement Collaborative Mode” (A_9) with “Collaborative Mode selected” (c_9). With only one out of the three modes being implemented at any one time, the RMS flow cycle then is completed with “control Traffic Light” (A_{10}) according to the mode implemented.

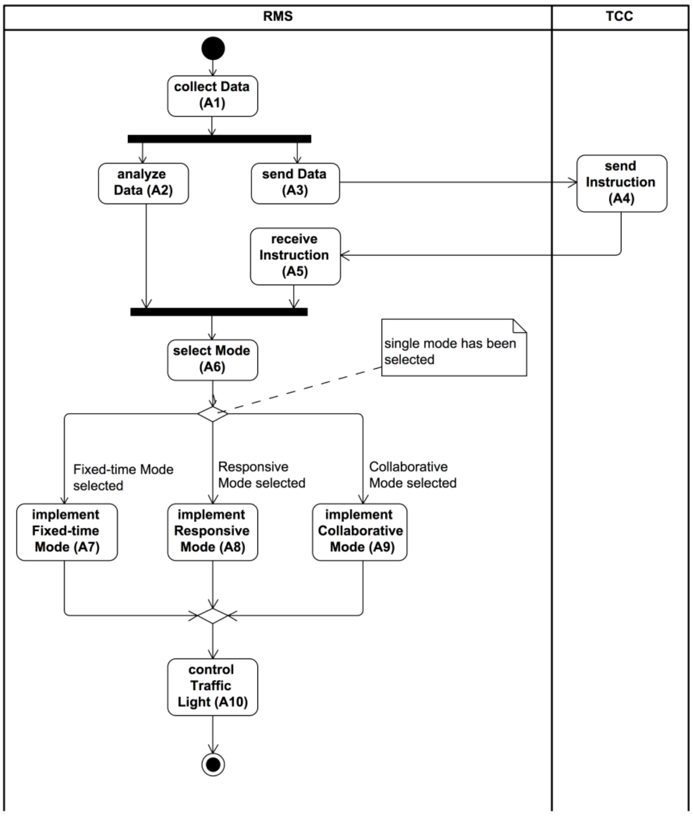

RMS
TCC
collect Data (A1)
analyze Data (A2)
send Data (A3)
send Instruction (A4)
receive Instruction (A5)
select Mode (A6)
single mode has been selected
Fixed-time Mode selected
Responsive Mode selected
Collaborative Mode selected
implement Fixed-time Mode (A7)
implement Responsive Mode (A8)
implement Collaborative Mode (A9)
control Traffic Light (A10)

Figure 3.10: The Activity Model of the Ramp Meter System

The structure of CSs of TMSoS is modelled by using UML Class as depicted in Figure 3.11. In this thesis, system component and system are referred as facility. In the

Class model, a Class is modelled by the name of the facility with the first letter of each word capitalised. Each presentation of Class is embedded with "attributes" and "operations". In addition, a proposition denoted by c_i is used to model the availability of the corresponding facility, F_i (note that i is integer). From the technical description of TMSoS, 13 Classes are identified to model as facilities of the TMSoS. From the 13 Classes, nine of them are modelled for RMS and the rest for the associate systems; TCC and vehicle. Three types of relationships that include Association, Composition, and Generalisation, are modelled to structure the relationships of the Classes. The first modelled Class, "Ramp Meter System Control Unit" (F_1), associates with five other Classes that modelled for RMS, "Data Collector" (F_2), "Data Analyser" (F_3), "Data Transceiver" (F_4), "Mode Operator" (F_7), and "Traffic Light" (F_8), and an associate system, "Traffic Control Centre" (F_{10}). In the Class model, the RMS control unit is the centre of RMS which has most relation to other facilities. The "Data Collector" (F_2) and "Data Analyser" (F_3) has Association relationship between each other. The local traffic data is collected and gathered by data collector before store the data. The stored data is mined by data analyser to analyse, evaluate, and measure before it pass to RMS control unit. Based on the passed local traffic data, the RMS control unit gives instruction to mode operator to implement one of the mode. The local traffic data is also sent to the TCC by data transmitter. The "Data Transmitter" (F_5) together with "Data Receiver" (F_6) are the part Classes that have Composition relationship with "Data Transceiver" (F_4). This means the data transmitter and data receiver are included in the data transceiver. The relation between the associate systems, "Motorway Vehicle" (F_{13}) and "Traffic Control Centre" (F_{10}), is modelled with Association. The TCC collects regional traffic data of vehicle on motorway and analyses both local and regional traffic data. When it is necessary, the TCC will transmit instruction to override RMS control unit's instruction. The override instruction is received by data receiver of RMS. This means the RMS has to give immediate instruction to the mode operator. The "Traffic Light" (F_8) and "Induction Loop" (F_9) have Association relationship with the same Class that modelled for associate system, "Ramp Vehicle" (F_{12}). The traffic light switches to red and green as an indication to vehicle at ramp which has relative position and speed either to accelerate, cruise, or stop at the end of slip road before entering the motorway. The relative position of the vehicle is detected by induction loop as a local traffic data.

The vehicle in the TMSoS is generalised into two; vehicle on ramp and vehicle on motorway. The “Ramp Vehicle” (F_{12}) and “Motorway Vehicle” (F_{13}) in which the child Classes of “Vehicle” (F_{11}) are modelled with Generalisation relationship. The vehicle can accelerate, cruise, and stop which change its speed and give its relative position.

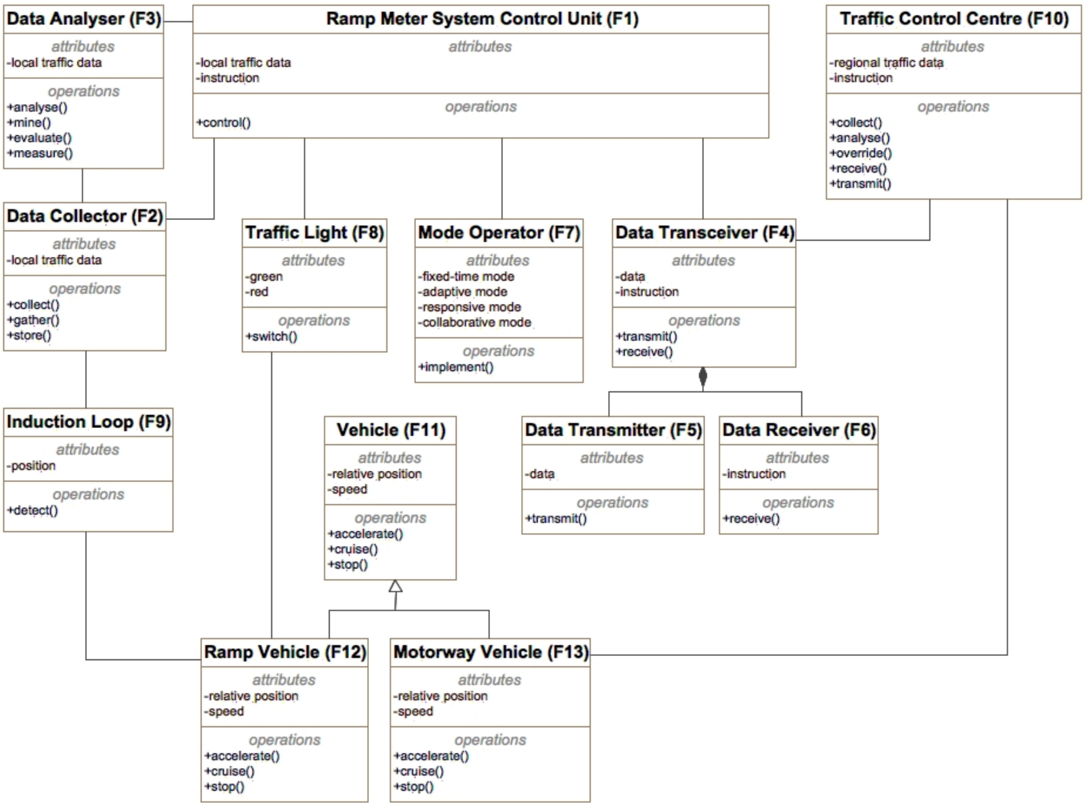


Figure 3.11: The Class Model of Ramp Meter System

3.4.2 Fault Tree: Ramp Meter System

In the original study of the RMS by (Ingram et al. 2014), five faults of the RMS have been identified that could lead to non-optimal traffic flow. The faults are:

1. Lights stuck on green or no lights at all,
2. RMS fails to adopt Collaborative Mode when instructed,

3. Lights stuck on red,

4. RMS fails to exit Collaborative Mode when instructed,

5. RMS calculates an incorrect rate for vehicles to be admitted.

The Fault 1 and Fault 3 can be traced to the faulty of traffic light. These faults (Fault 1 and Fault 3) can be classified as component-based (e.g. traffic light). Whilst the rest of the faults (Fault 2, Fault 4, and Fault 5) can be classified as functional-based. These faults (Fault 2, Fault 4, and Fault 5) are traced back to the operation faults of the RMS.

3.5 Summary of Foundational Research Knowledge

The foundational knowledge of a technical basis provided for the development of the formal transformation methods later in Chapter 4 and Chapter 5 is divided into two segments: (i) model-based approach, and (ii) mathematical representation. As the aim of this research is the transformation methods for generating Fault Tree from UML system models, the UML Specification and ARP 4761 are used as the basis of the model-based approach for the representation of system and occurrence of the system failure. The system is represented in functional and structural viewpoints by using UML Activity and UML Class respectively.

The development of reduced metamodels consider the connection between the respective models that include Activity, Class, and Fault Tree for the formal transformation. However, the reduced metamodels do not provide complete metaclasses which can limit the capability of transforming other structure of the system models into Fault Trees. For the future work, a modification of respective metamodels is required for considering additional details of the original UML metaclasses. Furthermore, an extension to the existing methods and rules is needed along the inclusion of the additional metaclasses in the formal transformation from UML diagrams into Fault Tree. For instance, the application of Predicate Calculus allows complex element of presentation. The extension can be seen as a significance value to the formal transformation methods.

The metamodel of the respective UML diagrams and the metamodel of Fault Tree developed will be used in the development of overarching metamodel later in in Chapter 4 and Chapter 5.

Despite of the model-based approach, for UML system models which has to be seen as a less formal graphical language, propositional calculus is used to present the system models in mathematical-based. The use of negation is also discussed to implement in the representation of fault of the system models. For Fault Tree, mathematical basis in the Fault Tree representation adheres from probability theory and Boolean logic gates. The failure probability of an event that represents by fault event in the Fault Tree is explained. In the Fault Tree, the Boolean logic gates that used to connect fault events express the mathematical operation in the probability calculation.

The presented TMSoS in the Subchapter 3.4 is an authorised system operated in Netherland. The presented TMSoS is used as the case study for the formal transformation methods in Chapter 4 and Chapter 5. The system functions and system components of the RMS, the focused system of TMSoS, are modelled by using behavioural and structural diagrams of the UML. They are modelled in UML Activity and UML Class respectively. Based on the faults identification from the reference study, the five faults can be classified into functional and structural faults of the RMS. A weak Fault Tree can be mistakenly developed with the faults being basic events and functional and structural faults as the intermediate events of the ‘Non-optimal traffic flow’ top event. However, the faults can be used as reference to a Fault Tree development using systematic techniques such as transformation of models.

CHAPTER 4

A FORMAL TRANSFORMATION METHOD FOR AUTOMATED FAULT TREE GENERATION FROM SINGLE UML SYSTEM MODEL

4.1 Introduction

In this chapter, methods of formal transformation from UML system models to Fault Trees generation are developed. The development of the formal transformation methods are separated into two parts. In the first part, the development of the formal transformation method is based on system behaviour that modelled in UML Activity. The logical model of control flows in the UML Activity is defined based on the propositional calculus and material implication established in the foundational research knowledge in Chapter 3. A new concept of Fault Propagation Chain (FPC) that formed based on contraposition and probability theory for fault viewpoint is introduced. Thus, a semantic mapping rules is structured. The application of the transformation method generates functional Fault Tree which defines system failure as the top event. In the second part, the development of the formal transformation method is based on system component that modelled in UML Class. In the UML Class, by implementing propositional calculus, the relationships between Classes are explored to define the structure of the Classes. Similar to the former part, the contraposition is implemented for the structure in the fault viewpoint. The contraposition is applied to material equivalence that structured based on the correspond Classes. Based on the formal transformation method, a component Fault Tree is generated. An overarching metamodel of each formal transformation method is also developed to bridge the correspond metamodels and the transformation method in the abstraction level. The formal transformation methods are applied to Traffic Management System to demonstrate the formal transformation from the UML system models to Fault Trees generation that preserve the relational structure of the UML system models into the generated Fault Trees.

The remainders of this chapter are structured as follows:

Subchapter 4.2: From Activities to Fault Tree

This subchapter is concerned with developing a formal transformation method that maps control flows modelled in UML Activities to semantically equivalent Fault Trees. The use of propositional calculus and probability theory are featured in the developed transformation method. The propositional calculus and together with material implication are used to define logical model of Activities elements. Contraposition is applied to the logical model to develop another logical model in fault viewpoints. Fault Propagation Chains are introduced to facilitate the transformation method by cooperating the faults flows. Furthermore, probability of the functional faults of the system can be analysed and Fault Tree can be formed. An overarching metamodel comprised of transformations between models is developed. The formal transformation method is applied to an understood RMS Case Study to demonstrate the approach.

Subchapter 4.3: From Classes to Fault Tree

A development of a formal transformation method from Classes to Fault Tree is discussed in this subchapter. Prior to the formal transformation method development, a logic representation of main relationship types between Classes is presented. Propositional calculus and material equivalence are applied to form logical model of the Classes. For the fault viewpoint, similar to the previous method, contrapositive is applied to the logical model. Probability analysis is applied to the fault viewpoint which lead to component Fault Tree transformation. An overarching metamodel comprised of transformations between models is developed. The formal transformation method is applied to an understood RMS Case Study to demonstrate the approach.

Subchapter 4.4: Summary

The development of the formal transformation methods from single UML model; Activity and Class, to generate two different Fault Trees; functional and component, and two respective overarching metamodels are summaries.

4.2 From Activities to Fault Tree

In this subchapter, based on stated behavioural proposition in Subchapter 3.3, logical formalisation of UML Activities is developed to logically define activities flow by using material implication. The logical formalisation of UML Activities is used as the basis to develop a set of semantic mapping rules to transform system behaviour as represented by Activity models to behavioural faults. Fault Propagation Chain (FPC) is introduced to represent the transformed behavioural faults. Probability theory (Weiss 2006) is then applied to further transform the FPCs into a conventional Fault Tree representation. This subchapter is started with formalisation of control flows as implication chains that hold relational structure of system behaviour as represented by Activity models. This subchapter will be ended with demonstration of the approach by applying the transformation method to the authorised TMSoS case study presented in Subchapter 3.4.

The application of the method to broader classes of engineering problems can be understood in two ways. The first includes complicated problems (i.e. with similar form but greater detail than the elementary forms and case study) that can be reduced to conjunctive structures. This will simply result in longer Conjunctive Material Form (CMF), such as in (4.2.3), or longer conjunctions than the elementary form in (4.2.7). On the other hand, complexity can arise from combinations of disjunctive forms with conjunctive and material forms that do not necessarily admit reduction into a single chain. Complexity is addressed at the end of this subchapter by means of transforming a nested control structure into a Fault Tree.

4.2.1 Control Flows as Implication Chains

A control flow models a sequence of actions in an Activity model. Two adjacent actions, A_i and A_{i+1}, in a control flow can then described as A_i proceeds to A_{i+1}. Assuming that there are no other actions preceding A_{i+1}, one can infer that if A_{i+1} has completed its execution, then the action A_i must have also completed its execution. Using the

The Subchapter 4.2 has been published for the publication in IEEE Transactions on Reliability, volume 67, issue 3.

propositions introduced in (3.1), the above statement is logically expressed by a material implication,

$$p_{i+1} \rightarrow p_i. \tag{4.2.1}$$

Control flows involving concurrent and alternative flows can then be expressed by conjunctions and disjunctions of these implications to form a Logical Model, e.g. the control flows depicted in the Activity model in Figure 3.10 of Subchapter 3.4. In the following subchapter, this thesis shows how to transform these implications to describe fault structures.

4.2.1 Fault Propagation Chains

To transform a Logical Model into behavioural fault representation, instead of using the negation of individual action proposition, e.g. $\neg p_i$, the contrapositive is applied to the material implication in (4.2.1). Incorporating the definition of a behavioural fault event given in (3.5), the contrapositive form of $p_{i+1} \rightarrow p_i$ is defined as,

$$a_i \rightarrow a_{i+1} \overset{\text{def}}{=} \neg p_i \rightarrow \neg p_{i+1}\,. \tag{4.2.2}$$

This expression can be interpreted as follows: if the Action A_i failed to complete execution, then action A_{i+1} will fail to execute. This statement also makes sense from the behavioural viewpoint described in an Activity model. For instance, if A_i failed to complete execution, a token will not be generated and passed onto the next action A_{i+1}. Hence, A_{i+1} will not execute. The consequence of A_{i+1} failing to execute can be therefore understood by the concept of fault propagation. Multiple chains may be combined by a conjunction or disjunction; and this will be discussed later in detail. Some event structures may involve the use of disjunctions, for example:

$$a_i \vee a_j \rightarrow a_k\,. \tag{4.2.4}$$

By refactoring, a form using conjunctions instead, may be produced:

$$(a_i \rightarrow a_k) \wedge (a_j \rightarrow a_k). \tag{4.2.5}$$

However, note that this is not a chain because the antecedent of the second material implication is not the consequence of the first material implication.

For situations where an event structure reads,

$$a_i \wedge a_j \rightarrow a_k, \tag{4.2.6}$$

the conjunction of the multiple fault events as a contracted fault event is defined, for example, two events in conjunction is defined as

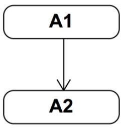
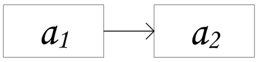
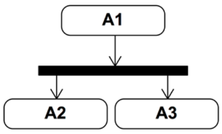
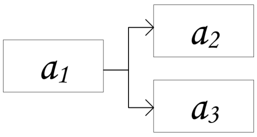
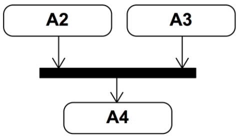
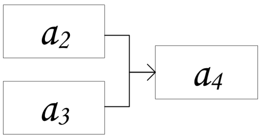
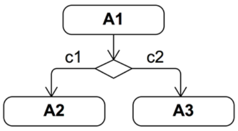
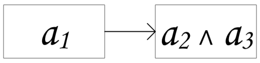
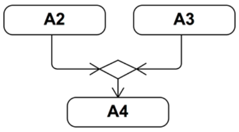
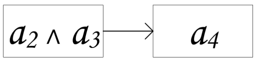
$$a_{i,j} \overset{\text{def}}{=} a_i \wedge a_j. \tag{4.2.7}$$

Logical forms used in (4.2.3), (4.2.5), (4.2.6), and (4.2.7) are defined for describing fault event structures Conjunctive Material Form (CMF). Specifically, this is a conjunction of material implications in which the two propositions within each material implication can be a conjunction. The primary reason for introducing CMF and contracted fault events is to simplify the representation of a long logic formula into simpler chains of implications. Although not all logic formulae can be written in CMF, the transformation method demonstrated later is applicable to chains that are in CMF as well as combined chains that are not in CMF. In the rest of this thesis, the graphical representation of chains of implications will be referred to as a Fault Propagation Chain (FPC).

4.2.2 Semantic Mapping Rules: Activities to Fault Propagation Chains

Based on the formalisation introduced in the previous chapter, representative types of elementary control flows seen in typical Activity model are semantically mapped into logical models, i.e. implication chains, and then transformed into FPCs by applying the contraposition of the logical models. Finally, graphical representations of the FPC are depicted in Table 4.1. Detailed discussion on the various control flow structures and their associated semantic mapping rules are provided follows:

Table 4.1: Semantic Mapping Rules

Control Flow	Logical Model	Contrapositive	Fault Propagation
 	$\begin{array}{c} p_2 \\ \downarrow \\ p_1 \end{array}$	$\begin{array}{c} \neg p_1 \\ \downarrow \\ \neg p_2 \end{array} \overset{\text{def}}{\equiv} \begin{array}{c} a_1 \\ \downarrow \\ a_2 \end{array}$	
		(a)	
 	$\begin{array}{c} p_2 \vee p_3 \\ \downarrow \\ p_1 \end{array}$	$\begin{array}{c} \neg p_1 \\ \downarrow \\ \neg p_2 \end{array} \wedge \begin{array}{c} \neg p_1 \\ \downarrow \\ \neg p_3 \end{array} \overset{\text{def}}{\equiv} \begin{array}{c} a_1 \\ \downarrow \\ a_2 \end{array} \wedge \begin{array}{c} a_1 \\ \downarrow \\ a_3 \end{array}$	
		(b)	
 	$\begin{array}{c} p_4 \\ \downarrow \\ p_2 \wedge p_3 \end{array}$	$\begin{array}{c} \neg p_2 \\ \downarrow \\ \neg p_4 \end{array} \wedge \begin{array}{c} \neg p_3 \\ \downarrow \\ \neg p_4 \end{array} \overset{\text{def}}{\equiv} \begin{array}{c} a_2 \\ \downarrow \\ a_4 \end{array} \wedge \begin{array}{c} a_3 \\ \downarrow \\ a_4 \end{array}$	
		(c)	
 	$\begin{array}{c} p_2 \vee p_3 \\ \downarrow \\ p_1 \end{array}$	$\begin{array}{c} \neg p_1 \\ \downarrow \\ \neg p_2 \wedge \neg p_3 \end{array} \overset{\text{def}}{\equiv} \begin{array}{c} a_1 \\ \downarrow \\ a_2 \wedge a_3 \end{array}$	
		(d)	
 	$\begin{array}{c} p_4 \\ \downarrow \\ p_2 \vee p_3 \end{array}$	$\begin{array}{c} \neg p_2 \wedge \neg p_3 \\ \downarrow \\ \neg p_4 \end{array} \overset{\text{def}}{\equiv} \begin{array}{c} a_2 \wedge a_3 \\ \downarrow \\ a_4 \end{array}$	
		(e)	

1. The control flow presented in (a) consists of two actions sequenced in series. As already discussed in Subchapter 3.3.1, this control flow can be modelled by using propositional logic as $p_2 \rightarrow p_1$, i.e. the completed execution of A_2 implies that the execution of A_1 must have been completed. Then, applying contraposition to the logical model, a Fault Propagation Chain that reads $a_1 \rightarrow a_2$ is derived.

2. Row (b) and (c) are concerned with the control flows that involve a pair of folk and join nodes. In (b), one action flows into two concurrent actions via a fork node while in (c), two concurrent actions flow into a single action via a join node.

The logical model for (b) can be formulated as $p_2 \vee p_3 \rightarrow p_1$, which means that the completed execution of either A_2 or A_3 implies that A_1 has been executed. Applying contraposition to the logical model leads to the expression $a_1 \rightarrow (a_2 \wedge a_3)$ or equivalently, $(a_1 \rightarrow a_2) \wedge (a_1 \rightarrow a_3)$.

The logical model in (c) can be formulated as $p_4 \rightarrow (p_2 \wedge p_3)$, which means that the completed execution of A_4 implies that both A_2 and A_3 must have been completed. Applying contrapositive to the logical model leads to the expression $(a_2 \vee a_3) \rightarrow a_4$. However, this is not in the CMF introduced in Subchapter 4.2.2. Hence, by using (4.2.5), the expression is refactored into a CMF that reads as $(a_2 \rightarrow a_4) \wedge (a_3 \rightarrow a_4)$. Immediately, one realises that how this CMF connects naturally with the previous CMF $(a_1 \rightarrow a_2) \wedge (a_1 \rightarrow a_3)$ in (b) to form two concurrent chains, $(a_1 \rightarrow a_2) \wedge (a_2 \rightarrow a_4)$ and $(a_1 \rightarrow a_3) \wedge (a_3 \rightarrow a_4)$. The FPC is depicted in the last column in (b) and (c), where a_1 bifurcates into two chains and converge back into one chain at a_4.

3. Row (d) and (e) are concerned with the control flows involving a pair of decision and merge nodes. Different from (b), one action can only flows into one of the two paths based on the decision being made. Hence, only one of the two later actions, A_2 and A_3 can be executed. Then, in (e), the two paths merge into a single path where the control sequence flows either from A_2 to A_4 or from A_3 to A_4.

Although the control flow in (d) is very different from (b), they share the same logical model $(p_2 \vee p_3) \rightarrow p_1$. This is because that no matter which path is taken, as long as there is a completed execution of either A_2 or A_3, A_1 must have been executed. Similarly, the contrapositive of the logical model can be expressed as $a_1 \rightarrow (a_2 \land a_3)$. Further using the definition of a contracted fault event introduced in (4.2.7), a FPC for (d) that reads $a_1 \rightarrow a_{2,3}$ is attained.

The control flows in (b) and (d) present execution of actions with two different control nodes. The complete execution of actions of using both control flows leads them to share the same logical model. Both have exactly the same interpretation in their logical model build up. However, the application of contraposition to the logical model of the respective control flows must reflect their semantic relational structure. The additional semantic of the logical model in (b) means both A_2 and A_3 have been executed and the logical model does not consider the merging. Whilst, in (d) the logical model means A_2 and A_3 are exclusive paths. The control flows in (b) and (d) are not captured by the propositions and the logical models reflect the formal implication. Therefore, the contrapositive structures of the two different control flows, in (b) and (d), by looking at backward implication, which are used to assist the construction of fault propagation chain, hold different expressions.

The logical model for (e) can be formulated as $p_4 \rightarrow (p_2 \lor p_3)$ (this is different from (c)), which means that the completed execution of A_4 implies that either A_2 or A_3 has been executed. Applying contraposition to the logical model leads to the expression $(a_2 \land a_3) \rightarrow a_4$. Similarly, this expression becomes $a_{2,3} \rightarrow a_4$ by defining the conjunction on the left as a contracted fault event. The two FPC in (d) and (e) also connect naturally to form a single chain, $(a_1 \rightarrow a_{2,3}) \land (a_{2,3} \rightarrow a_4)$, where $a_{2,3} \overset{\text{def}}{=} a_2 \land a_3$ is a contracted fault event as defined in (4.2.7).

The formal semantic of elementary control flow structures has been demonstrated by applying material implication and contraposition to form the semantic mapping rules. The semantic mapping rules will be used in the transformation for fault Tree generation from Activity Diagram which is comprised of the elementary control flows. A modification of Activity Diagram which is comprised of other than the elementary sontrol flows such as control loops and object flows is needed for using the semantic mapping rules.

However, the essential purpose of using control loops and object flows such as for modelling redundant systems will be discussed in Subchapter 4.3 from facilities viewpoint.

4.2.3 Fault Propagation Chains: Probability Analysis

In conventional FTA, the edges connecting lower and higher level events do not possess a well-defined semantic meaning. For instance, an edge may mean composition where a higher level event is composed by a lower event and another different lower event by an AND-gate; an edge can also mean causality where an intermediate event is caused by either one of the basic events that are connected by an OR-gate. In contrast, the directed edges in the FPC proposed in this thesis mean exactly material implications and fault propagates along these directed edges.

In the case of series (and non-exclusive), an FPC can be thought as the conjunction of a combination of non-repeating elementary unit of the form $a_i \to a_j, (i \neq j)$ where a_i can be a contracted fault event; and a disjunction in the case of exclusive chains. For every elementary unit, one can ask the question: what is the truth value of a_j given a truth value of a_i being True (or False)? The answer can be obtained from building a truth table as in Table 4.2: if a_i is True, then a_j is True; but if a_i is False, a_j can be either True or False.

Table 4.2: Truth Table of $\boldsymbol{a_i \to a_j}$

a_i	a_i	$\boldsymbol{a_i \to a_j}$ **=** $(\neg a_i \vee a_j)$
True	True	True
True	False	False
False	True	True
False	False	True

Assuming that one can define a probability for the fault event a_i being True, an observation experiment to count the number of times that event a_j has returned a True value can then be conducted. Then, one would observe that the probability of a_j denoted as $P(a_j)$, is greater than the probability of $a_i$denoted as $P(a_i)$, i.e. $P(a_j) \geq P(a_i)$. This mathematical fact, interpreted in the language of fault analysis, immediately leads to the concept of fault propagation. For instance, if the fault event, a_i has happened, one concludes automatically that a_j will happen; and if a_i had not happened, a_j may still happen.

To better describe the probability of the occurrence of these fault events, with the aid of Figure 4.1, the following definitions and notations will be adopted:

Definition 1: $P(a_i)$ is defined as the probability of the fault event, a_i, independent of the occurrence of any other fault events, i.e. prior faults propagated to a_i, if any, are not counted toward $P(a_i)$.

Definition 2: $P(a_{i+})$ is defined as the probability of observing a system failure right after the (supposed) execution of A_i. Similarly, one can also define a probability measurement, $P(a_{i-})$, as the probability of observing a system failure just before the execution of action A_i. In the simple construct shown in Figure 4.1, one notices that $P(a_{i+}) = P(a_{j-})$. Note that a_{i+} and a_{i-} are fictitious events that are not originally presented in the Activity model.

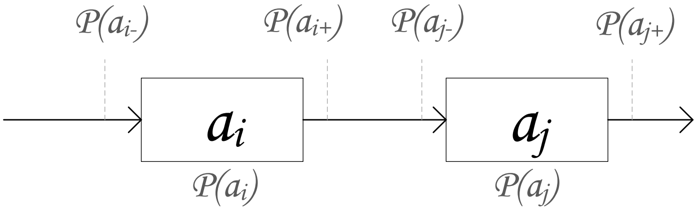


Figure 4.1: Example of Fault Propagation Chains in Series Propagation

Using the above definitions, for any given chain, $a_i \rightarrow a_j$, the following mathematical relation can be established:

$$\begin{cases} P(a_{i+}) = P(a_{i-}) + P(a_i) - P(a_{i-})P(a_i) \\ P(a_{j+}) = P(a_{i+}) + P(a_j) - P(a_{i+})P(a_j)' \end{cases} \tag{4.2.8}$$

with $P(a_{i+}) = P(a_{j-})$. To see how these relations are derived, the following thought experiment is considered (Weiss 2006). An observer runs the system for n times and observes the number of system failures, i.e. failed to complete the intended system behaviour as modelled in the UML Activity. Up to the point at where system function A_i is about to execute, the observer would measure a probability of system failure equals to $P(a_{i-})$ based on Definition 2. Then, statistically, there are $nP(a_{i-})$ number of experiments in which the system would fail at this point and $n(1 - P(a_{i-}))$ number of experiments in which the system successfully proceed to the execution of A_i. Given that there is a probability of failure in the execution of A_i, i.e. $P(a_i)$, based on Definition 1, in addition to the experiments in which the system failed before the execution of A_i, the observer would further expects a $m = n(1 - P(a_{i-}))(P(a_i))$ number of experiments in which the system would fail right after the supposed execution of A_i. Therefore, one can calculate $P(a_{i+})$ by the quotient of the total number of failed events at this point and the total number of experiments as,

$$P(a_{i+}) = \frac{nP(a_{i-}) + m}{n}$$

$$= P(a_{i-}) + (1 - P(a_{i-}))(P(a_i))$$

$$= P(a_{i-}) + P(a_i) - P(a_{i-})P(a_i). \tag{4.2.9}$$

The second relation in (4.2.8) can be derived by using exactly the same concept.

In a simplified case where a_i is assumed to be the first fault event in the entire chain, i.e. $P(a_{i-}) = 0$, the relations is reduced to:

$$P(a_{j+}) = P(a_i) + P(a_j) - P(a_i)P(a_j). \tag{4.2.10}$$

The relation in (4.2.9) and (4.2.10) will form the basis of the analysis of FPCs and the transformation of FPCs to Fault Trees.

As shown in Table 4.1, certain control flows after the transformation can lead to bifurcation and convergence in the FPC. In these cases, the probability calculation needs to be carefully dealt with to avoid over-counting of the probability of repeated events. Using the example depicted in Figure 4.2, based on the derived relations in (4.2.8) and (4.2.10), one can establish the following relations:

$$\begin{cases} P(a_{i+}) = P(a_o) + P(a_i) - P(a_o)P(a_i) \\ P(a_{j+}) = P(a_o) + P(a_j) - P(a_o)P(a_j) \\ P(a_{k-}) = P(a_{i+}) + P(a_{j+}) - P(a_{i+})P(a_{j+} | a_{i+}), \\ P(a_{k+}) = P(a_{k-}) + P(a_k) - P(a_{k-})P(a_k) \end{cases} \tag{4.2.11}$$

with a_o being an initial fault event, i.e. no other fault propagates to it. The first and second relations in (4.2.11) are derived based on the result in (4.2.10) by considering the bifurcation as two parallel (non-exclusive) chains, i.e. a_o to a_i; and a_o to a_j respectively. The fourth relation can be derived using the same thought experiment in deriving the relations in (4.2.8).

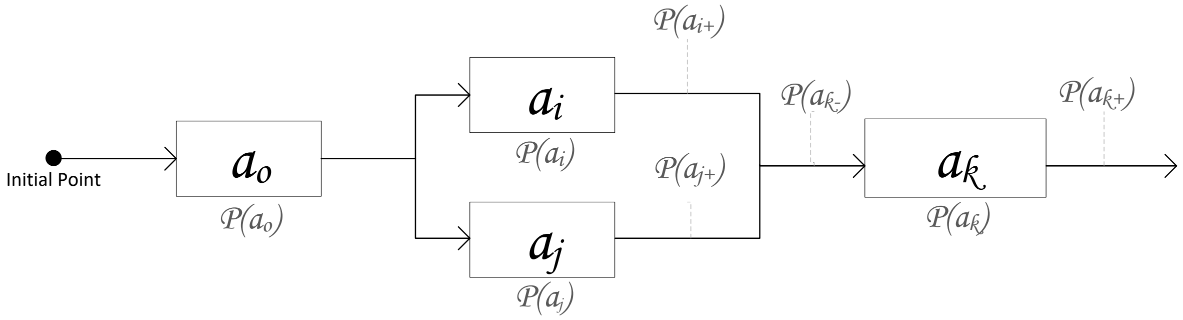


Figure 4.2: Example of Fault Propagation Chains in Bifurcated and Merged Propagation

The derivation of the third relation in (4.2.11) requires a consideration of the repeated event, a_o in the convergence of the two paths. Instead of directly using the relations derived in (4.2.8), again the thought experiment in which the observer runs the system of n times is considered. Based on previous derivations, the observer would first expect $nP(a_o)$ number of system failures due to the failed execution of A_o. The observer would also expect an additional $m = n(1 - P(a_o))(P(a_i))$ number of system failures right after the supposed execution of A_i; and an additional $q = n\big(1 - P(a_o)\big)(P\big(a_j\big))$ number of system failures right after the supposed execution of A_j. As such, the total number of system failures right before the execution of A_k is given by $nP(a_o) + m + q$. This then allows $P(a_{k-})$ to be calculated as,

$$P(a_{k-}) = \frac{nP(a_o) + m + q}{n}$$

$$= P(a_o) + P(a_i) - P(a_o)P(a_i) + P\big(a_j\big) - P(a_o)P\big(a_j\big)$$

$$= P(a_{i+}) + P(a_{j+}) - P(a_o), \tag{4.2.12}$$

where the last step has used the first two relations in (4.2.11) as substitutions. In this representation, as seen in Figure 4.2, a_o is a repeated event, and hence, can be considered as the intersection of the two events, a_{i+} and a_{j+} in which

$$P(a_{i+} \cap a_{j+}) = P(a_{i+})P(a_{j+} \mid a_{i+}) = P(a_o), \tag{4.2.13}$$

where $P\big(a_{j+}\big|a_{i+}\big)$ is the conditional probability of a_{j+} given a_{i+}. Substituting (4.2.13) into (4.2.12) would arrive the third relation in (4.2.11). It is worth noting that due to the presence of the repeated event, a_{i+} and a_{j+} can be dependent. For instance, if an observation of a_{i+} returns a True value due to the occurrence of the repeated a_o, immediately, one would expect a_{j+} may or may not return a True value depending whether a_j has occurred or not. The situation where a repeated fault event happens in a FPC is analogous to a repeated basic event seen in a conventional Fault Tree.

A contracted fault event, such as $a_{i,j}$, can be viewed identically as a normal fault event such that it follows exactly the same analysis rule. However, from both the system modelling viewpoint and fault analysis viewpoint, it can be useful to understand how individual fault event embedded in the contraction can affect the overall FPC. Tracing back to the Activity model, a contracted event embeds exclusive paths, i.e. if one path is taken, no other paths will be taken. Theoretically, it is possible to attribute each of the paths a probability for which it might be taken. The sum of the probabilities of the conditional events, such as c_i and c_j, has to be exactly one. At a time, only one conditional event contributes to the probability of the contracted event; this is to reflect that a path is always taken. Now, coming back to the fault analysis viewpoint, an embedded fault event then becomes only relevant when its corresponding path (in the Activity model) is taken. This allows one to establish the mathematical relation as follows:

$$P(a_{i,j}) = P(a_i)P(c_i) + P\left(a_j\right)P(c_j), \tag{4.2.14}$$

where $P(c_i)$ is the probability of the system taking the path, c_i. Note also that c_i and c_j are not fault events and that they are mutually exclusive. In this case, unlike conditional events, the sum of probabilities for the contracted fault event, such as in (4.2.14), is not equal one. The conditional statement, c_i, traces back to the guard contents on an activity edge in a control flow that contains a decision node (c.f. first column of Table 4.1(d)).

4.2.4 Semantic Transformations: Fault Propagation Chains to Fault Tree

The probability definition proposed for the FPCs are based on a theoretical concept where the failed execution of an action (system function), i.e. a_i, can be somehow observed experimentally. However, in reality, a failed execution of a system function may only be observed when the function is implemented, for example on hardware. In the perspectives of system behaviour and structure modelling, the execution of system function is often referred to as allocation of system functions to system components. For example, if a system function A_i, is allocated to two system components, then each individual failure rate of the two components will contribute to the calculation of the probability of functional failure, $P(a_i)$. As such, a different allocation strategy will likely result in different Fault Tree structure when physical components are considered. As this subchapter is only concerned with system functional faults, the allocation mechanism is pushed into the next-level of detail in the decomposition process of FTA, and this will be a topic of next subchapter.

Taking the FPC in Figure 4.1 as an example, with a simplified situation where a_i is the first fault event in the entire chain, i.e. $P(a_{i-}) = 0$, the FPC can be transformed into a Fault Tree that consists of an OR-gate as depicted in Figure 4.3. This is because that based on (4.2.10), $P(a_{j+})$ can be considered as a union of $P(a_i)$ and $P(a_j)$, where a_i and a_j are basic events, and a_{j+} is an intermediate event that can further contributes to the probability calculation of fault events after a_j. If there is no fault event after a_j, then the intermediate event a_{j+} can be replaced by the top event, “system failure”, a_s.

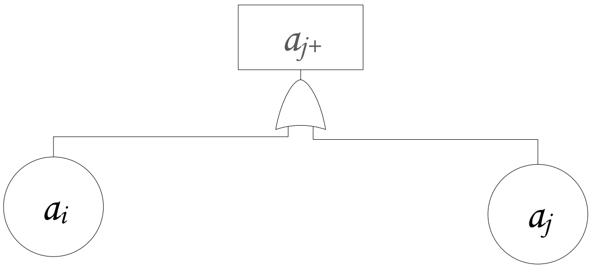


Figure 4.3: Example of Fault Tree transformed from Fault Propagation Chain in Figure 4.1

For the FPC in Figure 4.2, following the transformation developed above, the set of equations in (4.2.10) is translated exactly into Fault Tree probability calculations in the following way:

$$\begin{cases} P(a_{i+}) = P(a_o \text{ OR } a_i) \\ P(a_{j+}) = P(a_o \text{ OR } a_j) \\ P(a_{k-}) = P(a_{i+} \text{ OR } a_{j+}) \\ P(a_{k+}) = P(a_{k-} \text{ OR } a_k) \end{cases}, \qquad (4.2.15)$$

with

$$P(a_i \text{ OR } a_j) = P(a_i \cup \ a_j) = P(a_i) + P(a_j) - P(a_i \cap a_j), \qquad (4.2.16)$$

and the intersection

$$P(a_i \cap a_j) = P(a_i)P\big(a_j \big| a_i\big) = P(a_j)P\big(a_i \big| a_j\big), \tag{4.2.17}$$

if a_i and a_j are dependent events. And the intersection is reduced to

$$P(a_i \cap a_j) = P(a_i)P(a_j), \tag{4.2.18}$$

if a_i and a_j are independent events, but both are not necessarily mutually exclusive.

The new set of equation in (4.2.15) can be immediately depicted graphically in a Fault Tree, as shown in Figure 4.4, by defining a_o, a_i, a_j , and a_k as basic events (independent of each other), and $a_{i+}, a_{j+}, a_{k-},$ and a_{k+} as intermediate events.

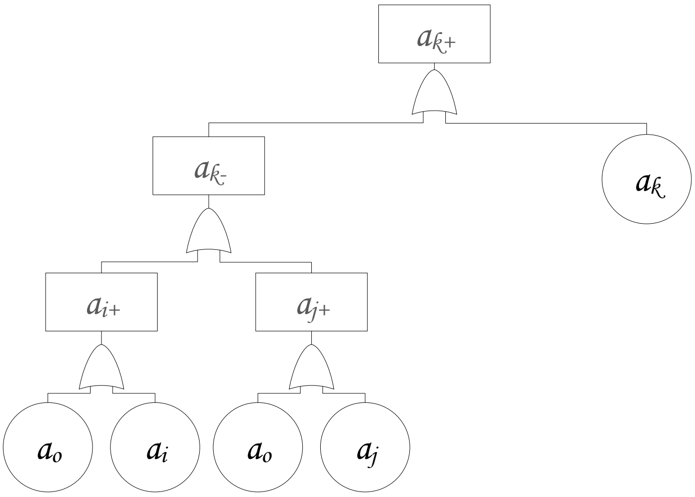


Figure 4.4: Example of Fault Tree transformed from Fault Propagation Chain in Figure 4.2

For a contracted fault event, at the contraction level, it can be treated similar to a normal fault event. Hence, the mapping to Fault Tree is straightforward. For the fault events embedded in the contraction, (4.2.14) can be translated into

$$P(a_{ij}) = P((a_i \cap c_i) \cup (a_j \cap c_j)). \tag{4.2.19}$$

The Fault Tree corresponding to (4.2.19) is depicted in Figure 4.5. Because $c_i$and c_j are not fault events, but conditional statements, instead of using the normal AND-gates, inhibit-gates are used. Moreover expanding the expression on the right hand side of the (4.2.19), an intersection term that reads $(a_i \cap c_i) \cap (a_j \cap c_j)$ is expected. However, this term yields zero as c_i and c_j are mutually exclusive events.

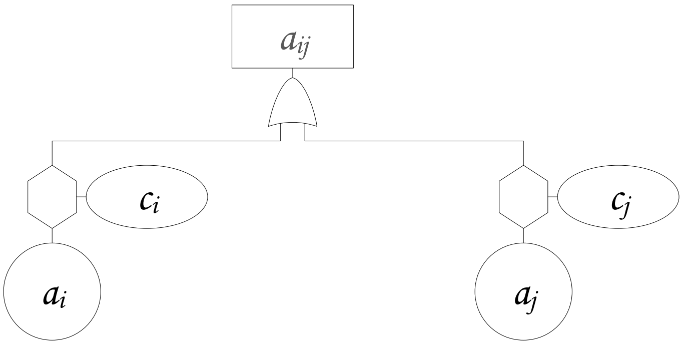


Figure 4.5: Example of Fault Tree transformed from Fault Propagation Chain that derived from an expanded Contracted Fault Event

4.2.5 Nested Control Flows

In this subchapter, the unrestricted of transformation method to simple connections of elementary control flows is demonstrated. The transformation method is also applicable to complex control flows that involve the nesting of elementary control flows. The transformation method is applied to the nested control flow shown in Figure 4.6 to obtain a semantically equivalent Fault Tree. It is worth noting that in this nested control flow, Action, A_j does not proceed to the Action, A_m.

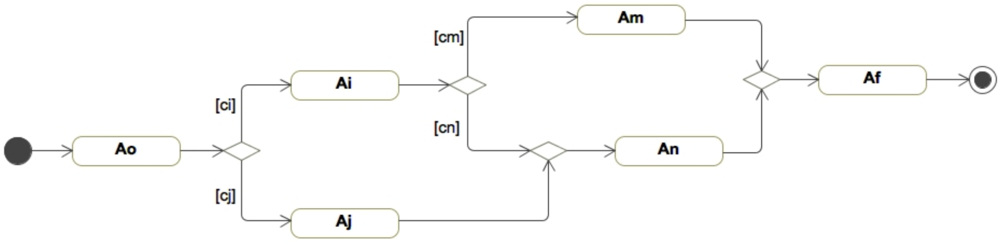


Figure 4.6: A Nested Control Flow modelled in UML Activity

Using the elementary control flows as the basis, the structure of the nested control flow can be logically expressed by the following expressions,

$$\begin{cases} (p_i \vee p_j) \rightarrow p_o \\ p_n \rightarrow (p_i \vee p_j) \\ (p_m \vee p_n) \rightarrow p_i \\ p_f \rightarrow (p_m \vee p_n) \end{cases}. \tag{4.2.20}$$

Applying contraposition to the expression in (4.2.20), the following elementary unit is attained,

$$\begin{cases} (a_o \rightarrow a_i) \wedge (a_o \rightarrow a_j) = a_o \rightarrow (a_i \wedge a_j) \\ (a_i \rightarrow a_n) \vee (a_j \rightarrow a_n) = (a_i \wedge a_j) \rightarrow a_n \\ (a_i \rightarrow a_m) \wedge (a_i \rightarrow a_n) = a_i \rightarrow (a_m \wedge a_n) \\ (a_m \rightarrow a_f) \vee (a_n \rightarrow a_f) = (a_m \wedge a_n) \rightarrow a_f \end{cases}, \tag{4.2.21a}$$

or alternatively, using the concept of contracted fault event, (4.2.21a) is arrived to

$$\begin{cases} a_o \rightarrow a_{i,j} \\ a_{i,j} \rightarrow a_n \\ a_i \rightarrow a_{m,n} \\ a_{m,n} \rightarrow a_f \end{cases}. \tag{4.2.21b}$$

Although the conjunction of the above relations,

$$(a_o \rightarrow a_{i,j}) \wedge (a_{i,j} \rightarrow a_n) \wedge (a_i \rightarrow a_{m,n}) \wedge (a_{m,n} \rightarrow a_f), \tag{4.2.22}$$

is in CMF, this is not a chain as described in (4.2.3) because it breaks at ... $a_n) \wedge (a_i$ .... To resolve this issue, only one contracted fault event, $a_{m,n}$ is introduced, such that the FPC can be expressed by a disjunction of the two relations,

$$\begin{cases} (a_o \rightarrow a_i) \wedge (a_i \rightarrow a_{m,n}) \wedge (a_{m,n} \rightarrow a_f) \\ (a_o \rightarrow a_j) \wedge (a_j \rightarrow a_n) \wedge (a_n \rightarrow a_f) \end{cases}, \tag{4.2.23}$$

in which these two chains are two mutually exclusive paths. As the relation in (4.2.23) is emphasised as a FPC, (4.2.23) is not in CMF due to having a disjunction in between the two relations.

Continuing with (4.2.23), the FPC is depicted as in Figure 4.7. In addition to the bifurcation, the necessary condition for each of the exclusive chains resulted from the disjunction of the two relations in (4.2.23) is indicated.

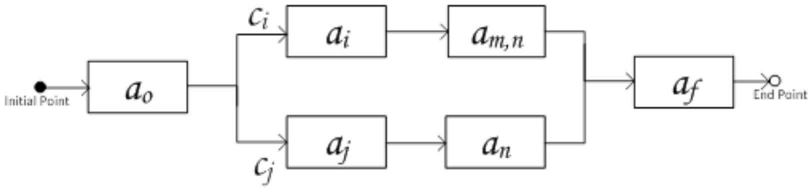


Figure 4.7: A derived FPC for the nested control flow as described in (4.2.23)

Based on the transformation method developed in the previous subchapter, the FPC is transformed into a Fault Tree that depicted in Figure 4.8, in which the detailed probability analysis for the FPC in Figure 4.7 is provided.

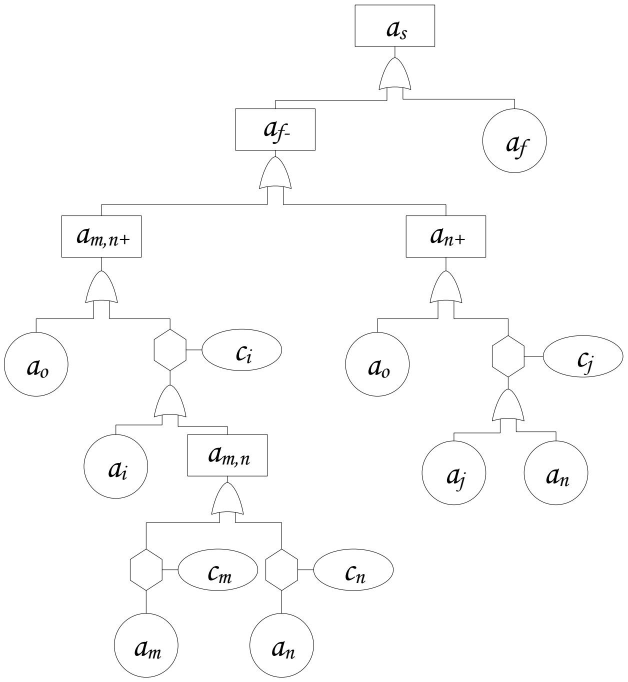


Figure 4.8: The transformed Fault Tree for the Nested Control Flow

To demonstrate that the FPC in Figure 4.7 indeed maps into the Fault Tree depicted in Figure 4.8, a detailed probability analysis for this FPC is performed. Again, the concept of running the system for n times and observing the number of system failure is used.

By running the system for n times, based on the derivation of (4.2.10), the expected number of system, failure, q_1, up to the point at a_{o+} can be determined as,

$$q_1 = nP(a_o). \tag{4.2.24}$$

The FPC then bifurcates into two chains with exclusive conditions after the fault event, a_o. The number of system success that goes through the top chain with condition, c_i, can be calculated as,

$$m = n\big(1 - P(a_o)\big)P(c_i). \tag{4.2.25}$$

Then, the additional number of system failure as a_{i+} can be calculated as,

$$q_2 = mP(a_i), \tag{4.2.26}$$

and the additional number of system failure at $a_{m,n+}$ as,

$$q_3 = m\big(1 - P(a_i)\big)P(a_{m,n}), \tag{4.2.27}$$

where $P(a_{m,n})$ is the probability of the contracted fault event, $a_{m,n}$, and can be calculated based on the formula derived in (4.2.14). Using (4.2.24) to (4.2.27), $P(a_{m,n+})$ can be calculated by

$$\begin{aligned} P(a_{m,n+}) &= \frac{q_1 + q_2 + q_3}{n} \\ &= P(a_o) + \big(1 - P(a_o)\big)P(c_i)(P(a_i) + \big(1 - P(a_i)\big)P\big(a_{m,n}\big)) \\ &= P(a_o) + \big(1 - P(a_o)\big)P(c_i)P\big(a_i \cup a_{m,n}\big) \\ &= P(a_o \cup \Big(c_i \cap \big(a_i \cup a_{m,n}\big)\Big)) \\ &= P(a_o \cup \Big(c_i \cap \Big(a_i \cup \big((a_m \cap c_m) \cup (a_n \cap c_n)\big)\Big)\Big)), \end{aligned} \tag{4.2.28}$$

where the last step has used the derived result provided in (4.2.19) for a contracted fault event.

For the bottom chain, condition c_j is required to proceed to a_j. Hence, similar to (4.2.25), the number of system success that goes through the bottom chain with condition, c_j, can be calculated as,

$$m' = n\big(1 - P(a_o)\big)P(c_j). \tag{4.2.29}$$

Then, the additional number of system failure at a_{j+}can be calculated as,

$$q_4 = m'P(a_j), \tag{4.2.30}$$

and the additional number of system failure at a_{n+} as,

$$q_5 = m'\Big(1 - P\big(a_j\big)\Big)P(a_n). \tag{4.2.31}$$

Using (4.2.24), (4.2.28) to (4.2.30), $P(a_{n+})$ can be calculated by,

$$
\begin{aligned}
P(a_{n+}) &= \frac{q_1 + q_4 + q_5}{n} \\
&= P(a_o) + (1 - P(a_o))P(c_j)\Big(P(a_j) + \big(1 - P(a_j)\big)P(a_n)\Big) \\
&= P(a_o) + (1 - P(a_o))P(c_j)P(a_i \cup a_j)
\end{aligned}
$$

$$= P(a_o \cup \big(c_j \cap (a_i \cup a_j)\big)). \tag{4.2.32}$$

For the convergence of the two chains, the third relation in (4.2.11) and in (4.2.15) are used, to obtain $P(a_{f-})$, where

$$P(a_{f-}) = P(a_{m,n+}) + P(a_{n+}) - P(a_{m,n+})P(a_{n+} \mid a_{m,n+})$$

$$= P(a_{m,n+} \cup a_{n+}), \tag{4.2.33}$$

with $P(a_{m,n+})P(a_{n+} \mid a_{m,n+})$ containing the repeated events, a_o. Finally, the derived relation in (4.2.8) is used to obtain the top event probability, $P(s)$, as,

$$P(s) = P(a_{f+}) = P(a_{f-}) + P(a_f) - P(a_{f-})P(a_f)$$

$$= P(a_{f-} \cup a_f). \tag{4.2.34}$$

In the above transformations, basic events, a_o, a_i, a_j, a_m, a_n, and a_k; conditional events, c_i, c_j, c_m, and c_n; and specifically defined intermediate events, $a_{m,n+}, a_{n+}$, and a_{f-}, are used. Representing relations in (4.2.27), (4.2.31), (4.2.32), and (4.2.33) all together gives the Fault Tree depicted in Figure 4.8. The minimal cut sets of this Fault Tree are: $\{a_o\}, \{a_i, c_i\}, \{a_j, c_j\}, \{c_i, c_m, a_m\}, \{c_i, c_n, a_n\}, \{a_n, c_j\}$, and $\{a_f\}$.

The nested control flow example demonstrates that the transformation method can be applicable to complex control flows in which the five elementary control flows as shown in Table 4.1 are used as building blocks.

4.2.6 Overarching Metamodel of the Formal Transformation from UML Activity Model to Fault Tree

This subchapter abstracts the developed rules of mapping an Activity model to a FPC and rules of transforming the FPC to a Fault Tree into an Activity model-Fault Propagation Chain-Fault Tree (AM-FPC-FT) overarching metamodel as depicted in Figure 4.9. The overarching metamodel is composed of 12 metaclasses that are differentiated by a white-grey-dark scale (white, yellow, and blue for the coloured version). On the left, three metaclasses (shown in grey), ControlFlow, Action, and ValueSpecification (for guard), are inherited from the RAM. In the middle, four metaclasses (shown in white), are introduced for the metamodeling of FPCs. In particular, the elementary structure of a FPC is in the form of fault events connected by directed edges (material implications). Therefore, similar to ControlFlow and Action in the RAM, two metaclasses, FaultEvent and MaterialImplication are introduced to reflect this structure. In addition, FaultEvent in the FPC metamodel is further specialised into two types: SingleFault and ContractFault. On the right, five metaclasses (shown in dark), ORGate, InhibitGate, OutputEvent, BasicEvent, and ConditionalEvent, are inherited from the FTM. The metaclasses, where relevant, inherit the relations established in their domain metamodel. These include the relations between the three metaclasses inherited from the RAM. The overarching metamodel is developed and depicted in a way such that it also reflects, from left to right, the transformation method in which an Activity model is mapped into a FPC and then transformed into a Fault Tree.

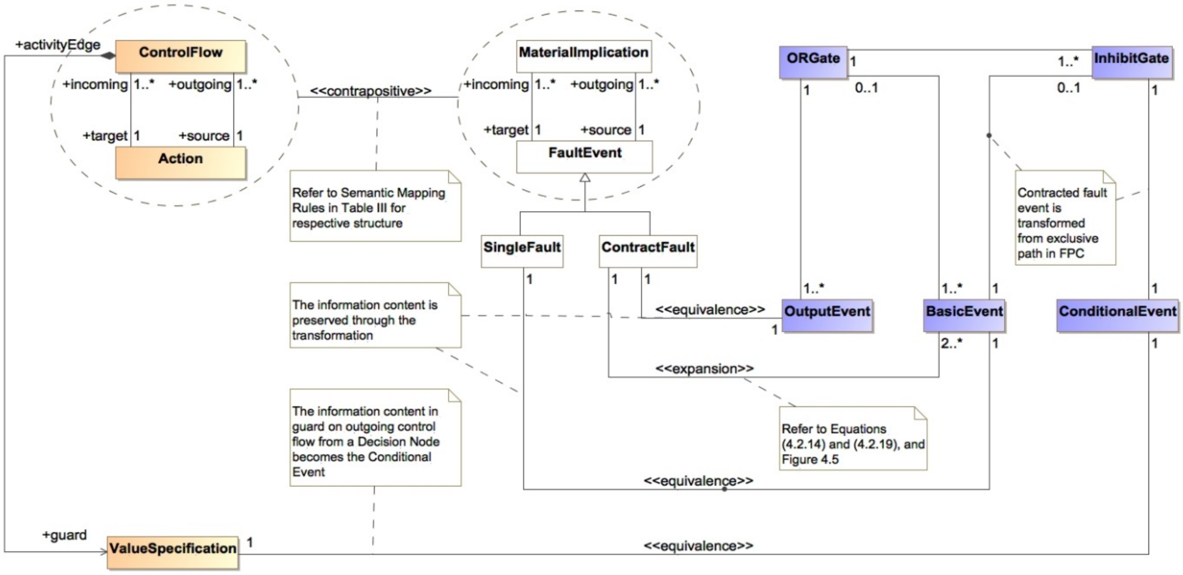


Figure 4.9: Activity model-Fault Propagation Chain-Fault Tree-Overarching Metamodel

The mapping from an Activity model to a FPC is captured by a stereotype, <<contrapositive>>, which is introduced to refer to the Semantic Mapping Rules shown in Table 4.1. As these mapping are structure-based rather than component-based, instead of associating individual elements, the ControlFlow - Action structure is connected to the MaterialImplication - FaultEvent structure (circled in dash lines) via the <<contrapositive>> relation. As different elementary control flows would give different fault propagation structure, an additional note is attached to the <<contrapositive>> relation to indicate the detailed rules as captured in Table 4.1.

Moving on with the transformation from FPCs to Fault Trees, an <<equivalence>> stereotype is defined to illustrate a relation in which an entity is mapped exactly onto another entity without the need of modifications. In particular, three transformations that satisfy an <<equivalence>> relation: (i) SingleFault is mapped exactly onto BasicEvent; (ii) ContractFault is mapped exactly onto an OutputEvent (as per intermediate event); and (iii) Guard information as captured by ValueSpecification is mapped exactly onto a ConditionalEvent. All of the above exact mappings are apparent from Figure 4.1 to Figure 4.5. In addition to the <<equivalence>> stereotype, an

<<expansion>> stereotype is introduced to capture the expansion of a contracted fault event into basic events as seen in Figure 4.5. and Figure 4.8. Since, a contracted fault event contains at least two exclusive paths, the multiplicity for the BasicEvent on the <<expansion>> relation is defined as 2…*. Additional notes are given to each of these stereotypes relations to provide detail explanations on the specific transformation mechanism.

As seen in the resultant Fault Trees, the transformations further impose a specific set of relationships between Fault Tree metaclasses. These are captured in the AM-FPC-FT overarching metamodel. Firstly, an OR-gate can be connected directly to one or more inhibit-gates as seen in Figure 4.5 and Figure 4.8 due to the expansion of the contracted fault event. This is captured by an association between the metaclasses ORGate and InhibitGate with multiplicity 1 and 1 … * on each side. Secondly, depending whether a basic event is transformed from an Action that is within an exclusive path, i.e. proceed from satisfying its guard condition, or not, the basic event is either directly connected to an OR-gate or an inhibit-gate. Again, corresponding multiplicities are defined and additional notes are provided on the associations to provide details explanation.

4.2.7 Transformation Method Application to Traffic Management System of Systems Case Study

In this subchapter, the developed transformation method and the AM-FPC-FT overarching metamodel are evaluated through an application of both to the RMS studied in (Ingram et al. 2014).

Ramp Meter System Functional Fault Trees

In the rest of this subchapter, the transformation method is applied to the Activity model in Figure 3.10 and how the generated Fault Tree can provide useful information in the identification of functional faults as well as inferring a fault structure is demonstrated. The fault structure is then used to analyse the original design model.

For convenience, a set of fault events as the negation of the propositions, $a_i \overset{\text{def}}{=} p_i$, is defined. Based on the semantic mapping rules (a), (b), and (c), the initial fault event a_1 is bifurcated into two chains, $(a_1 \rightarrow a_2) \wedge (a_1 \rightarrow a_3)$, with one chain has a single fault event, a_2, and the other chain has three fault events connected in series, $(a_3 \rightarrow a_4) \wedge (a_4 \rightarrow a_5)$. Then, the two chains are merged into a single chain that leads to fault event a_6 with $(a_2 \rightarrow a_6) \wedge (a_5 \rightarrow a_6)$. The fault event a_6 is continued into a contracted event, $a_{7,8,9} = a_7 \wedge a_8 \wedge a_9$. The contracted fault event, $a_{7,8,9}$, is further propagated to the last fault event, a_{10}, through the unitary implication $a_{7,8,9} \rightarrow a_{10}$. Grouping all of the unitary implications together, a complete FPC is generated and depicted graphically in Figure 4.10. An initial point and an end point are added to the chain to show where the fault propagation starts and ends. These points are mapped to the Initial Node and Activity Final Node in the original Activity model.

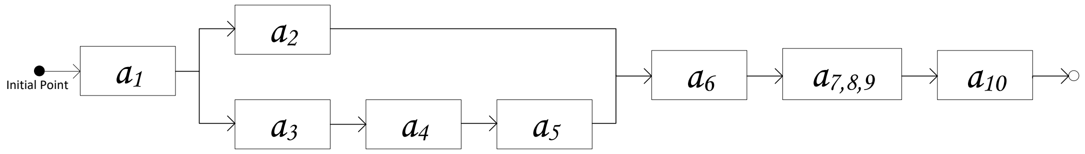


Figure 4.10: The Fault Propagation Chain Transformed from the RMS Activity Model

Next, the FPC is transformed to a Fault Tree using the established transformation method. The detailed physical architecture, i.e. allocation of functions to system component will not be considered. Without going through the detailed mathematical derivation, one can obtain the Fault Tree by: firstly, elaborating the relations in (4.2.15) for the bifurcation and convergence of the two parallel chains $(a_1 \rightarrow a_2) \wedge (a_2 \rightarrow a_6)$ and $(a_1 \rightarrow a_3) \wedge (a_3 \rightarrow a_4) \wedge (a_4 \rightarrow a_5) \wedge (a_5 \rightarrow a_6)$; and then elaborating the relation in (4.2.19) for the contracted event, $a_{7,8,9}$ for three exclusive chains; and finally using the relations in (4.2.15) again to integrate everything to obtain the final RMS Fault Tree as depicted in Figure 4.11. The top event of the RMS Fault Tree, system failure, a_s, is defined as the system failing to complete the intended system behaviour as modelled in the UML Activity.

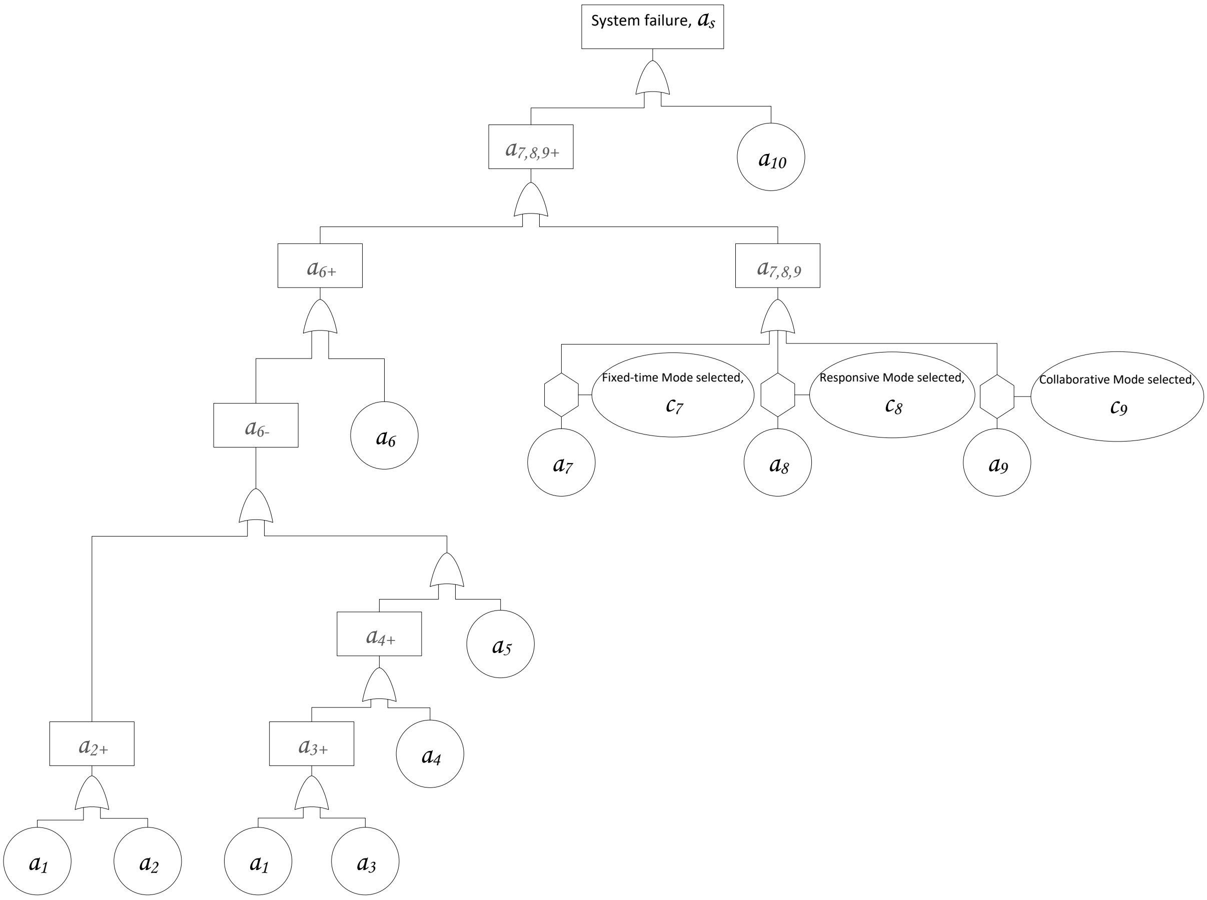


Figure 4.11: The RMS Fault Tree transformed from the Fault Propagation Chain in Figure 4.10

Qualitative Analysis of the Transformed Fault Tree

Without a designated probability for each of the basic events, it is difficult to provide a meaningful quantitative analysis. Nonetheless, based on the structure of the RMS Fault Tree, the following qualitative analyses are provided:

Firstly, every system function, as modelled by actions, becomes a basic (fault) event in the final RMS Fault Tree after the transformations. The minimal cut sets are: $\{a_1\}, \{a_2\}, \{a_3\}, \{a_4\}, \{a_5\}, \{a_7, c_7\}, \{a_8, c_8\}, \{a_9, c_9\}$, and $\{a_{10}\}$. The single event minimal cut sets align with the formal interpretation of the Activity model based on the UML specification where failed execution of an action stops the control flow. In addition, fault events a_7, a_8, and a_9 do not individually lead to system failure unless their corresponding conditions are met. This also aligns with the Activity model in which the three alternative paths are exclusive, i.e. only one mode is adopted at a time. Hence, despite the detailed structure of the Fault Tree, it is obvious that ensuring a high reliability for each of the designed system function is a straightforward way to minimise the probability of the occurrence of the top event.

Secondly, on the right-hand side of the RMS Fault Tree, the set of inhibit-gates used for the contracted fault event, $a_{7,8,9}$, implies that the actual contributions from $P(a_7), P(a_8)$, and $P(a_9)$ to $P(a_{7,8,9})$ is suppressed subject to the statistical probabilities of their corresponding mode selection. For instance, if Responsive Mode is rarely selected for a particular local RMS, the contribution of $P(a_8)$ to the top event will be insignificant (note that this is not to say that the "implement Responsive Mode" function is insignificant). Reversely, ensuring a high reliability for the operational mode mostly selected can reduce the top event probability.

Lastly, on the left-hand side of the RMS Fault Tree, one of the basic events, a_4, is observed can be considered as an external event. Tracing this fault event and its surrounding Fault Tree structure to the original functional design, it is realised that if TCC fails to "send Instruction" to the RMS or the RMS fails to "receive Instruction" from the TCC, the system will not be able to "select Mode". Instead of letting the system stuck at this point, the designer may want the system to treat this situation as if no TCC instruction is received. As such, though the local RMS will not operate in an optimal mode, it can continue to operate. The designer then has various choices to model this mechanism. For instance, the modelling can be done through revising the current Activity model to include an additional decision point or through embedding this decision logic into the next level detail under the function, "select Mode".

The top events in the two Fault Trees are different, the top event, system failure, a_s, is recognised, in fact, is contributed to the top event, non-optimal traffic flow as discussed in Chapter 3. This is because when system failure happens, vehicles entering the ramp would not be controlled under the desired strategy, hence leading to a non-optimal traffic flow. In addition, overlaps between the two Fault Trees are identified. 'Fails to adopt Collaborative Mode' as captured in the original Fault Tree is also captured in the transformed Fault Tree y the basic event, a_9. However, as suggested by the transformed Fault Tree, this fault event itself will not be a minimal cut set. In fact, it will only lead to system failure and eventually non-optimal flow when the system decides to adopt Collaborative Mode. Similarly, the fault event, 'Fails to exit Collaborative Mode' can be considered as an overlap to a_7 and a_8 collectively in which the system situationally decides to adopt one of the two other modes when it is originally in the Collaborative Mode. Again, an inhibit-gate is necessary in these two cases. From the above arguments, the developed transformation is concluded with firstly can identify additional basic events based on system behaviour models; and secondly can provide meaningful revisions to a Fault Tree constructed by engineers. From the comparison, it is also clear that due to the scope of the transformation method, the transformed Fault Tree does not capture component failures and faults that relate to objects, e.g. incorrect rate calculation.

To summarise the case study evaluation, a functional Fault Tree has been successfully generated by applying the proposed transformation method to the RMS Activity model. As the generated Fault Tree preserves structural knowledge of the original model, the qualitative of Fault Tree is analysed to derive important system reliability information and provide design improvements. By comparing the transformed Fault Tree to the one derived based on common practice, the transformation can provide meaningful addition and revision.

4.3 From Classes to Fault Tree

In this subchapter, the discovery of logical formalisation of Composition relationship of UML Classes is justified for the basis to transform system structure as represented by Class models to structural faults as represented by Fault Tree. Prior to the discovery of logical formalisation of Composition relationship, the main relationships which include three other relationships (Association, Aggregation, and Generalisation) in UML Classes are discussed as for the pertinent transformation. Probability theory (Weiss 2006) is then applied to further transform the Composition Classes structure into a conventional Fault Tree representation. The subchapter starts with discussion of mathematical semantic used for Classes relationships in UML Class diagrams. The subchapter will end with demonstration of the approach by applying the transformation method to the authorised case study introduced earlier.

System structures that modelled by using UML Classes likely employs to generate Fault Tree as relation amongst the Classes can be interpreted as causal relationships and failure propagation for fault events (Cepin & Mavko 1999). A framework to facilitate automated dependability analysis in Fault Tree representation by using UML Class model as introduced by Pai et. al. applies semantic construction based on types of Classes relationships such as Generalisation and Dependencies (Dependencies is not emphasise in this thesis). The constructed semantic is used to define stereotype for indicating the existence of Generalisation and Dependencies relationships in UML Class model and developing an algorithm (Pai & Dugan 2002). The developed algorithm is used in converting the UML Class model to dynamic type of Fault Tree. All the modelled Classes including hardware and software is converted into basic event for the tree. In the Fault Tree, only one logic gate, OR-gate, is used to connect all the basic events to the top event of the tree. By using the proposed framework, a dynamic Fault Tree can be developed as concerning fault tolerant system that considers redundant components for spare and functional dependencies. However, as claimed by (Cepin & Mavko 1999) and (Pai & Dugan 2002), the application of formal methods to precisely define modelled models removes any ambiguities about the models from the designer's mind is absence in their research.

4.3.1 Mathematical Semantic Used for Classes

Although the purpose and way of modelling system structure by using UML Class have been recognised, there is a considerable difference of users' interpretation on the relationship types by the means of mathematical semantic. Schmitt defined the mathematical semantic of Association, Aggregation, and Composition as a binary relation presentation between Classes (Schmitt 2003). This means two Classes are connected together as a pair. Whilst, Souri et. al. defined only Association is used to realise relationship between Classes and Aggregation is used to present a Class as a collection of Classes (Souri et al. 2011). The Aggregation defined by Souri et. al. brings almost the same presentation as Generalisation. Classes that are connected by Generalisation to a Class had subset and set relationship (Schmitt 2003). However, Souri et. al. call it as Inheritance which is not specified as in OMG UML Specification as the subset of Classes inherits attributes of the set Class (Souri et al. 2011). For example, Car and Bike are two Classes that connected together to Vehicle Class with Generalisation relationship, as car and bike are the types of vehicle.

The semantic of the relationships can be used to model Class Diagram from textual interpretation. Another research on Class Diagram, the diagram can be constructed from natural language processing (NLP) in C language (Ibrahim & Ahmad 2010). In the scope of NLP on textual requirement basis, two contexts (based on noun, noun phrase, and verb analysis) in a text which define as Classes can be designed using Association relationship between them. The application of Association is observed to have a similar concept of binary relation as defined by Schmitt. A part of the research, the means of using Generalisation, Aggregation, and Composition to design textual requirement in a Class Diagram are also defined. An extraction element from the context requirements can be presented as element of Class to the context Class. The relationship between the element Class and the context Class can be modelled by using Generalisation. Whilst, when the context is found to be contained in something else, Aggregation and Composition are used (Ibrahim & Ahmad 2010). Similar to UML specification, the implementation of Aggregation and Composition is subjected to strong level of relationship in the context.

In this thesis, a total failure of a system is considered as the sole failure semantic. The total failure is modelled by using propositional calculus. For instance, the failure of a facility is when the facility is partially not available such as being out of operating range, erroneous, and malfunction. Note that, inthis case, the structural proposition of interest for the modelling is facility is available such as in the range of 0-1 Although the proposition can be defined, a systematic way of defining the proposition could not be taken without knowing the possible failures. Furthermore, at this point, the UML and SysML do not facilitate the modelling of such failures. For instance, even when all the possible failures semantics of a facility are defined, i.e. the facility is operating at the range of >1, the failure logic cannot be defined through Boolean logic from the UML diagrams presentation. However, the OMG has an intention to extend the UML with safety and reliability to facilitate the modelling of failure conditions and logics (Object Management Group 2017c).

4.3.2 Application of Binary Relation

In this thesis, binary relation is used for mathematically defining two Classes in a relationship. Classes with a specific relationship are gathered in a group according of the binary relation. All Classes are assigned with unique identification, i.e. F_i with i being an integer. Taking Composition relationship of two connected Classes as an example, the Classes in binary relation is presented as,

$$R_{composition}(F_0, F_1)\ . \tag{4.3.1}$$

The ordered pair of Class F_0 and Class F_1 in (4.3.1) has a Composition relation. It is worth to note that the expression is a set of collection of ordered pairs with specific relations. For the set presentation, the ordered pair, Class F_0 and Class F_1, is the element of Composition relationship, $R_{composition}$, which can be illustrated as,

$$(F_0, F_1) \in R_{composition}. \tag{4.3.2}$$

For consistency, element on the left of the ordered pair is the owner Class and element on the right of the ordered pair is part Class. When there are two part Classes owned by F_0, the ordered pair of binary relation $R_{composition}$ is illustrated as,

$$\{(F_0, F_1), (F_0, F_2)\} \in R_{composition}. \tag{4.3.3}$$

This reads as the ordered pairs of (F_0, F_1) and (F_0, F_2) are the elements of binary relation of Composition relationship. The same rule can be applied for elements in the Aggregation binary relation set as,

$$\{(F_0, F_1), (F_0, F_2)\} \in R_{aggregation}, \tag{4.3.4}$$

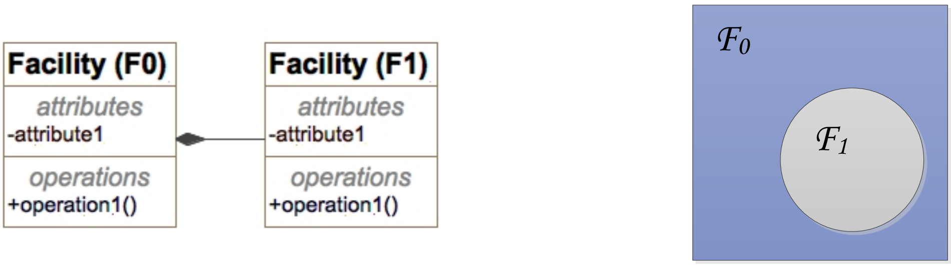
with the element on the left of the ordered pair is the container Class and the element on the right of the ordered pair is the part Class.

Composition and Aggregation relationships are specialisations of Association relationship. The Association in a diagram shows the application of binary relation which can also be applied onto Generalisation. The Composition and Aggregation relationships are classified as giving close relationship between Classes. Unlike Composition, which has the concept of child 'belongs to' a parent, Aggregation implies a relationship where the part Class can exist independently from the container Class(es). In this subchapter, the formal transformation methods are focused on transformation of hierarchical structure into component Fault Tree. Classes with Composition and Aggregation relationships are modelled in hierarchical structure. The Association between Classes is not considered as it applies on the Classes at the same level. Furthermore, the Association can lead to varies interpretation of the actual relationship being modelled. For example, a Class called Clerk represents a clerk and has an Association with a Class called report which represents a report. One can read the diagram as a clerk write a report and it also can be read as a clerk compile a report. This create the Association to be pessimistic as it can be informal and general which open to various interpretation. In a case where the Association is specified, the Association is commonly referred to as behaviour which is not defined in the UML specification. For example, the relation specified on the Association link with a solid triangle indicates the order of reading the Association of the Classes. The Class structure only meaningful with semantic of the modelled Association which can be case-by-case situation. Therefore, general rule for the Association may find it difficult to apply. Furthermore, the consideration of the Association behaviour is beyond the scope of component Fault Tree which only composes structural fault events. Similar to the application of Generalisation, the transformation of the Generalisation can lead to fictitious fault events in the Fault Tree. The Generalisation is used to represent sub-Classes by a general Class. In the actual practical system, the general Class is not presented as an individual and physical component. The transformation of the Class may have been counted by other fault events of the exact component in the tree. Thus, Association and Generalisation are not considered in the transformation from Class model to Fault Tree.

4.3.3 Relationships between Classes as Application to Logical Statement

In this subchapter, the Composition and Aggregation relationship between Classes is demonstrated by using set and relationship. This demonstration is used as a basis of formal transformation from Classes to Fault Tree.

A structure of Classes with Composition relationship is depicted in Figure 4.12(a). The Classes can be interpreted as sets as presented in Euler Diagram given in Figure 4.12(b). In the Figure 4.12(a), Facility (F_0) and Facility (F_1) that represent owner and part Classes respectively are connected together with Composition relationship which uses solid coloured diamond shape at one end of the edge of the owner Class. This relationship means that Facility (F_1) is owned by Facility (F_0). This relationship can also be illustrated by using Euler Diagram as F_1 is a subset of F_0 depicted as in Figure 4.12(b). The relations between part and owner Classes can be interpreted as in code programming. The part Class in code programming simply means reference which is created in the main program. If the reference is deleted, the code programming could not execute accurately.



(a) (b)

Figure 4.12: (a) Composition Relationship Structure of two Classes (b) Euler Diagram of (a)

By applying structural proposition in (3.2) on the Classes structure in Figure 4.12(a), the relationship can be presented as,

$$c_0 \leftrightarrow c_1. \tag{4.3.5}$$

The relation is read as the availability of Facility F_0 is necessary and sufficient for the availability of Facility F_1 in an operational system.

For the multiple of part Classes that connect to an owner Class by Composition relationship, the use of conjunction is applied to form the material equivalence relation. By using the ordered pairs of Composition relationship in (4.3.3) as example, two material equivalence statements can be produced with the use of conjunction: $(c_0 \leftrightarrow c_1) \wedge (c_0 \leftrightarrow c_2)$. The relations can be simplified as,

$$c_0 \leftrightarrow (c_1 \wedge c_2). \tag{4.3.6}$$

This relation means as the Facility F_1 and Facility F_2 are available when the Facility F_0 is available and when the Facility F_0 is available, the Facility F_1 and Facility F_2 are understood to be available as well.

However, practically, the part Classes are not necessarily complete when modelling the Classes. This does not fit to identify a complete set and complete structure of the Classes. To complement the complete set of Composition relationship, one has to define the complete part Classes. The complete set of Composition relationship can be defined as,

$$F_{complete} = (F_1 \wedge F_2 \wedge \ldots F_n) \wedge F_c. \tag{4.3.7}$$

Note that, in this case, $F_{complete} \stackrel{\text{def}}{=} F_0$, F_n represents the last modelled Class, and F_c represents a complementary Class. The complementary Class F_c is introduced to acknowledge the unpresented part Class that assumed to be the remaining unpresented part Classes of the owner Class. The complementary Class F_c could be null and it is fictitious. For brevity, with the extension to Figure 4.12(b), an Euler Diagram of the set that represents the Classes in (4.3.7) is illustrated as in Figure 4.13. The F_1, F_2, and F_n are the subset of F_0. When the subsets are taken out from the set F_0, the remaining set of the F_0 can be assumed as complementary set. The complementary set is represented by F_c.

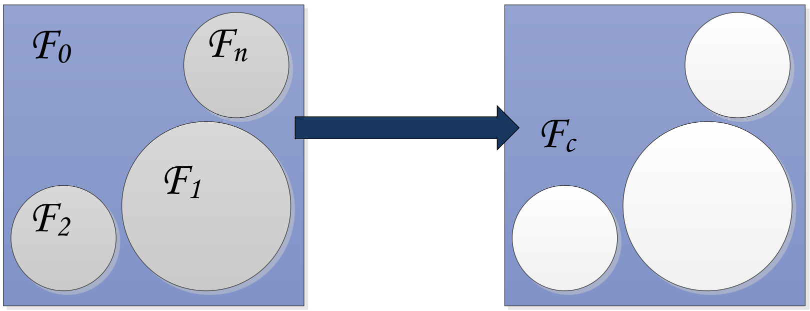


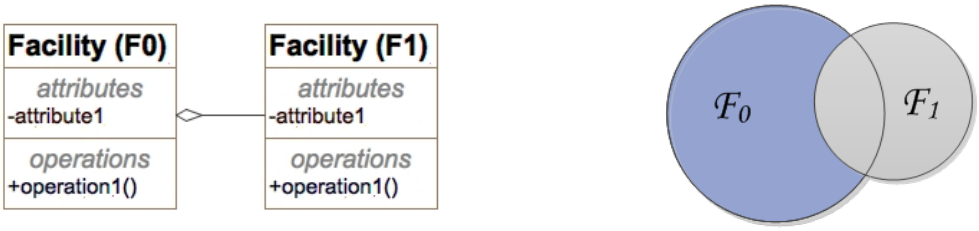
Figure 4.13: Complementary Class as presented by using Euler Diagram

Based on the relation in (4.3.7), the material equivalence of the Classes Composition relationship is defined as,

$$c_0 \leftrightarrow (c_1 \wedge c_2 \wedge \dots\ c_n) \wedge c_c. \tag{4.3.8}$$

The relation is read as the Facility F_0 is available if and only if the Facility F_1, Facility F_2, …, Facility F_n, and complementary Facility F_c are available. The … in the array of facilities represents all the facilities that modelled in the structure.

A structure of Classes with Aggregation relationship and the set representation in Euler Diagram are illustrated as in Figure 4.14. Relatively, the Aggregation relationship between part Class and container Class is not as strong as Composition relationship. One of the part Classes can have Aggregation relationship with more than one container Class which make the container Classes have the same part Class. Furthermore, in the Class model, the part Class can stand alone without its container Class.



(a) (b)

Figure 4.14: (a) Aggregation Relationship Structure of Two Classes and (b) Euler Diagram of (a)

As depicted in Figure 4.14(a), only a part Class and a container Class are required to form the simplest structure of Aggregation. Contrary to Composition relationship which the existence of part Class is dependent to its owner Class, the existence of the part Class and the container Class in Aggregation relationship are independent to each other. The implication of container and part Classes, $c_0 \rightarrow c_1$, that reads as if the Facility F_0 is available, then the Facility F_1 is available cannot hold the logical statement for the Aggregation. In particular, the deletion of container Class does not affect the part Class, i.e. the availability of Facility F_0 does not imply the availability of Facility F_1. In addition, the part Class can have more than one container Class which cannot hold the implication structure. Therefore, the logical statement of Classes in Aggregation relationship cannot be formed.

4.3.4 Application of Contrapositive to Classes with Composition Relationship

A material equivalence statement remains True if the first (left-hand side) and second (right-hand side) propositions change the position. This is because both of the propositions are necessary for each other. For a standardisation, by using the same concept of forming functional fault event from Activities, contrapositive is used for forming fault from the viewpoint of system structure, i.e. structural faults of a system. By applying contrapositive to the material equivalence statement of Composite relationship structure in (4.3.5) and considering complementary Class as described in (4.3.7) and (4.3.8), the fault viewpoint form is presented as $\neg c_1 \wedge c_c \leftrightarrow \neg c_0$. And by using the structural fault event as declared in (3.5), the proposition statement can be simplified as,

$$f_1 \vee f_c \leftrightarrow f_0. \tag{4.3.9}$$

The relation of the structural fault events in (4.3.9) is read as Facility F_1 or Facility F_c or both unavailable if and only if Facility $F_0$is unavailable.

4.3.5 Failure of Structural Probability Analysis

In the conventional Fault Tree, the basic events of the tree are consisted of system structure (facility) failures that lead to the system function (action) failures. For instance, the probability of top event of the tree is calculated by the given probability of failure of facilities defined in the basic events through intermediate event.

Here, the defined relation of structural fault events can be used to express probability of failure of the facilities. In the case where the complementary structural fault event f_c of the relation in (4.3.9) is not required, f_c is null. Therefore, the probability of the complementary structural fault event is zero, i.e. $P(f_c) = 0$. The new relation of the structural fault can be presented as,

$$f_0 \equiv f_1. \tag{4.3.10}$$

Thus, the probability of f_0 is equal to the probability of f_1, $P(f_0) = P(f_1)$. For the relation of two part Classes and an owner Class as in (4.3.6), the fault viewpoint of the Classes can be presented as,

$$f_0 \equiv f_1 \vee f_2. \tag{4.3.11}$$

Similar to the previous case, assuming that the declared complete Classes structure do not require complementary Class. A related point to consider is the complementary Class does not involve in the transformation into Fault Tree and it is introduced for the architecture design analysis. Therefore, the probability of the relation in (4.3.11) is defined as $P(f_0) = P(f_1 \cup f_2)$ with $P(f_c) = 0$. The probability of failure can be calculated as $P(f_0) = P(f_1) + P(f_2) - P(f_1 \cap f_2)$. The intersection of probability of f_1 and f_2, $P(f_1 \cap f_2)$, is less than 0.001% and the value does not give a big impact to the probability calculation. Thus, the probability analysis on the complete form as in (4.3.8) for Composition relationship is allowed. The material equivalence statement in (4.3.8) can be expressed with equivalence sign, ≡, as,

$$f_0 \equiv (f_1 \vee f_2 \vee \ldots \vee f_n) \vee f_c. \tag{4.3.12}$$

The relation can be defined as, with disjunction related concept in set theory (Thomas 1968),

$$P(f_0) = P((f_1 \cup f_2 \cup \ldots \cup f_n) \cup f_c). \tag{4.3.13}$$

Based on the (4.3.13), the probability of fault event f_o, $P(f_o)$, is equal to the probability of the union of fault events f_1, f_2, …, f_n, and f_c. The generic equation for the result as defined by (Weiss 2006),

$$P(f_o) = \cup_{i=1}^{n} P(f_i). \tag{4.3.14}$$

According to the relation, one can understand that the union of probability of fault events which transformed from part Classes of Composition relationship is equal to the probability of fault event which transformed from owner Class of the Composition relationship with the assumption of the part Classes are complete.

4.3.6 Classes to Fault Tree Transformation

The probability equation of fault events (4.3.13) is used to structure the Fault Tree that shown in Figure 4.15. The complete set of fault events f_1, f_2, …, f_n, and f_c are defined as basic events of the tree with the fault event f_0 is defined as intermediate event or top event of the tree. All the basic events are connected by OR-gate as translated from the probability calculation in (4.3.13).

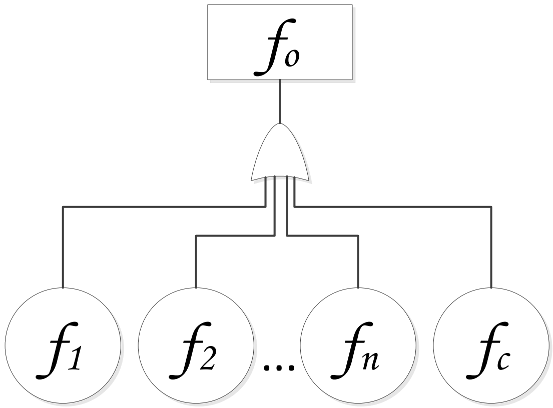


Figure 4.15: Fault Tree of Composite Relationship Structure

4.3.7 Multiplicity of Relationships for Fault Tree Transformation

In this subchapter, the multiplicity that specifies the cardinality (numbers of facilities) in a Class model is observed to expand the structural fault events in the component Fault Tree that have been transformed from Classes. Two logic gates are used to connect the expanded structural fault events in the component Fault Tree based on the classification of redundant facilities or multiple facilities that operate concurrently. The expansion of structural fault events is limited to the unlimited number of element which uses asterisk, '*', for multiplicity in the Class model.

Every component (facility) that designed for a system has to be counted because the quantity gives significant impact to the design and safety of the system. A system may be consisted of more than one of the same facility. In the development of safety-critical system, the same facilities that supply the same function can be observed to be designed in a system as redundant or back-up. This is because the redundant facilities are designed for the safety of the system. As referring to ARP 4761, redundancy is an approach of reliability and availability improvement even for short time operation (National Instruments 2008). There is also more than one facility that fabricated in a system which operate at the same time. For example, two traffic lights of a traffic management system that give indicator to two different roads are operated at the same time but not interfering each other.

In system structure modelling, the multiplicity is used to define the number of facility in a system. The details of the facilities can be further explored with the consideration of Object Diagram in the system design. Class Diagram and Object Diagram can be thought as in metamodelling of abstraction layers of higher model (instance graph) and lower model (type graph) (Gogolla et al. 2005). This defines the closed relationship between the respective diagrams. Thus, all of the abstraction layers can be presented into a single Object Diagram (Gogolla et al. 2005). Object Diagram and Class Diagram provide relationship and multiplicity of specific Objects that instance to the Classes (Kuske & Gogolla 2002). The information of Classes is broken down in depth with the presentation of Objects. In this thesis, the system structure modelling for Fault Tree transformation is limited to the Class Diagram.

In the fault viewpoint, for the facilities that have the same function and fabricated in a system, the failure of these facilities could affect the function. For the Fault Trees of this thesis, at the lowest level of the tree, the fault events of these facilities can be tied together with two logic gates: OR-gate or AND-gate. The OR-gate is used to tie structural fault events of facilities that operate at the same time. These facilities are not redundant as the failure of only one facility can lead to the failure of the function. In the Fault Tree, the output event of these structural fault events (basic events) can be identified as the general name of the basic events and it is fictitious. This means the output event will occur if any of the connected basic events occurs. In another case, the AND-gate is used to tie the basic events of facilities that are not operated at the same time but redundant to each other. This refers to the redundant facilities in a system. When one of the redundant facilities is failed, the system can depends on the redundant facility to operate and continue to serve the function. Thus, when all of the redundant facilities are failed, the function that should be provided by the facilities cannot be executed and could affect the overall system performance.

4.3.8 Overarching Metamodel of the Formal Transformation from UML Class Model to Fault Tree

In this subchapter, the developed method for transforming Classes in Composition relationship to a Fault Tree is abstracted into Class model-Fault Tree (CM-FT) overarching metamodels. The CM-FT overarching metamodel bridges the Reduced Classes Metamodel (RCM), the transition method, and Fault Tree Metamodel (FTM) as depicted in Figure 4.16.

The CM-FT overarching metamodel is composed of 11 metaclasses that are differentiated by white-grey-dark scale (white, grey, and blue for the coloured version). On the left, four metaclasses (shown in grey), Class, Property, Association, and Multiplicity, are inherited from the RCM. These metaclasses are the instance of Classes (including owner and part Classes) in Composition. In the middle, three metaclasses (shown in white), are introduced for the metamodeling of transition method. In particular, the transition method is in the form of the relation of structural fault events. On the right, four metaclasses (as shown in blue), ORGate, ANDGate, OutputEvent, and BasicEvent, are inherited from the FTM. These metaclasses inherit the relations established in their domain metamodel.

The transformation from Class model to Fault Tree involves transition method. The mapping of Classes in Composition to the transition method is captured by a stereotype, <<contrapositive>>. The mapping is a structure-based which connects Property (owned by Class)-Association (presented as Composition) structure to the FacilityFault-MaterialEquivalence structure. These structures are circled in dash lines and connected via the <<contrapositive>> relation.

The transformation from transition method to Fault Tree structure is classified as entity-to-entity mapping. The <<equivalence>> stereotype is introduced to define the mapping without the need of modification. Two transformations are defined by the <<equivalence>> stereotype from Facilityfault:

1. FacilityFault transformed from owner Class is mapped onto OutputEvent.

2. Facilityfault transformed from part Class is mapped onto BasicEvent.

In the resultant Fault Tree, the BasicEvent are connected by ORGate to the OutputEvent when the number of the BasicEvent is more than one which can refer to the multiplicity 1…* to 1. In the CM-FT overarching metamodel, the Class is associated with MultiplicityElement to specify the cardinality of the modelled Class which may or may not present redundancy. The MultiplicityElement provides the information to expand fault event through BasicEvent. The BasicEvent are connected via two types of logic gates: ORGate or ANDGate. Respectively, the use of different types of logic gate is dependent to the redundant facilities or multiple facilities as discussed in the previous subchapter.

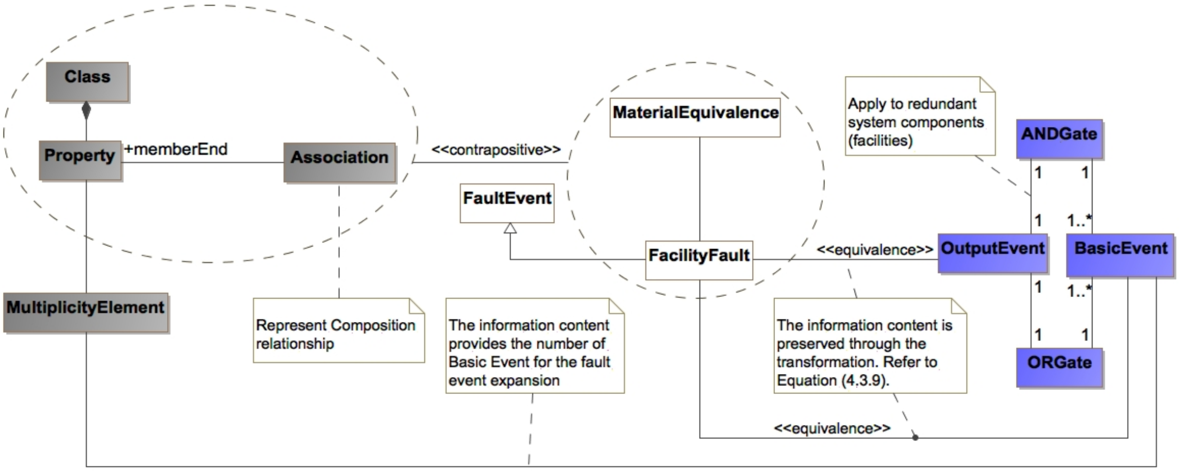

Class
Property
+memberEnd
Association
<<contrapositive>>
MaterialEquivalence
FaultEvent
FacilityFault
<<equivalence>>
Apply to redundant system components (facilities)
ANDGate
OutputEvent
BasicEvent
ORGate
1
1
1
1..*
1
1..*
1
1
MultiplicityElement
Represent Composition relationship
The information content provides the number of Basic Event for the fault event expansion
The information content is preserved through the transformation. Refer to Equation (4.3.9).
<<equivalence>>

Figure 4.16: Class model-Fault Tree Overarching Metamodel

4.3.9 Ramp Meter System Component Fault Tree

In the Subchapter 3.4.2, the constructed conventional Fault Tree based on the five RMS faults identified in the original study of the RMS in (Ingram et al. 2014) as depicted in the **Error! Reference source not found.** is revisited in Subchapter 4.2 for the basis of functional-based faults (Operational faults). Again, in this subchapter, the constructed conventional Fault Tree is revisited for the basis of structural-based faults (Faulty traffic light) of RMS. The developed transformation method is applied to the Class model of RMS in Figure 3.11 to generate a component Fault Tree that concerns the facilities of the RMS that modelled in UML Classes.

Based on the Classes to Fault Tree transformation discussed in Subchapter 4.3.3, Classes with Composition relationship are allowed for the Fault Tree transformation. Classes with other relationships such as Association, Generalisation, and Aggregation are not involved in the Fault Tree transformation. According to the Class model of RMS, three Classes in Composition are identified: “Data Transceiver” (F_4), “Data Transmitter” (F_5), and “Data Receiver” (F_6). The part Classes, “Data Transmitter” (F_5), and “Data Receiver” (F_6), are assumed as a complete part Classes to the owner Class, “Data Transceiver” (F_4). Therefore, complementary Class is not necessary to consider in the Class structure.

The material equivalence statement of the Class is structured as $c_4 \leftrightarrow c_5 \wedge c_6$. Then, contrapositive is applied to the statement for fault viewpoint. Without considering the probability analysis, by applying the rules developed in (4.3.9.), (4.3.10), (4.3.11), and (4.3.12), the fault events of the facilities are generated as $f_4 \equiv f_5 \vee f_6$. Finally, based on the structural fault events, a component Fault Tree of the facilities is generated as depicted in Figure 4.17. The top event of the generated Fault Tree is transformed from the owner Class and all the part Classes are transformed into basic events which connected by OR-gate to the top event. The facilities of RMS is modelled in UML Class without multiplicity. Therefore, the consideration of multiplicity for redundant and more than one of the same facilities in the system is not applicable for the expansion of structural fault events.

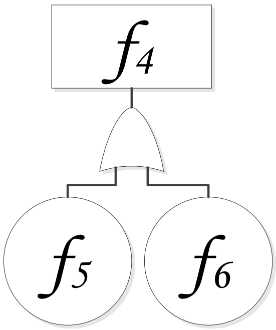


Figure 4.17: The Ramp Meter System Structural Fault Tree

The Class model in Figure 3.11 is a modified version from the original work in (Ingram et al. 2014). In the original work, the TMSoS model is structured for considering fault tolerant design within the SoS. The fault tolerant is designed for analysing redundant CSs in the SoS. In this thesis, a complete Class model is considered for a holistic viewpoint of the corresponding traffic management system. The complete Class model of TMSoS can be used to demonstrate the transformation from facilities that modelled in UML Class to Fault Tree.

4.4 Summary of Formal Transformation Method for Automated Fault Tree Generation from Single UML System Model

In this chapter, two formal transformation methods have been separately developed for generating static type of Fault Trees from system models developed by using the graphical language, UML. The methods developed align with current industrial practices in early state system assurance (Zeller et al. 2016) and advances existing approaches in terms of accommodating system model availability (Majdara & Wakabayashi 2009), (Bhagavatula et al. 2016) and incorporated mathematical rigor (Mhenni et al. 2014), (Yakymets et al. 2013).

The development of the first transformation method is based on propositional logic and probability theory to allow control flows modelled in UML activities to be transformed into semantically equivalent Fault Trees. A new concept, FPC, is introduced as an intermediate step to facilitate the transformation method. The formal basis of the transformation method guarantees the generated fault Tree to be semantically correct. An important finding is revealed where the formal approach suggests that mappings should be based on the relational structure between entities and not just entity-to-entity relationships. Therefore, as the relational structure of system behaviour is preserved through the transformation, the generated Fault Tree will be named functional Fault Tree in the rest of this thesis.

The development of the second transformation method is based on propositional logic and relationship to allow the hierarchical structure of systems modelled in UML Classes to be transformed into semantically equivalent Fault Trees. To enable the transformation, complementary Class is introduced as an assumption to ensure completeness in the hierarchical modelling of the system. In this transformation, only the hierarchical structure of a system is considered, the operations modelled in the UML Classes will be considered in the transformation method in Chapter 5. The generated Fault Tree will be named as component Fault Trees in the rest of this thesis.

The semantic interpretation of the Activity and Class Diagrams is grounded to the used of propositional calculus. The semantics interpretation works based on the five control flows in Activity Diagram and Composition relationship in Class Diagram which are used in system modelling. By using propositional calculus, the semantic interpretation of the five control flows and Compositional relationship become formal. This gives a direct and fixed interpretation. However, the models have to be changed whenever new specification is needed for the system. Furthermore, the failure semantics for the Activity Diagram and Class Diagram are addressed at the crash failure. This means the failure modelling is designed from the perspective of total failure of the system.

To support the implementation of the transformations, two overarching metamodels were developed to bridge the domain-specific metamodels introduced in Chapter 3. First, an AM-FPC-FT overarching metamodel was developed to bridge the metamodels of Activity models, FPC, and Fault Trees. Second, a CM-FT overarching metamodel was developed to bridge the metamodels of Classes in Composition relationship and Fault Trees. The formal basis of the transformation method together with the overarching metamodels should facilitate platform-independent implementations of automated Fault Trees generation from well-formed UML models.

To demonstrate and evaluate the applicability of the methods, the developed transformation methods were applied to the RMS case study introduced in Chapter 3.

Fault Tree qualitative analyses were then carried out to evaluate the quality of the modelled system architectures.

CHAPTER 5

AUTOMATED FAULT TREE GENERATION FROM INTEGRATED UML SYSTEM MODELS

5.1 Introduction

The functional and component Fault Trees in the previous chapter are generated from the respective UML Activity and Class models by using two separated formal transformation methods. However, in a conventional Fault Tree, functional and structural aspects of fault and failure are often bounded in one tree. Therefore, in this chapter, a transformation method Fault Tree is developed to integrate functional and structural aspects of system failure based in one tree. The implementation of allocation for system modelling in SysML is discussed in this chapter. The concept of allocation is adopted in facilitation that will be introduced in the transformation method. The concept of facilitation is introduced to differentiate the usual use of allocation in SysML which is not capable to model complicated situations in which individual functions are not allocated to individual components. The implementation of the concept in the transformation method uses Activity model as the backbone for generating Fault Tree that will be named integrated Fault Tree. The presentation of the integrated Fault Tree demonstrates a hierarchy structure of functional and structural fault events. The transformation method with facilitation also introduces separation of contracted fault event in the FPC into individual fault event for identifying structural fault event that forms basic events. The embedded functions (operations) in the Classes can then be used to decompose the basic events of the integrated Fault Event. By using the concept of ownership by which the Classes own the respective functions, an elaborated Fault Tree is generated. The elaborated Fault Tree presents transfer fault events and an insight of exploring the transformation method. The transformation method with facilitation and ownership for integrated Fault Tree and elaborated Fault Tree generation is applied to TMSoS case study. The generated Fault Trees can be used to analyse the safety and reliability of a system through system functions and system components.

The remainder of this chapter is organised into three subchapters as follows.

Subchapter 5.2: Allocation

In the modelling language, the concept of allocation has been introduced by the OMG. The allocation is implemented in SysML as a stereotype. The application of allocation in SysML is discussed. The discussion also includes the types of allocation that has been organised in SysML.

Subchapter 5.3: Facilitation

The concept of facilitation is introduced in a development of a formal transformation method for generating a Fault Tree that integrates functional and structural faults. The Fault Tree which then called integrated Fault Tree presents hierarchy structure of functional and structural fault events. The transformation method uses Activity model as the backbone for the facilitation and the integrated Fault Tree generation. An overarching metamodel of the formal transformation method is developed to bridge the metamodels of FPC and Fault Tree. The formal transformation method is applied on TMSoS case study for demonstrating the concept of facilitation in the transformation method.

Subchapter 5.4: Ownership

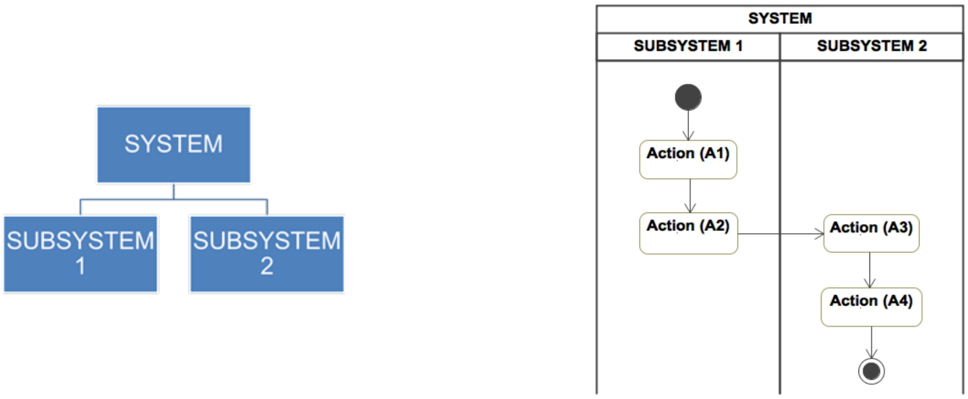
A concept of ownership is also introduced as a part of the transformation method. The ownership is applied to Classes by which facilities own the embedded functions, i.e. Operations. With the implementation of ownership, the basic events in the integrated Fault Tree will be elaborate. A new Fault Tree named elaborated Fault Tree is presented with the implementation of transfer events.

Subchapter 5.5: Summary of Automated Fault Tree Generation from Integrated UML System Models

A development of formal transformation method for integrating functional and structural viewpoints of UML system models in a Fault Tree is summarised. In addition, the structure of formal transformation methods developed in Chapter 4 and Chapter 5 is presented.

5.2 Allocation

A function needs a medium in order to be realised, i.e. a component. A collection of functions can also produce a higher-level functionality that can be understood as emergent behaviour. As mentioned in Subchapter 4.2.4, through allocation of functions to system components, the achievement of system behaviour can be observed. Furthermore, in system thinking and modelling, function at the system level is initiated by functions at subsystem level. As depicted in Figure 5.1(a), a system can be composed of subsystems that may or may not have further composition. According to INCOSE, a system can be presented as a black box which supressing the subsystems and their interrelation. These subsystems are established and has their interrelations by which enable the system to achieve its stated purpose (International Council on Systems Engineering 2015). The interrelation between subsystems can be interpreted in the presentation of Activity Diagram as shown in Figure 5.1(b). The functions of both subsystems, Subsystem 1 and Subsystem 2, are manifested by sequence of actions which started with Action A_1 of Subsystem 1 and ended with Action A_4 of Subsystem 2. Therefore, the functions of Subsystem 1 and Subsystem 2 are interrelated. Each of the functions are contributed to the functionality of the system.



(a) (b)

Figure 5.1: Structure Level of Functions and System presented by:(a) System Hierarchy (b) Activity Diagram

Allocation of functions plays an important role in system modelling especially system architecture modelling. As stated in (Price 1985), allocation of function is the most basic system design decisions in design phase. This is because, allocation of functions has become a systematic approach in designing man and machine integrated system. This is supported by (Lowe & Lowe 2015) which emphasises on how system design influences human performance and safety with different parts of system can be modelled with allocation of functions. The concept of allocation of functions is an approach for modelling based on multiple viewpoints. Hence, system modelling can be decomposed into several categories such as behaviour and structure modelling. This includes interaction and dependencies in the system that can be viewed on the system architecture design.

In UML Activity, allocation of functions can be presented by using swimlane (c.f. Table 3.1). However, SysML has an improvised version for modelling Activities with the implementation of allocation. The concept of allocation is extensively defined in which to organise cross-association elements within various structure models. Allocation is the term used to represent general relationships that map one model element to another. Allocation can be used in various ways for specific purpose such as to allocate functions to structure of a system. In some cases that involve computing, allocation technique can be applied to allocate software to hardware. Based on SysML Specification, the typical types of allocation in system engineering are organised as follows (Object Management Group 2017a):

1. Behaviour allocation. Allocation of behaviour (function) to structure, or behaviours (functions) to behavioural (functional) features such as operation.
2. Flow allocation. Allocation of flow of one system representation to flow in other system representation such as flow in functional to flow in structural, i.e. flow in Activity and flow in Block, and control flow to object flow in a Activity.
3. Structure allocation. Allocation of components between separated systems.

According to SysML specification, the common practice of using allocation is with the allocation of Activities to Blocks. This is classified as behaviour allocation that allocate functions to components. In system modelling, allocation can be presented in several ways. As depicted in Figure 5.2, in SysML Activity Diagram, a compartment of 'allocatedTo' is displayed on an action to define which element in system model that the action is allocated to the stated place.

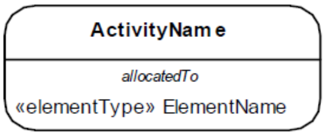


Figure 5.2: Application of Allocation for an Action in SysML (Object Management Group 2017a)

In another example on the presentation of using allocation in SysML Activity model, a swimlane that specified for <<allocate>> is displayed. The <<allocate>> is continued with specific system component's name. With this presentation, one can notice that all actions that modelled in the specific swimlane are allocated to the system component (facility).

Nevertheless, the application of behaviour allocation is restricted to view the allocation from the lens of action. In addition, the structure allocation which can allocate to multiple facilities is typically applied to just between facilities. Therefore, an improved concept, namely Facilitation, which extends the concept of allocation is introduced to enable a clearer representation of more complex allocations. The idea is look the allocation reversely by describing the mechanism through facility (facilities) *facilitating* action(s). This enables the modelling of complicated allocations such as a set of facilities facilitating an action. The main contribution of facilitation is to allow modelling a subset of a set of facilities facilitating one action, while another subset of the same set of facilities facilitating another action.

5.3 Facilitation

In this subchapter, the concept of allocation is adopted into the concept of facilitation that will be introduced. In the transformation method, facilitation allows different viewpoint through the system modelling. The concept of facilitation combines system models that modelled in UML Activity and UML Class. Facilities in the UML Class are identified to which facilitate actions in the UML Activity. The unique identification of facility in the Class model is used to remark the facilitation on the actions in the Activity model and keep the traceability during the facilitation process.

The application of facilitation is not a new innovation. The facilitation can be seen as allocation of information in UML Deployment Diagram. The allocation of information in UML Deployment Diagram shows the distribution of logical or physical artefacts including software. Similar to other UML diagrams, the information in the Deployment Diagram is represented as nodes. The nodes are connected to create networked systems. The networked system is described as an architecture of a system at specification level or instant level. The idea of facilitation is not recreating the Deployment Diagram but to locate lower level into higher level information of a system being modelled. As the developed method is based on transformation from the behavioural viewpoint, the facilitation connects a higher level to a lower level through the use of a system element (i.e. physical or abstract) facilitating a function. This allows hierarchical structure between Activity and Class.

Facilitation of component to function of an individual system has a strong correlation. For a normal condition of a system, multiple components are integrated as a system to facilitate the system for serving its specified functions. This can be illustrated by an example of a cyber-physical system (CPS), functionality of a vehicle control system is relied upon integrated various subsystems (Baheti & Gill 2011). Another example of facilitation of system component to system function, in a viewpoint of computerised system, a multiple of software application is supported by a piece of hardware (Laprie et al. 1990). Apart of achieving specified purposes, from a failure viewpoint, both are anticipated for the system failure dependency. For example, from a viewpoint of on ground transport system, the function of fuel system for supplying fuel to engine can be affected by broken cylinder or leaked storage tank. Conversely, structural failure can be caused by functional failure as well as by malfunction. For example, with considering filament wound composite tubes, an operating tube is subjected by internal pressure

(Martins et al. 2012). This means a higher internal pressure can damage the composite tube.

The concept of facilitation is implemented for the transformation method from UML system models to Fault Tree generation. Practically, a Fault Tree is generated to analyse safety and reliability of a system through the elements of the system such as function and components. Therefore, by implementing the concept of facilitation, the transformation method integrates functional failure and structural failure of a system in the generated Fault Tree. The concept is applied to the system modelling. The actions that modelled in UML Activity are facilitated by facilities that have been modelled in UML Class. Then, the model is viewed in the fault viewpoint prior the transformation to Fault Tree. In the rest of this thesis, the generated Fault Tree will be named integrated Fault Tree.

5.3.1 Activity Model as the Backbone for Facilitation of Class

In the work presented in Chapter 4, by using the formal transformation method, a complete transformation from UML Activity model to Fault Tree generation with preserved relational structure of system behavior through the transformation is observed. However, in the work presented in Subchapter 4.2, the generated functional Fault Tree is exclusively consolidated of only functional faults and failures. Therefore, the UML Activity model will be used as the backbone to implement the concept of facilitation in the transformation method. Furthermore, in the common practice of Fault Tree construction, the system failure is defined earlier for deductive reasoning of the occurrence of the failure.

To implement mathematical basis in the facilitation, the propositions for action, p_i = The Action i complete execution, and facility, c_i = The Facility i is available, defined in Chapter 3, (3.1) and (3.2), are employed. Rationally, for an action to complete its execution, the action must be facilitated by at least one facility. Similarly, in vice versa, the availability of a facility means execution of action is expected. Thus, mathematically, the relationship between action and facility can be depicted by using Euler's diagram as depicted in Figure 5.3. From the Euler's diagram, one can relate that the Facility i facilitates Action i to complete its execution.

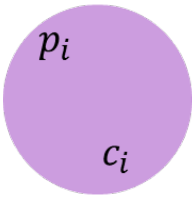


Figure 5.3: Relationship between an Action and a Facility

With the defined propositions, the relation between Action i and Facility i can be formed as follow,

$$p_i \leftrightarrow c_i. \tag{5.1}$$

Similar to the concept implemented for Composition relationship, the formation of material equivalence statement can be read as Activity i complete execution if and only if the Facility i is available. This means, in reverse direction, this statement can be also read as Facility i is available if and only if the Activity i completes its execution. Thus, as depicted in Table 5.1, when both of Activity i and Facility i are equivalent then the statement is True. This means, the Activity i will completes its execution only with the availability of Facility i. In other words, the Activity i will not complete its execution without the availability of Facility i.

Table 5.1: Truth Table of Material Equivalence between Action and Facility

p_i	c_i	$p_i \leftrightarrow c_i$
False	False	True
False	True	False
True	False	False
True	True	True

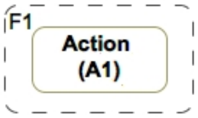
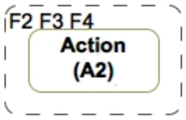
In most of the cases, at the high level system, a function (action) could be facilitated by more than one component (facility), and a component can facilitate more than one function. For example, when two facilities (assume these facilities are not redundant), Facility F_1 and Facility F_2, are required to facilitate only one Action A_1, mathematically, the statement can be performed as,

$$p_1 \leftrightarrow c_1 \wedge c_2. \tag{5.2}$$

This statement represents the Action A_1complete its execution if and only if both Facility F_1 and Facility F_2 are available. From system modelling viewpoint, one can observe that the Action $A_1$is facilitated by both Facility $F_1$and Facility F_2.

The facilitation are categorised into two types: single and multiple. Both of the types of facilitation can be applied to a facility. Based on the discussion of the facilitation of system components to system functions, one way to present the facilitation relation in system modelling is by using UML system models. The facilitation is firstly presented in Activity model. The facilitation can be presented on each action that has been modelled in UML Activity. The action is facilitated by at least one facility which has been modelled in Class model of the same system. The facilitation provides a representation of a subset of a set of facilities facilitating one action, while another subset of the same set of facilities facilitating another action although these actions are modelled in the same swimlane. The facilitation uses a dashed box with identification number of the respective facility in the Class model located at the top left corner. The dashed box is placed as the outer layer of the corresponding action to express the facilitated facility of the action. Further description and example of each type of facilitation are explained in Table 5.2.

Table 5.2: Types of Facilitation

Types of Facilitation	Description
 Single	A facilitation of one facility to an action. This type of facilitation is used as the one facility facilitates an action.
 Multiple	A facilitation of multiple facilities to an action. This type of facilitation is used as a set of facilities facilitates an action.

The example of multiple type of facilitation in which three facilities facilitate an action is depicted as in Figure 5.4. The action is surrounded by a dashed box. At the top left corner of the dashed box are the unique identification of the facilities that facilitate the particular action. Based on the figure, the "collect Data" (A_1) action is facilitated by three facilities, Facility F_1, Facility F_2, and Facility F_9. The Action A_1 has an outer layer of dashed line with F_1, F_2, and F_9 identification number of facilities that can be traced in Class model located at the top left corner of the box.

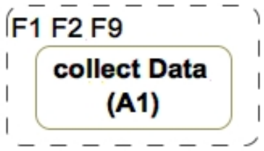


Figure 5.4: Facilitation of Three Facilities to an Action

5.3.2 Facilitation in Fault Propagation Chain

In this subchapter, the facilitation of facilities to actions in the UML Activity is applied to transformation method for generating integrated Fault Tree. As the UML Activity can be transformed into FPC for functional fault viewpoint, the facilitation of facilities to action can also be transformed into the FPC for structural fault viewpoint. This transformation integrates structural fault events and functional fault events into the FPC presentation.

For the FPC presentation, following the formal transformation methods in Chapter 4, contrapositive is applied to present the facilitation of facilities to actions in fault viewpoint. By using the statement in (5.2) as the example, the fault viewpoint of the statement is formed as follows,

$$\neg p_1 \leftrightarrow \neg c_1 \vee \neg c_2, \tag{5.3}$$

and the negated propositions can be simplified as,

$$a_1 \leftrightarrow f_1 \vee f_2. \tag{5.4}$$

The application of contrapositive is to acquire fault viewpoint for Fault Tree generation. As the contraposition statement presented in (5.4), Action A_1 is failed to complete its execution if and only if either Facility F_1 or Facility F_2 is not available at the least. This relates back to the logic of Action and Facility in the first row of Table 5.1 where the statement is True when Action and Facility are both False.

The facilitation of facilities to actions in the fault viewpoint can be presented into the FPC. The structural fault events are also presented by using blocks with specific identification. The structural fault events are connected to the corresponding functional fault events, as facilitate to in fault viewpoint, by using double (start and end) dashed arrow. For example, as depicted in Figure 5.5, the two facilities fault events are connected to an action fault event by using the double dashed arrow. The double dashed arrow is used as following logical model as in (5.4), i.e. material equivalence. The facilitation is classified as multiple type of facilitation.

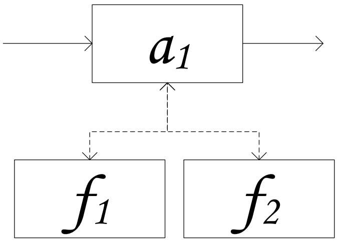


Figure 5.5: Multiple Type of Facilitation on Fault Propagation Chain

The facilitation of facilities to actions can be observed in both success and fault viewpoints. Based on the concept of facilitation of facilities to actions, in fault viewpoint, one can observe the failure of facilities contributes to the failure of correspond actions. This can be clarified by presenting the calculation of probability of the failures. The probability calculation is supported by two assumptions as follows:

1. Every facility has probability of failure, and

2. Probability of failures of different facilities is independent to each other.

The latter assumption leads to another level of detail which is not considered in this current situation, i.e. where anything propagates into facility is independent to the facility. Therefore, the probability of failure of action which is discrete as defined in

Chapter 4, is determined by probability of failure of the contributed facility, i.e. the output of action is determined by facility.

Probability of failure of each fault event on the chain in Figure 5.5 is denoted as $P(a_1)$ for a_1, $P(f_1)$ for f_1, and $P(f_1)$ for f_2. With the consideration of Assumption 1 and Assumption 2, the probability of failure of the Action A_1 , $P(a_1)$, is contributed by probability of failure of Facility F_1 and Facility F_2, $P(f_1)$ and $P(f_2)$. One way to calculate the probability of failure is through calculating the probability of success which is the yield of one minus the value of probability of success. This can be calculated based on n experiment runs on the system. To calculate the probability, a truth table of the two facilities is constructed as in Table 5.3.

Table 5.3: Truth Table of Probability of Failure

f_1	f_2	$f_1 \wedge f_2$
Failure	Failure	$\frac{nP(f_1)P(f_2)}{n}$
Failure	Success	$\frac{nP(f_1)(1-P(f_2))}{n}$
Success	Failure	$\frac{n(1-P(f_1))P(f_2)}{n}$
Success	Success	$\frac{n(1-P(f_1))(1-P(f_2))}{n}$

In the las row of the right column of the table defines the probability of success of both Facility F_1 and Facility F_2. The alternative way to calculate the probability of failure of both Facility F_1 and Facility F_2 is by deducting the probability of success from one. Thus, based on the facilitation of facilities to action, probability of failure of Action A_1 can be calculated from the probability of failure of Facility F_1 and Facility F_2 as follows,

$$P(a_1) = 1 - (1 - P(f_1))(1 - P(f_2)),$$

$$P(a_1) = P(f_1) + P(f_2) - P(f_1)P(f_2). \tag{5.5}$$

From the probability calculation in (5.5), one can relate with (5.4) of facilitation of facilities to action. Furthermore, the calculation can be observed having the same operation with OR-gate in Fault Tree. Therefore, a Fault Tree of structural and functional fault illustrated as in Figure 5.6 can be generated with two structural faults as basic events and functional fault as the top event. The failure probability of the top event can be calculated based on statistical probabilities of the corresponding basic event on facility failure.

The probability definitions proposed for the analysis of facilitation is based on probability calculation on conventional Fault Tree. The probability is defined based on the failure rate of facility which can be calculated accordingly. This is because, in reality, a failed system component may often be noticed when the expected system function has not been served. In terms of structure and behavior modelling, this is referred to as facilitation of system components to system functions in this research. For example, if two system functions are facilitated by a system component, then the probability of failure of the system component contributes to probability of failure of each system function.

Taking the facilitation on FPC in Figure 5.5 as an example, based on the probability calculation, the FPC can be transformed into a Fault Tree that consists of two basic events that are connected by an OR-gate to an intermediate event as depicted in Figure 5.6. Furthermore, based on probability equation in (5.5), $P(a_1)$ can be considered as a union of $P(f_1)$ and $P(f_2)$. By considering the Fault Tree structure, the occurrence of a functional failure caused by the occurrence of a collection of basic events of structural failures can be observed.

The connection between basic event (structural fault event) and intermediate event (functional fault event) is determined by the type of facilitation, i.e. single and multiple. In the case of single category of facilitation, one will observe a basic event is attached to its intermediate event without any logic gate. However, a group of basic events that connect together by an OR-gate to the intermediate event is obeyed to the multiple type of facilitation.

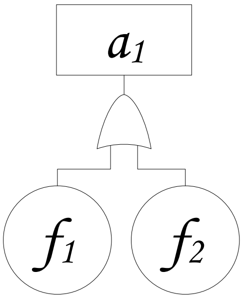


Figure 5.6: Integrated Fault Tree derived from (5.4)

The derivation of (5.5) can go beyond facilitation of two facilities to an action i.e. three or more facilities are facilitating an action. Therefore, a generic equation of the contributing failure probability is derived as,

$$P(a_i) = n - n\prod_i^m (1 - P(f_i)). \tag{5.6}$$

The probability of failure of Action A_i, $P(a_i)$, is contributed by the deduction of sum of probability of success of m number of facilities from the total number of experiment, n. Note that, i of functional fault, a, and i of structural fault, f, are not the same but they are related in terms of failure contribution of 'to' and 'from', i.e. facilitation.

5.3.3 Overarching Metamodel of the Transformation of Modelled Facilitation to Fault Tree

In this subchapter, the metamodels of FPC with facilitation and Fault Tree are considered for the overarching metamodel. The metamodel of facilitation of facilities to actions is not considered for the overarching metamodel as the facilitation can be presented exactly in fault viewpoint of FPC. An F-FPC-FT overarching metamodel is presented to bridge modelled facilitation in FPC presentation and Fault Tree Metamodel. The F-FPC-FT overarching metamodel is composed of seven metaclasses that are differentiated by white and dark scale (white and blue for the coloured version) as depicted in Figure 5.7. Respectively, the two colours are used to differentiate the FPC and Fault Tree groups, i.e. left and right positions in the F-FPC-FT overarching metamodel.

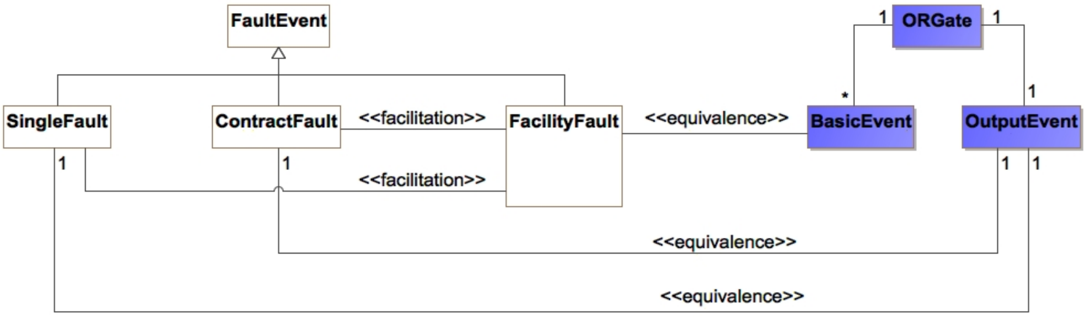


Figure 5.7: Facilitation-Fault Propagation Chain-Fault Tree Overarching Metamodel

In FPC, there are three types of fault events identified as SingleFault, ContractFault, and FacilityFault metaclasses in the F-FPC-FT overarching metamodel. The first two fault events are defined for functional fault (c.f. refer AM-FPC-FT overarching metamodel in Subchapter 4.2.7) and the later fault event is defined for structural fault. The SingleFault and ContractFault are generated from transformation method developed in Subchapter 4.2. The FacilityFault in the FPC is generated based on the facilitation of facility to action discussed in previous subchapters. The connections between functional and structural faults based on facilitation are captured by <<facilitation>> stereotype.

The functional faults are transformed as output event in Fault Tree. This is shown by the mapping from SingleFault and ContractFault to OutputEvent. The <<equivalence>> stereotype on the mapping lines means the entity from FPC is mapped exactly onto the entity in Fault Tree. For the structural fault, in the Fault Tree, it is transformed to the basic event. This presents the position of structural fault events are at the lowest level of the Fault Tree that shown by the connection between FacilityFault and BasicEvent. Similar to connection between functional faults and output event, the connection between structural fault and basic event applies <<equivalence>> stereotype to show the exact mapping between entities. In the presentation of Fault Tree, all the basic events are connected by OR-gate to their correspond output events in Fault Tree.

The F-FPC-FT overarching metamodel is the guideline for mapping the transformation of faults defined in the modelled facilitation in FPC to the integrated Fault Tree generated. By using the overarching metamodel as a guideline, the process of transformation which includes structural fault and functional fault are presented in the generated Fault Tree.

5.3.4 Facilitation Application to RMS Case Study

The Activity model and Class model of the RMS are referred to demonstrate the concept of facilitation. By using the transformation method with facilitation, the integration of functional and structural fault events are presented in the generated Fault Tree. The generated Fault Tree will be named integrated Fault Tree. The Fault Tree is then used to analyse the safety and reliability of system through the occurrences of functional and structural fault events.

The modelled facilities in the Class model are identified by which facilitate each of actions and displayed by using the introduced facilitation presentation on the Activity model. The facilities of TMSoS that facilitate each of the modelled actions in Activity model are depicted as in Figure 5.8. Based on the figure, four single types of facilitation are observed. The single type of facilitation is applied to the set (F_{10}, A_4), (F_7, A_7), (F_7, A_8), and (F_7, A_9). In this case, only one facility facilitates one action. The multiple type of facilitation is applied to the six set of (F_1, F_2, F_9, A_1), (F_1, F_3, A_2), (F_1, F_5, A_3), (F_1, F_6, A_5), (F_1, F_7, A_6), and (F_1, F_8, A_{10}). From the set of multiple type facilitation, one can observe multiple facilities facilitate to one action. In an operating system that one can really observe, several functions are facilitated by the same facility. For this case study, the Facility F_1 and Facility F_7 can be seen to facilitate more than one action.

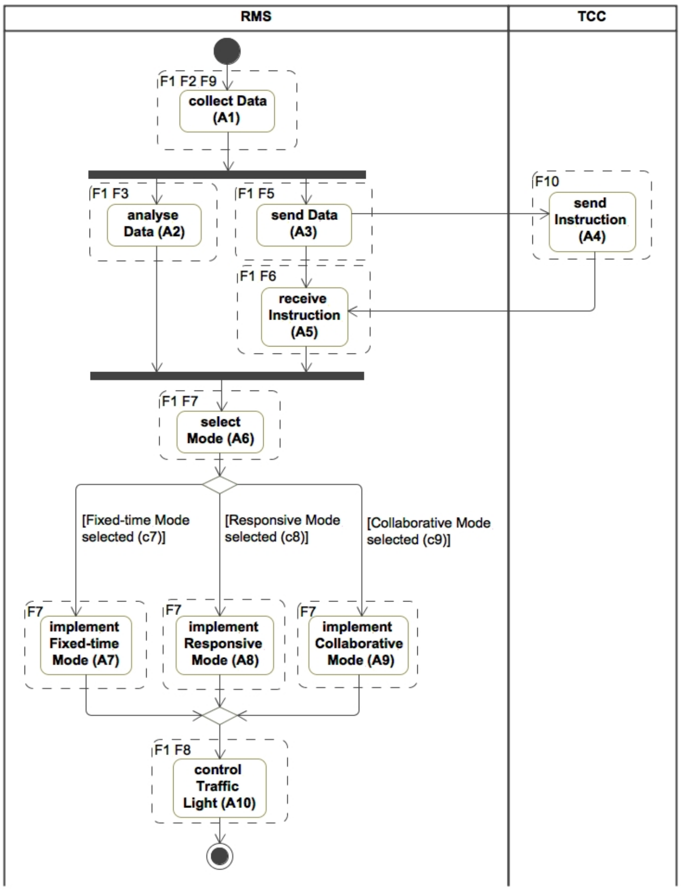

RMS
TCC
F1 F2 F9
collect Data (A1)
F1 F3
analyse Data (A2)
F1 F5
send Data (A3)
F10
send Instruction (A4)
F1 F6
receive Instruction (A5)
F1 F7
select Mode (A6)
[Fixed-time Mode selected (c7)]
[Responsive Mode selected (c8)]
[Collaborative Mode selected (c9)]
F7
implement Fixed-time Mode (A7)
F7
implement Responsive Mode (A8)
F7
implement Collaborative Mode (A9)
F1 F8
control Traffic Light (A10)

Figure 5.8: Facilitation on Activity Model of Ramp Meter System

Since the Activity model can be presented in fault viewpoint on FPC, by using material equivalence, the structural fault can also be presented on the same FPC as depicted in Figure 5.9. At this point, system functions and system components of RMS can be seen in fault of view in FPC. For the contracted fault event $a_{7,8,9}$, as it is composed of more than one functional fault event, for the facilitation application, the contracted event can be designed with an envelope of individual functional fault events. As the same Facility F_7 facilitates Action A_7, Action A_8, and Action A_9, the structural fault event f_7 is attached to outer box that rounded the separated functional fault events a_7, a_8, and a_9. However, the separated functional fault events of the pairs are gathered into a block to preserve the original structure of the whole FPC. The structure of expanded fault event must be grouped as the structure of FPC can turns into the same structure as the sequence of actions modelled from the initial to final nodes. In Activity model, the contracted fault events are traced back to the actions that flow after decision nodes. The separation is performed to assist facilitation (double-dashed-arrow) to each fault event. This gives more meaningful reason if there are different facilities facilitate the actions when implementing facilitation.

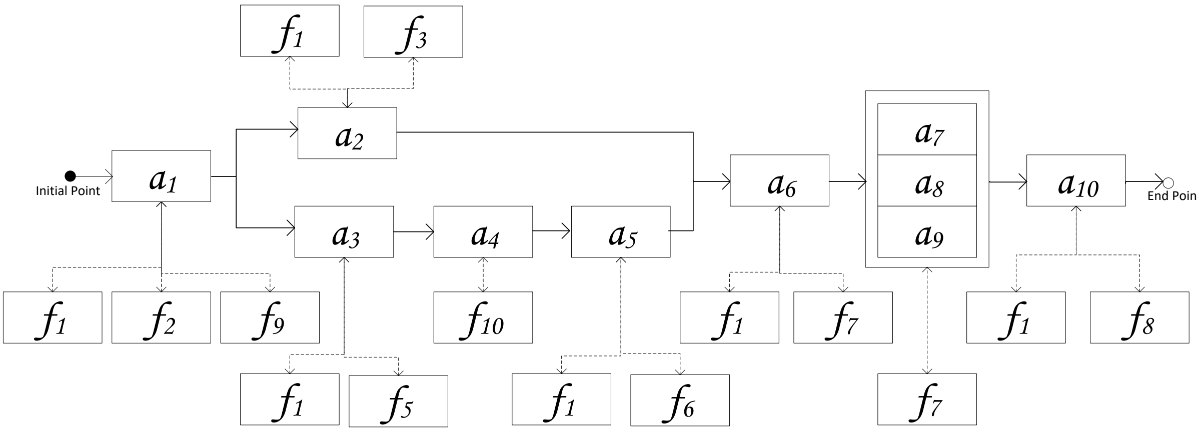


Figure 5.9: FPC with Facilitation of Facilities Implementation

The FPC can be transformed into a Fault Tree by adding the structural fault events as the new basic events in the Fault Tree created in Subchapter 4.2 (c.f. refer Figure 4.11). The generated Fault Tree based on the facilitation called integrated Fault Tree is depicted as in Figure 5.10. The basic events f_7 and f_{10} are connected to the corresponding intermediate events without logic gate. This can be referred to the single type of facilitation which only one facility facilitates one action. This is different from multiple type facilitation in terms of the connection between basic events and intermediate event. The basic events are connected in a group (according to the group of multiple facilitation) by an OR-gate to the corresponding intermediate events. The frequency of the basic events to appear in the Fault Tree can be observed based on how many time of a facility involved in the facilitation to the actions. For example, basic event f_1 appears six times in the integrated Fault Tree.

As the basic events are consisted of structural fault, the probability of failure at each level of the tree up to the top event can be identified. The tree has a similar structure with previous tree (functional Fault Tree) except the minimum cut set of the tree is consisted of structural fault, i.e. all the basic events in integrated Fault Tree are structural fault events. The minimal cut sets are $\{f_1\}$, $\{f_2\}$, $\{f_3\}$, $\{f_5\}$, $\{f_6\}$, $\{f_7\}$, $\{f_8\}$, $\{f_9\}$, and $\{f_{10}\}$.

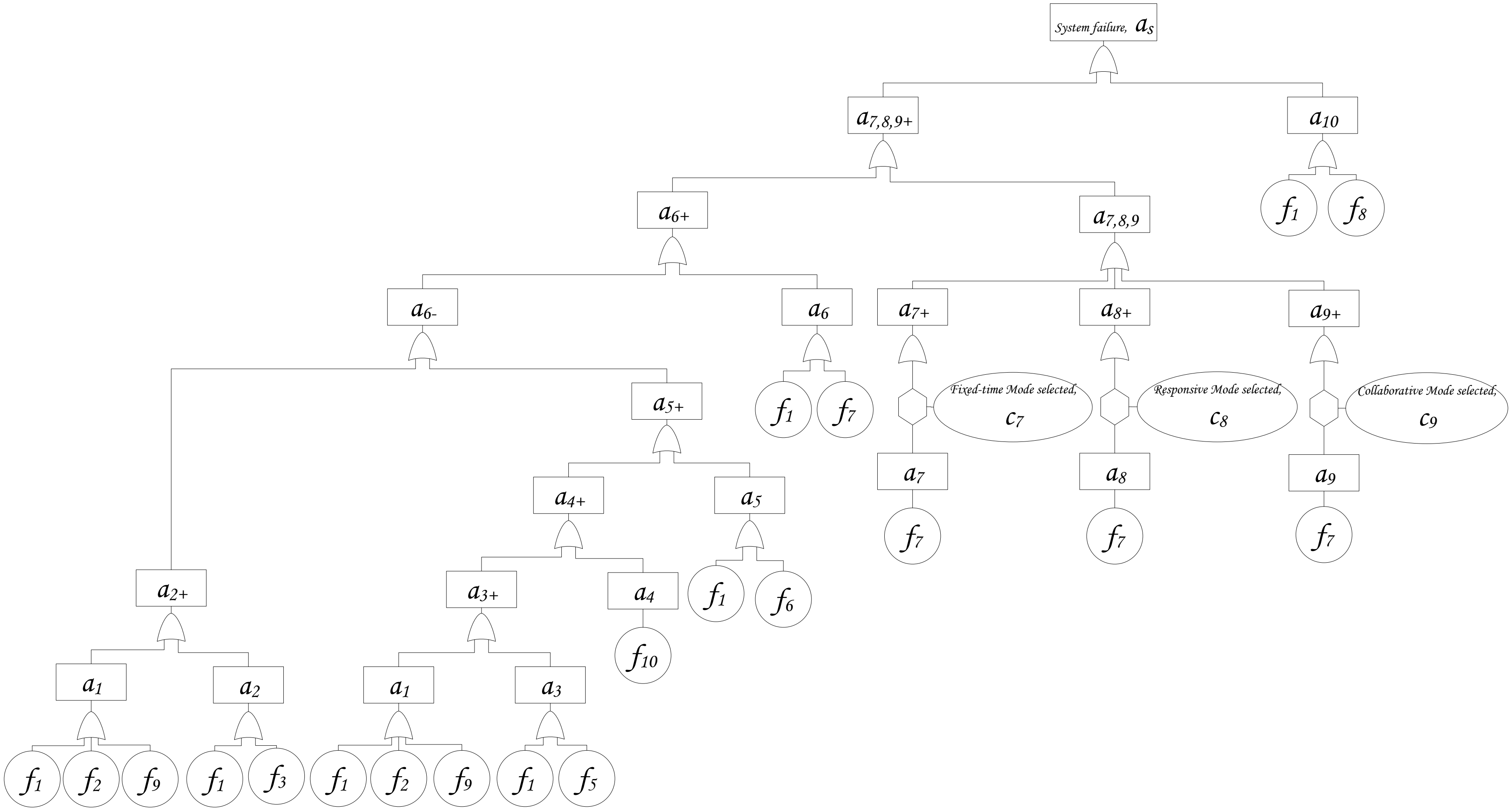


Figure 5.10: RMS Fault Tree transformed from the Fault Propagation Chain in Figure 5.9

5.4 Ownership

At the early stage of system design, at some point, the UML Class model optimises the design aspect with safety analysis aspect. For example, a Class model can provide information of a system for generating integrated Fault Tree (Cepin & Mavko 1999). As the UML Class model represents relation amongst Classes, the faults and failure structure of the Fault Tree is constructed based on causal relationships and failure propagation through the Classes. The fault and failures of facilities and actions embedded in the Fault Tree are derived from Classes that represent facilities and actions that integrated in the Classes. To examine failure from intended design of a system, Cepin et. al. applied logic rules to the information provided for the system such as requirement specification similar to work done in Subchapter 4.2.

System modelling by using UML allows flexibility to explore fault and failure viewpoints of the system in this research. The concept of allocation introduced in SysML can be used in UML in different perspective - Ownership. For system component (facility) modelling by using UML Class, Operations that embed in the Class are owned by Class. In comparison to UML Activity which models sequences of actions of systems, functions in the Operations of a Class are decomposed to a lower detail such as from system to lower level subsystem. For example, "Ramp Meter System Control Unit" (F_1), one of the facilities that facilitate "collect Data" (A_1) action, owns 'control' action (c.f. Figure 3.11). The action that owned by "Ramp Meter System Control Unit2 (F_1) is located in the Operations compartment. However, these actions could not be found in the Activity model earlier (c.f. Figure 3.10). Hence, the ownership of actions represents hierarchy decomposition of actions in subsystem level. For example, nowadays, complex systems are embedded with software that does lower level actions. Similar to system hardware, software has behaviour. The possible behaviour of the software includes computing, synchronising data, and data handling. These lower level function initiates facilities to achieve behaviour at system level.

The actions that owned by Classes can be used to decompose integrated Fault Tree generated previously into elaborated Fault Tree. Instead of continues down the tree with new branches, the fault Tree is elaborated for new separated Fault Trees based on the basic events of structural fault event in the integrated Fault Tree be the top events of the new trees. Therefore, the basic events will be replaced by transfer out event. The actions defined in Operation are used to expand the new developed trees as the intermediate fault events. Taking “Ramp Meter System Control Unit” (F_1) earlier as example, F_1 is transformed as f_1 in the integrated Fault Tree as a basic event. For the decomposition of the basic event f_1, f_1 becomes a transfer out event in the elaborated Fault Tree. Noting that every transfer out event in the original Fault Tree, i.e. in this case integrated Fault Tree, produces another tree that takes it as a top event, i.e. transfer in event. In the new tree, the top event f_1 has a basic event. The Fault Tree depicted as in Figure 5.11(a) is a partial of elaborated Fault Tree with transfer out events of f_1, f_2, and f_9. A note of number of page is enclosed on each transfer out event to determine the location of the details. The details of the transfer out event are continued to the Page 2 depicted as in Figure 5.11(b). In the Page 2, the basic event of f_1 is F_1fails to complete its behaviour which relates to ‘control’ action that owned by F_1.

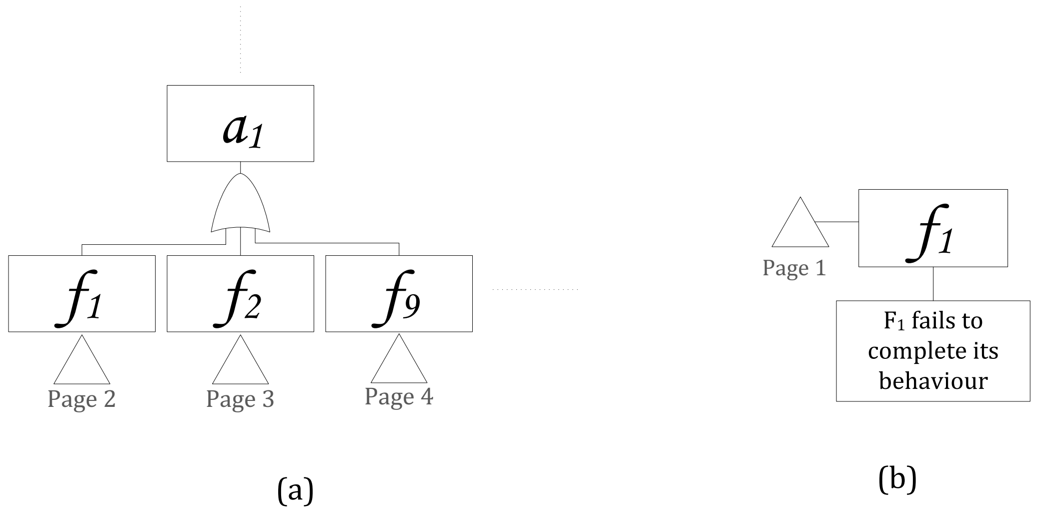


Figure 5.11: (a) Transfer Out Events (b) Transfer In Event

In reality, the actions that owned by facility could be expand and organised such as by using control flow. System architect can continue to model the actions in Activities and apply formal transformation method proposed in Subchapter 4.2 to generate Fault Tree. However, at best of the system architect's knowledge, if the actions could not be modelled in Activities, it could be modelled in different system model such as Sequence Diagram and State Machine Diagram. Consequently, new methods to formally transforming the models to Fault Tree could be explored.

5.5 Summary of Automated Fault Tree Generation from Integrated UML System Models

Generally, the conventional Fault Tree used for analysing safety and reliability of system under development is comprised of functional and structural faults and failures. In the previous chapter, two Fault Trees are generated based on a single perspective of system, i.e. functional and structural, which do not support the conventional Fault Tree. To complement the conventional Fault Tree practiced in the industry, a formal transformation method of integrating functional and structural perspectives of system is proposed in this chapter. The integrated perspectives are based on UML system models, i.e. Activity and Class.

The integration of behavioural (action) and structural (facility) perspectives is initiated by using the concept of facilitation which adopts allocation in SysML. The concept of facilitation is introduced to support multiple facilities that facilitate a function of a system that is not common in SysML practice. By using material equivalence, the facilitation is applied to Activity model that turns to be the backbone of the generated Fault Tree and FPC introduced in the previous chapter which consisted only functional fault event. The facilitation is classified into two types: single type of facilitation and multiple type of facilitation. According to the TMSoS case study demonstration, four single and seven multiple type of facilitation can be observed. One can observe the number of facility(ies) facilitates to each of the actions which can result in the attachment of structural fault events (basic events) to the functional fault events). For the single facilitation, the basic event is attached directly to the output event without any logic gate. For the multiple facilitation, a group of basic event are attached to output event through an OR-gate. The facilities also can be observed in how many times they facilitate the actions in the Activity model which can result in the appearance of the corresponding structural fault events in the integrated Fault Tree. In the integrated Fault Tree presentation, the probability of failure of structural fault event is contributed to the probability of failure of functional fault event. The implementation of the transformation method is supported by a development of the F-FPC-FT overarching metamodel to bridge the metamodels of FPC and Fault Tree.

Based on the transformation method, three notes for the transformed integrated Fault Tree are remarked. The first note is drawn from an operating system. Some system functions are facilitated by the same system component. When they are modelled as functional and structural fault events for the integrated Fault Tree, the same structural fault event can be seen in more than one branch of the tree. This can relate to the frequency of a facility involves in the facilitation.

On the second note, the facilitation of facilities to actions is dependent to the actions and facilities that have been modelled. In some cases, one can observe that functional fault event is rightly attached by one structural fault event in the integrated Fault Tree, i.e. without any logic gate. For this case, the facility can be decomposed. In addition, if facilitation cannot be done due to irrelevance modelled facilities to facilitate actions, a revision of the Class model need to be done. The approach of the concept of facilitation on the system modelling can be improved by profiling. An identification of each Action and Class can be introduced as an attribute which uses combination of number and letter. The additional dashed box of facilitation that wrap around the action can be replaced by specifying the facility identification profile in the action. At the moment, the use of dashed box is remained to give a flexible modelling and demonstration.

The third note is on the separation from contracted fault event into individual functional fault events. The facilitation of facility should be carried onto individual functional fault event. The functional fault events of the contracted event should be represented in individual block to avoid the ignorance of different facilities that might facilitate the functional fault events.

To conclude formal transformation methods developed in this thesis, the concept of ownership is applied. This illustrates a complete cycle of Fault Tree transformation. The current state of approach leaves the Fault Tree with functional fault as the lowest detail. For future work, the functional fault, in the system architecture modelling, the function can further be modelled. This is not limited to Activity but it can be modelled in other types of behavioural diagrams such as in Sequence Diagram and State Machine Diagram. Further for the future work, research on how to transform the behavioural diagrams to Fault Tree.

CHAPTER 6

VERIFICATION OF THE FORMAL TRANSFORMATION METHODS

6.1 Introduction

In this chapter, the formal transformation methods developed in Chapter 4 and Chapter 5 are applied to a Level Crossing Control System (LCCS) extracted from an authorised Railway System case study, for an insight of verifying applicability of the developed transformation methods into real system development. The transformation methods, which have been individually developed based on single and integrated aspects of system function and system component for Fault Tree generation are reorganised for a systematic demonstration. Technically, four types of Fault Tree are generated from the four transformation methods. The four types of generated Fault Tree are functional Fault Tree, integrated Fault Tree, extended Fault Tree (comprises of component Fault Tree), and elaborated Fault Tree. The demonstration of the developed transformation methods begins with transformation from UML Activities to functional Fault Tree generation. The method is supported by application of semantic mapping rules in Subchapter 4.2. The demonstration is followed by an application of the concept of facilitation for the integrated Fault Tree generation which involves UML Classes. As a result, failures of system function and system component are embedded in a Fault Tree. Then, the Classes with Composition relationship are identified. These Classes are transformed for generating extended Fault Tree by extending the existed Fault Tree. Finally, the demonstration of the developed transformation methods through LCCS case study is ended with application of ownership. This demonstrates the ownership of Class (facility) on the attribute (function) in elaborated Fault Tree with transfer out and transfer in fault event.

The remainder of this chapter is structured as follows:

Subchapter 6.2: Railway System Case Study Review

The LCCS is extracted from the authorised Railway System and taken as a case study for demonstrating transformation methods developed in Chapter 4 and Chapter 5. A technical description of the particular Railway System is presented to accommodate a holistic view of the system under analyse for the demonstration.

Subchapter 6.3: System Models: Level Crossing Control System

In this subchapter, system models of the LCCS which modelled in UML are presented based on the technical description in Subchapter 6.2. Use Case model, Activity model, and Class model are selected for presenting the architecture design of LCCS.

Subchapter 6.4: Application of Formal Transformation Method to Level Crossing Control System

The application of four transformation methods developed in Chapter 4 and Chapter 5 to the LCCS is demonstrated. The four transformation methods are demonstrated in an order as mentioned in the first paragraph of Subchapter 6.1. The four types of Fault Tree generation are expected from the application of each transformation method. The hierarchical elements in Fault Tree can be observed through fault events.

Subchapter 6.5: Comparative Analysis

The developed transformation methods are compared with methods developed in other researches by which transformation aspect from system models to Fault Tree generation. The comparison of the developed transformation methods is categorised into two sets. In the first set, the transformation from system models to Fault Tree is concerned. In the second set, the formalism of the transformation methods is concerned.

Subchapter 6.6: Summary of Verification of the Developed Formal Transformation Methods

A trial run of the developed transformation methods on Railway System case study for verifying the transformation methods is summarised.

6.2 Railway System Case Study Review

An authorised Railway System in German is taken as a case study to demonstrate formal transformation from UML models to Fault Tree. The Railway System is fully under control of a German railway company, Deutsche Bahn. The Deutsche Bahn works on the techniques of controlling LCCS for the system. Decentralised system and radio based LCCS are two systems that concern the control techniques. For that reason, the on route signals and sensors as in conventional system are replaced by radio communication and software computations installed on the LCCS and trains. The technology implemented on the Railway System offers cheaper and more flexible solutions by reducing involvement of many systems. Nevertheless, the safety critical functionality of the system of systems (SoS) also shifts from hardware to software.

The respective Railway System has been studied by many practitioners and academicians (Reif et al. 2000), (Schellhorn et al. 2002), (Xiang et al. 2004), (Xiang et al. 2005). The studies contribute to the extension of knowledge of the Railway System either technically or theoretically. The narratives for the LCCS case study have been collectively gathered and modified to achieve a relevant sequence of the system flow. It is presented in next paragraph and the image of Railway System is illustrated in Figure 6.1.

The train computes the position where it has to send a signal to the LCCS. The signal is sent by the train to the LCCS in order to check the status for the train to passing the level crossing. The signal has to be sent shortly before the train arrives at the latest braking point which is the possible position of the train to stop before the level crossing if it has to, e.g. due to safety reasons. After the LCCS receives the command, it switches on yellow traffic lights from green traffic lights as indication to the vehicles on the roadsides to be ready to stop before entering the level crossing. The indication from the traffic lights also applies to pedestrian on the roadsides which is assumed to be the same as the vehicles. Then, the traffic lights turn to red. After the traffic lights turn to red, barriers start closing. When the barriers start closing, an assumption of unoccupied level crossing with vehicles can be made since the vehicles receive indication to stop before entering the level crossing, e.g. traffic lights are on red before the barriers start closing. The closed barriers show the status of the level crossing is safe for a certain period of time for the train to cross the level crossing. One of two feedback signals will be sent by the LCCS to the train at a time. One, a 'release' feedback signal is sent to indicate that the train may pass the level crossing. Second, a 'stop' feedback signal is sent to indicate unsafe crossing which also initiated by computation and communication of the LCCS and the train. The train passes or breaks on the level crossing depending to the signal received from the LCCS. When the train needs to stop, it will stop for a while as LCCS the level crossing clearing the level crossing from vehicles and then crossing the level crossing. Despite of mobility control, the LCCS periodically performs self-diagnosis. The LCCS automatically informs central office about problem detected, e.g. defect. The central office is responsible for repairing and also providing route descriptions to trains. Basically, the information provided to the trains is positions indication of the LCCS and maximum speed on route, e.g. route with maximum speed of 160 km/h. After the train has passed the crossing, it sends back a 'passed' signal to the LCCS which allows the LCCS to switch back to its initial state; traffic lights are green and barriers are open. If there is no signal is being received in the safe period, the LCCS waits for some minutes before it switches to the initial state to protect vehicles against endless waiting. This is the unsafe period for the train to cross the level crossing.

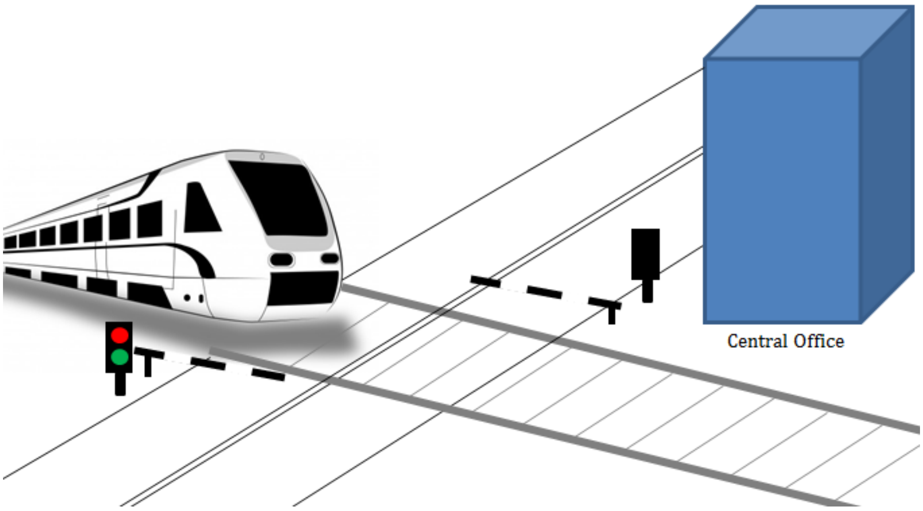


Figure 6.1: Railway Crossing System (Reif et al. 2000), (Schellhorn et al. 2002), (Xiang et al. 2004), (Xiang et al. 2005)

6.3 Systems Models: Level Crossing Control System

In this subchapter, the behaviour and structure models of the LCCS are presented. The system behaviour of LCCS is modelled in UML Use Case and UML Activity, and the system structure of LCCS is modelled in Class. In the presentation of system models of LCCS, elaborated Use Case description is also included in between the presentation of Use Cases and Activities.

For the Use Case presentation, two levels of Use Cases of the LCCS are modelled. In the first level Use Case as depicted in Figure 6.2, which known as high level Use Case, three Use Cases are identified and presented in eclipse shape. Three constituent systems (CSs) that interact with the LCCS are identified as Actors that presented as stick man. The first Use Case, “allow Crossing”, involves interaction between the LCCS and two of the Actors, i.e. Train and Vehicle. These CSs interact with the LCCS for getting permission to cross the level crossing. The second Use Case, “exchange Information”, presents interaction between the focused system, LCCS, and the three CSs, i.e. Train, Vehicle, and Central Office. All of the Actors interact with the LCCS by exchanging information. The third Use Case, “self-diagnosis”, is associated to Central Office. However, the indirect interaction between the LCCS and the Central Office through the third Use Case can be further explained in lower level Use Case model. Based on the presentation of the high level Use Case, one can observe three operational requirements of the LCCS can be determined according to the identified Use Cases.

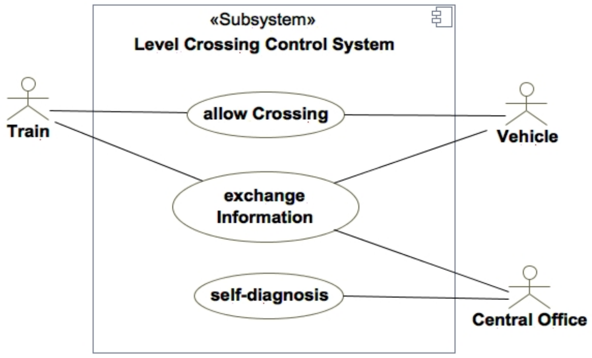


Figure 6.2: High Level Use Case Model of the Level Crossing Control System

In the second Use Case depicted as in Figure 6.3, one can observe the expansion of the high-level Use Case. The "allow Crossing" Use Case with the association to Train and Vehicle is remained as it does not need any Use Case elaboration. The "exchange Information" Use Case is elaborated into three distinct lower level Use Cases using <<include>> relationship. These distinct lower level Use Cases include: (i) "report Problem", (ii) "transmit Signal", and (iii) "receive Signal". From the elaboration result, the "exchange Information" Use Case is indirectly connected to Train and Central Office. For example, the "report problem" Use Case is connected to the Central Office. This shows the relationship between LCCS and Central Office is to report problem to Central Office. The "receive Signal" and "transmit Signal" Use Case can be further elaborated as additional behaviour under specific conditions, i.e. safe and unsafe crossing. For example, "transmit Traffic Signal" and "transmit 'Release' Signal" Use Case are the extending Use Cases to "transmit Signal" Use Case. Technically, with the specific conditions and specific types of signal, LCCS transmit traffic signal to the vehicle and transmit 'release' signal to the train. The "self-diagnosis" Use Cases, can be elaborated into distinct lower level Use Cases, "detect Problem", using <<extend>> relationship. As the LCCS performs self-diagnosis and report to the central office of any detected problem, the "detect Problem" Use Case and "report Problem" can be connected by <<extend>> relationship and "report Problem" can hold the interaction between "self-diagnosis" Use Case and Central Office. Thus, this supports the indirect relationship between "self-diagnosis" Use Case and Central Office in the high level Use Case. At this level of Use Case information, a comprehensive information of LCCS functional requirements are specified.

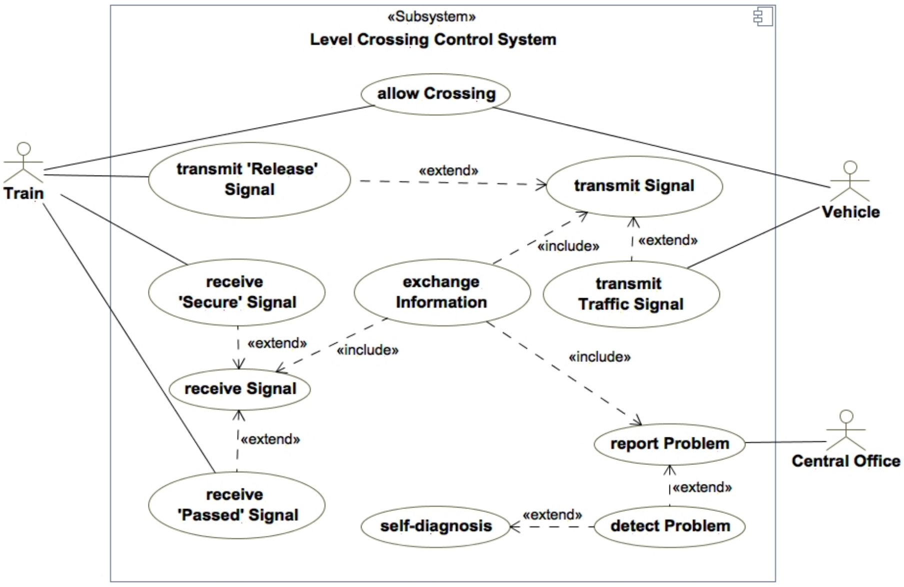


Figure 6.3: Use Case Model of the Level Crossing Control System

From the presented Use Cases, three Use Case descriptions can be formed to describe the specific behaviours of system in detail. The behaviour of the LCCS and the interaction of the LCCS with other CSs which accommodate its behaviour can be identified. The Use Case description depicted as in Table 6.1 describes the "allow Crossing" Use Case process of the responsibility of LCCS to the level crossing. The LCCS controls either train or vehicle by which should be on the level crossing. For the safety reason, one of them, i.e. train or vehicle, can enter the level crossing while the other stops from entering the level crossing at a certain time. There are eight activities for basic flows including four alternative flows are identified for LCCS to control the level crossing. The LCCS allows train to enter the level crossing by sending ‘release’ feedback signal. This means the train is safe to cross the level crossing. When the level crossing is unsafe for the train to enter, the LCCS will send ‘stop’ feedback signal and the train has to pull the brake to stop. The LCCS allows vehicle to enter the level crossing by controlling the traffic lights to switch on green and the barriers to open. In the case where the level crossing has been passed by the train, the LCCS needs to receive ‘passed’ signal from train or wait for several time before allows the vehicle and pedestrian to enter the level crossing. The LCCS resumes to the initial state at the end of the process.

Table 6.1: “allow Crossing” Use Case Description of Level Crossing Control System

Use Case Name	allow Crossing
Description	LCCS is controlling the position of train and vehicles on the level crossing.
Actors	Train and Vehicle
Pre-conditions	1. LCCS controls traffic light to switch on green. 2. LCCS controls barriers to open 3. Vehicles enter the level crossing.
Post-conditions	LCCS resumes to the initial state.
List of Activities for Basic Flow	1. LCCS receives ‘secure’ signal from the train. 2. LCCS controls traffic light to switch on yellow. 3. LCCS controls traffic light to switch on red. 4. LCCS controls barriers to start closing. 5. LCCS sends feedback signal to the train. 6. LCCS receives or not receive ‘passed’ signal from the train. 7. LCCS opens the barriers. 8. LCCS controls the traffic lights to switch back on green.
Alternative Flow	5a. LCCS sends ‘release’ signal to the train. 5b. LCCS sends ‘stop’ signal to the train. 6a. LCCS prevents counter from counting. 6b. LCCS enables counter for counting.

Next, in Table 6.2, the “exchange Information” Use Case description is presented for describing the exchange information between the LCCS and other CSs. For this Use Case, four pre-conditions and one post-condition are required in the execution of the “exchange Information” Use Case. From the technical description provided in Subchapter

6.2, the “exchange Information” Use Case process can be divided into two paths. First, the LCCS is exchanging information with the train and giving information to the vehicle in managing which one of them can enter to and stop from entering the level crossing. In the first path, the LCCS requires exchanging information with the train. The vehicle has to be aware of the signal provided by the LCCS based on the exchanging information with the train. Two alternative flows are identified for the feedback signal that has to be sent by the LCCS to the train. In the second path, the LCCS interacts only with central office by reporting any detected problem. These two paths are executed independently to each other.

Table 6.2: “exchange Information” Use Case Descriptions of Level Crossing Control System

Use Case Name	exchange Information
Description	LCCS is exchanging information with train and giving information to vehicle. LCCS also communicating with the central office to report any defect or problem detected during the self-diagnosis process.
Actors	Train, Vehicle, and Central Office
Pre-conditions	1. Train identifies distance to send ‘secure’ signal to LCCS. 2. LCCS controls traffic light to switch on green. 3. LCCS controls barriers to open 4. LCCS executes self-diagnosis.
Post-conditions	LCCS waits signal from the train.
List of Activities for Basic Flow	1. LCCS receives ‘secure’ signal from the train. 2. LCCS controls traffic lights to switch on yellow. 3. LCCS controls traffic lights to switch on red. 4. LCCS sends feedback signal to the train. 5. Train receives feedback signal from the LCCS. 6. Train break or pass the level crossing. 7. Train sends ‘passed’ signal to the LCCS. 8. LCCS receives ‘passed’ signal from the train. 9. LCCS reports problem to central office. 10. Central office receives problem reported by the LCCS.
Alternative Flow	4a. LCCS sends ‘release’ signal to the train. 4b. LCCS sends ‘stop’ signal to the train.

The “self-diagnosis” Use Case description presented in Table 6.3 presents self-diagnosis process executed by LCCS. In the self-diagnosis process, there is involvement of central office. Self-diagnosis has a short process with only two basic activity flows in the description. This short process does not require any pre-conditions and alternative flow. During the execution of self-diagnosis, LCCS detects problem which has been occurred and reports the problem to the central office. The central office, i.e. operator, will then take action to solve the reported problem. The execution of self-diagnosis process will stop at that point and the LCCS resumes the self-diagnosis process again.

Table 6.3: “self-diagnosis” Use Case Description

Use Case Name	self-diagnosis
Description	LCCS is executing self-diagnosis.
Actors	Central Office
Pre-conditions	None
Post-conditions	LCCS resumes the self-diagnosis.
List of Activities for Basic Flow	1. LCCS detects problem. 2. LCCS reports problem to the central office.
Alternative Flow	None

According to the Use Case descriptions, one can observe some of the activities for basic flows and alternative flows in a Use Case description can be found again in other Use Case description. For example, “LCCS controls traffic lights to switch on red” can be found in “allow Crossing” Use Case description and “exchange Information” Use Case description. This shows the overlap activities that required during execution of the system behaviour that serve different functions. Furthermore, the particular activity, i.e. LCCS controls traffic lights to switch on red, is a kind of information given by the LCCS to vehicle and pedestrian and permission given by the LCCS to the vehicle and pedestrian to enter level crossing. The generated Use Case descriptions are used to support the modelling of activities of the Railway System in UML.

The activities is modelled with three subsystems; (i) LCCS, (ii) Train, and (iii) Central Office, as depicted in Figure 6.4. The Vehicle which is another subsystem is not considered in the modelling as the Railway System does not require any input from them. The actions are modelled with an initial state where two traffic lights are switched to green and two barriers are opened. A pair of a traffic light and a barrier is located on each roadside. At the initial state, the vehicles are safe and allowed to cross the level crossing with the indicators given by the traffic lights and barriers. From the entry point, the path flows into a fork node before flowing into two concurrent series of actions. These actions flow in two different paths. These paths are ended with one activity final node flows from a join node. The first path, on the left side, is crossing on the level crossing activities path. The activities path flows into three actions in series "receive 'Secure' Signal" (A_1), "control Traffic Light" (A_2), and "control Barrier" (A_3) where the traffic lights are switching to red and barriers are closing for safe level crossing for the Train. Once the decision for the safe level crossing is made, the control flow continues into a pair of decision and merge nodes in which two alternative paths are modelled. Each path involves an action of instruction feedback, i.e. "send 'Release' Feedback" (A_4) with "safe crossing" (c_4), and "send 'Stop' Feedback" (A_5) with "unsafe crossing" (c_5). With only one instruction feedback being sent at any one time to the external function, "send 'Passed' Signal" (A_6), owned by the Train. And the series flows back to the LCCS, where LCCS shall "receive 'Passed' Signal" (A_7). However, the LCCS, in other possibility, may not receive the signal. Therefore, the control flow leads into the second pair of decision and merge nodes in which two alternative paths. Each path involves an action of designated counter, i.e. "prevent Counter" (A_8) with "signal receive" (c_8), and "enable Counter" (A_9) with "signal not receive" (c_9). Any path of the series taken leads LCCS to be in initial state again. The second path, on the right side is self-diagnosis activities path. This path is the shortest that involves a series of three actions. The activities path is started with "detect Problem" (A_{10}) that flows into "report Problem" (A_{11}) action. Then, the action leads to an external function owned by Central Office, "receive Report" (A_{12}) before the path ends.

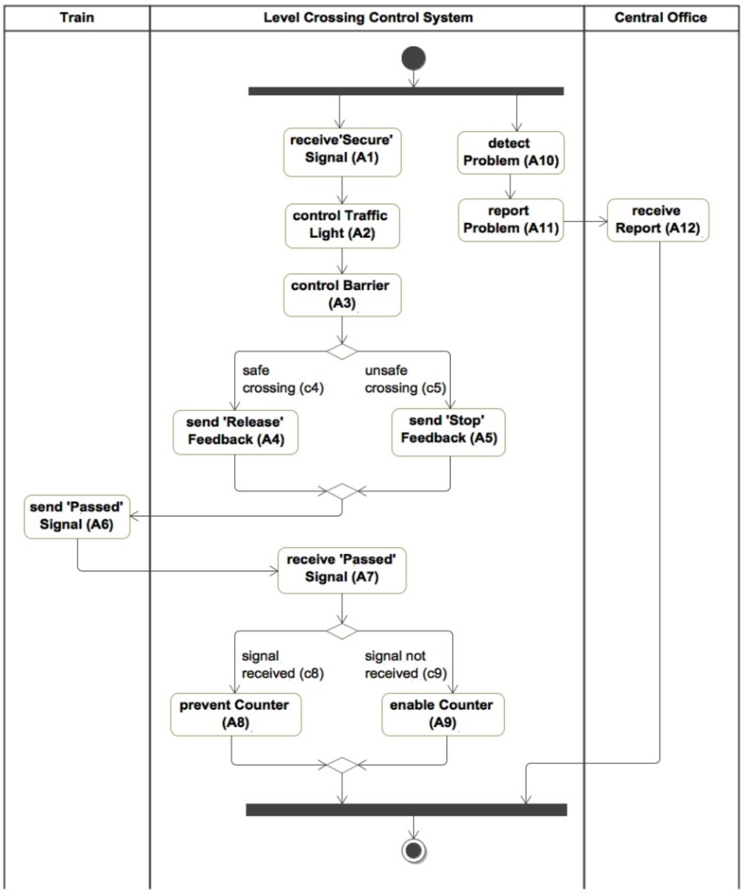


Figure 6.4: Activity Model of the Level Crossing Control System

The Railway System is modelled in UML Classes with 20 Classes as depicted in Figure 6.5. The first Class, “LCCS Control Unit” (F_1), is associated to six Classes, e.g. “Traffic Light” (F_2), “Barrier” (F_3), “LCCS Transceiver” (F_4), “Counter” (F_7), “Self-diagnosis System” (F_8), and “Level Crossing” (F_9), as presenting direct relationship with the Classes during its initial state and normal operation. As a user of the level crossing at the initial state, “Vehicle” (F_{10}) is associated to three of the Classes, “Traffic Light” (F_2), “Barrier” (F_3), and “Level Crossing” (F_9). Another user of level crossing, “Train” (F_{11}), is also associated to the “Level Crossing” (F_9). The “Train” (F_{11}) has Composition relationship with “Train Transceiver” (F_{12}) and “Train Driver” (F_{15}) as the owner Class. The central office forms a Class as “Central Office (CO)” (F_{16}) which also decomposed, with Composition relationship, by “Operator” (F_{17}) and “CO Transceiver” (F_{18}). In both of (F_{11}) and (F_{16}), human is modelled as a system which to operate them, i.e. “Train Driver” (F_{15}) and “Operator” (F_{17}). All of the modelled transceivers, e.g. “LCCS Transceiver” (F_4), “Train Transceiver” (F_{12}), and “CO Transceiver” (F_{18}), are decomposed by a pair of receiver and transmitter, i.e. “LCCS Transmitter” (F_5) and “LCCS Receiver” (F_6) are owned by “LCCS Transceiver” (F_4), “Train Transmitter” (F_{13}) and “Train Receiver” (F_{14}) are owned by “Train Transceiver” (F_{12}), and “CO Transmitter” (F_{19}) and “CO Receiver” (F_{20}) are owned by “CO Transceiver” (F_{18}). Since the Railway System is a computerised system, the communication is made through the transceivers.

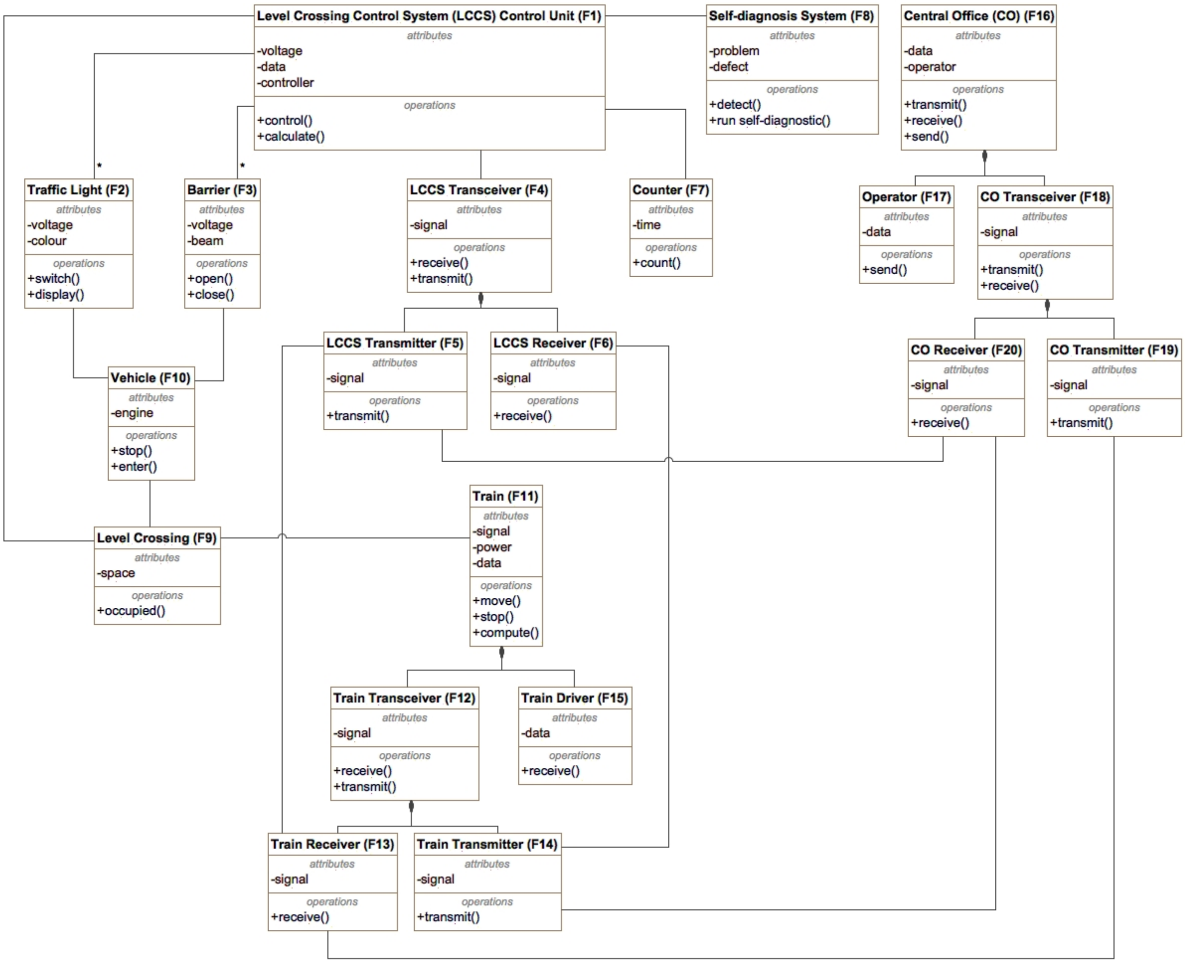


Figure 6.5: Class Model of the Level Crossing Control System

6.4 Application of Formal Transformation Methods to Level Crossing Control System

In this subchapter, the four developed transformation methods are applied to demonstrate transformation from UML systems model to Fault Tree generation. The demonstration is led by formal transformation method from Activities to Fault Tree. The application of the method will produce a functional Fault Tree which preserves the relational structure of system functionality. Then, the concept of facilitation is applied to integrate system functions and system components aspect in the Fault Tree which will produce integrated Fault Tree. The integrated Fault Tree is elaborated with the transformation method from Classes to Fault Tree. The transformation is ended with the concept of ownership with the implementation of transfer event of Fault Tree. However, the limitation of the demonstration is the details of probability of failure which is not publicly available.

6.4.1 Activities to Fault Tree

In this subchapter, the formal transformation method developed in Subchapter 4.2 is applied to the Activity model of LCCS in Figure 6.4. By using the formal transformation method, the useful information in the identification of behavioural faults as well as inferring a fault structure in FPC presentation is demonstrated. The fault structure in the FPC presentation is then transformed into a Fault Tree of LCCS functionality (behavioural). The generated functional Fault Tree of LCCS is then used to analyse the original Activity model of LCCS.

A set of fault events is defined as the negation of the proposition of actions, $a_i \overset{\text{def}}{=} \neg p_i$, modelled in the Activity model of LCCS. Based on the first control flow of Activity model of LCCS and semantic mapping rules (b) in Table 4.1, Initial Point of the FPC bifurcates into two chains with two separate fault events, $a_1$and a_{10}, as the initial fault events of each chain. These two chains represent two different failure functions of LCCS. The first chain denoted by initial fault event $a_1$is formed by a series of eight other fault event including two contracted fault events, $(a_1 \rightarrow a_2) \wedge (a_2 \rightarrow a_3) \wedge (a_3 \rightarrow a_{4,5}) \wedge (a_{4,5} \rightarrow a_6) \wedge (a_6 \rightarrow a_7) \wedge (a_7 \rightarrow a_{8,9})$. This is the chain of failure to control level crossing system. The two contracted events are $a_{4,5}$ and $a_{8,9}$. Both contracted fault events are conjunction of two fault events, $a_{4,5} = a_4 \wedge a_5$ and $a_{8,9} = a_8 \wedge a_9$. The other chain of FPC has three single fault events (non-contracted fault events) connected in a series, $(a_{10} \rightarrow a_{11}) \wedge (a_{11} \rightarrow a_{12})$. This is the chain of failure self-diagnosis. Grouping all of the unitary implications together, a complete FPC is generated and depicted graphically in Figure 6.6. The two chains of FPC, based on semantic mapping rule (c), are merged together into a single chain that leads to the End Point. This marks the ends of the FPC and map to the Initial Node and End Node of Activity model of LCCS.

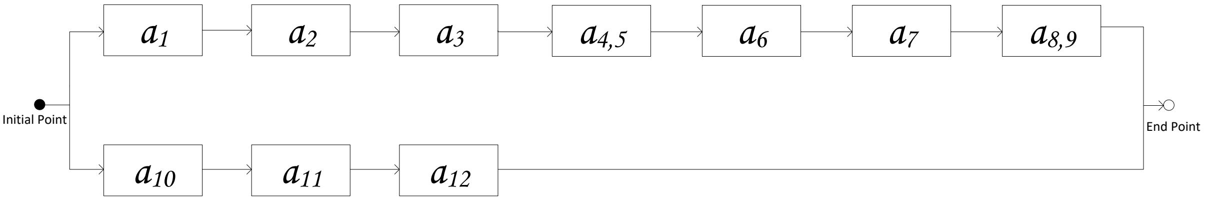


Figure 6.6: Fault Propagation Chain of Level Crossing Control System

Next, the developed FPC is transformed to a Fault Tree using the established formal transformation method. Without going through the detailed mathematical derivation, the Fault Tree can be obtained by elaborating the relation in (4.2.15) for the bifurcation and convergence of the two parallel chains, $(a_1 \rightarrow a_2) \wedge (a_2 \rightarrow a_3) \wedge (a_3 \rightarrow a_{4,5}) \wedge (a_{4,5} \rightarrow a_6) \wedge (a_6 \rightarrow a_7) \wedge (a_7 \rightarrow a_{8,9})$ and $(a_{10} \rightarrow a_{11}) \wedge (a_{11} \rightarrow a_{12})$. For the two contracted fault events in the middle of the first chain, the relation in (4.2.14) is elaborated for two exclusive chains of each contracted fault event, $a_{4,5}$ and $a_{8,9}$. The relation in (4.2.15) is elaborated again to integrate all of the fault events to obtain the functional Fault Tree of LCCS as depicted in Figure 6.7. There are nine fictitious fault events, $\{a_{2+}\}$, $\{a_{3+}\}$, $\{a_{4,5+}\}$, $\{a_{6+}\}$, $\{a_{7+}\}$, $\{a_{8,9+}\}$, $\{a_{10+}\}$, $\{a_{11+}\}$, $\{a_{12+}\}$; twelve basic events, $\{a_1\}$, $\{a_2\}$, $\{a_3\}$, $\{a_4\}$, $\{a_5\}$, $\{a_6\}$, $\{a_7\}$, $\{a_8\}$, $\{a_9\}$, $\{a_{10}\}$, $\{a_{11}\}$, $\{a_{12}\}$; and four conditional events, $\{c_4\}$, $\{c_5\}$, $\{c_8\}$, $\{c_9\}$; are captured in the Fault Tree. The top event of the functional Fault Tree of LCCS, system failure, a_s, is defined as the system failing to complete the intended LCCS behaviour as modelled in the Activity model of LCCS.

Based on the structure of the functional Fault Tree of LCCS, qualitative analyses can be done. Every system function, as modelled by actions, becomes a basic event in the behavioural Fault Tree of LCCS after the transformations as listed in the previous paragraph. As subject to the basic events, the minimal cut set of the Fault Tree are $\{a_1\}$, $\{a_2\}$, $\{a_3\}$, $\{a_4, c_4\}$, $\{a_5, c_5\}$, $\{a_6\}$, $\{a_7\}$, $\{a_8, c_8\}$, $\{a_9, c_9\}$, $\{a_{10}\}$, $\{a_{11}\}$, and $\{a_{12}\}$. The single event of minimal cut sets contributes to the occurrence of the top event, a_s, and align with the formal interpretation of the Activity model based on the UML specification, where failed execution of an action stops the control flow. In addition, the fault events which associated by conditional events, a_4, a_5, a_8, and a_9, do not individually lead to system failure unless their corresponding conditions, c_4, c_5, c_8, and c_9, are met. This also aligns with the Activity model in which the two alternative paths of the pairs of a_4, a_5 and a_8, a_9 are exclusive, i.e. 'release' or 'stop' feedback is sent (first pair) and 'passed' signal is received or not received (second pair) happened at a time.

There are two fault events, a_6 and a_{12}, of other CSs that contribute to the failure of LCCS are identified. The fault event a_6 is belonged to Train and the fault event a_{12} is belonged to Control Office. Tracing the fault event a_6 and its surrounding Fault Tree structure to the original functional design, can obviously see if the Train fails to "send 'Passed' Signal" to LCCS, this could cause LCCS fails to function, i.e. return to initial state; control traffic lights to turn green and control barriers to open. However, with the sufficient design intent of functionality of LCCS modelled in UML Activity, if the 'Passed' signal is not received by LCCS, the function of LCCS is return to initial state after some minutes to allow vehicles and pedestrians on the roadsides to cross the level crossing. However, it is technically different with fault event a_{12}, where when any of the LCCS functions is fail but Control Office fails to "receive Report", the self-diagnosis function of LCCS is failed. Supposedly, any failure of LCCS can be detected by the self-diagnosis function.

The functionality of the Railway System, based on the technical description, can be modelled in several ways. In this thesis, as LCCS is selected as the focused system, by using UML Activity, the system functionality is modelled in the viewpoint of LCCS. In system engineering, a system that is designed with less human-based control (replaced by computer-based control) can reduce potentiality of a system to break down that caused by human error. The computerised system is also easy to control and maintain. However, putting many tasks on a system, the system must go through extensive design and safety analysis.

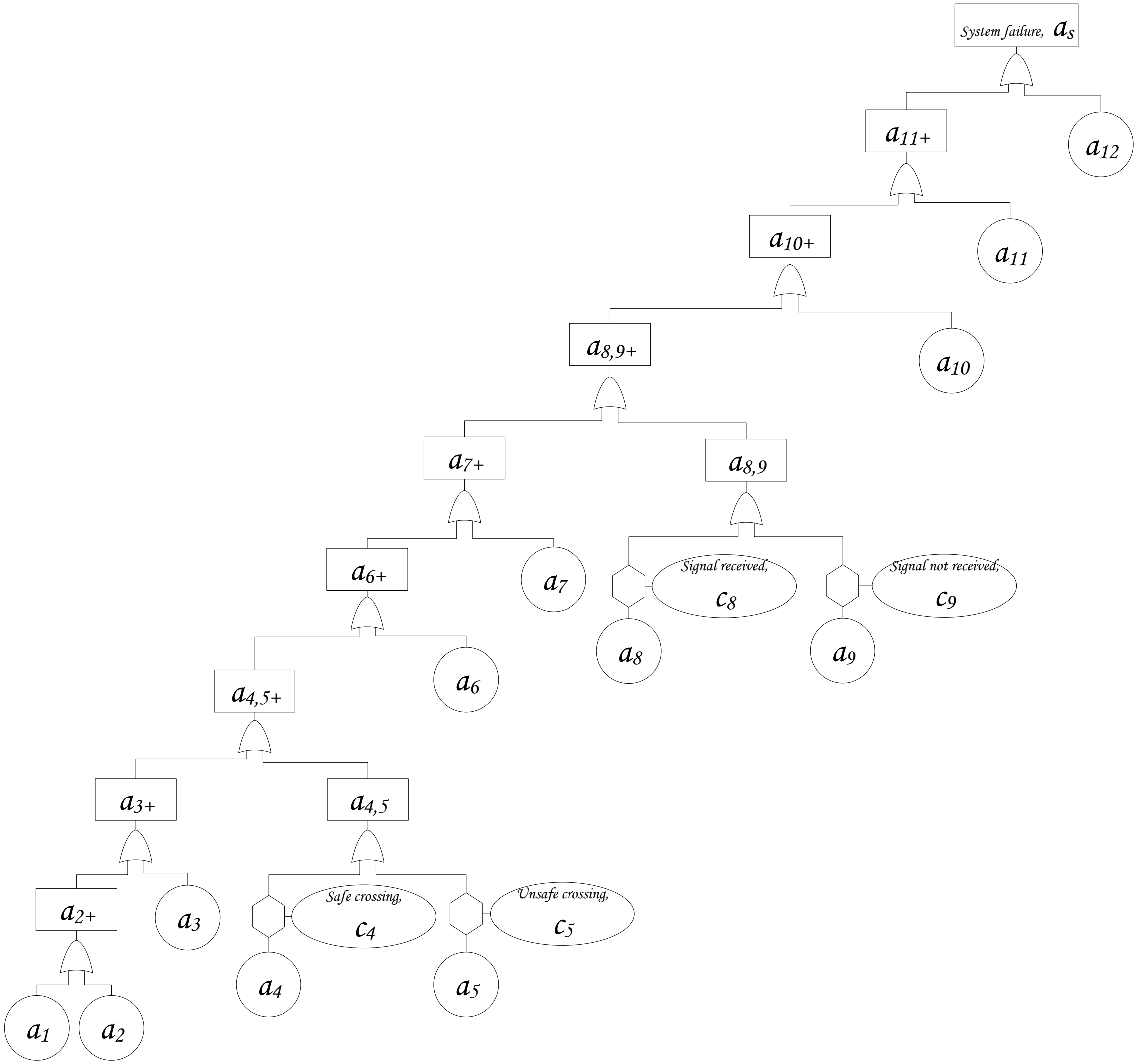


Figure 6.7: Functional Fault Tree of Level Crossing Control System

6.4.2 Transformation Method with Facilitation for Integrated Fault Tree

This subchapter presents an integrated Fault Tree with behavioural and structural failures of LCCS by applying the concept of facilitation to Activity model in based on Class model as depicted in Figure 6.8. The facilitation is observed based the types and how many times a facility involves in the facilitation. The multiple type of facilitation is applied to the set (F_1, F_6, F_9, A_1), (F_1, F_2, A_2), (F_1, F_3, A_3), (F_5, F_9, A_4), (F_5, F_9, A_5), $(F_9, F_{12}, F_{15}, A_6)$, (F_6, F_9, A_7), (F_1, F_7, F_9, A_8), (F_1, F_7, F_9, A_9), (F_1, F_8, A_{10}), (F_1, F_5, A_{11}), and (F_{17}, F_{20}, A_{12}). One can observe that each of the action is facilitated by multiple facilities. The multiple type of facilitation can be seen through all the 12 actions in the Activity model, which leave none of single type of facilitation in the demonstration.

The frequent of facilitation reflects the availability of facility that needs to facilitate the execution of the actions. Based on the Figure 6.8, seven facilities appear one time and five facilities appear more than one time for the facilitation. The seven facilities that shown up for one time in the facilitation are F_2, F_3, F_8, F_{12}, F_{15}, F_{17}, and F_{20}. However, these facilities are shown up as a group that facilitate the corresponding actions. The five facilities that shown up more than one time in the facilitation are F_1, F_5, F_6, F_7, and F_9. One can identify each of the facility to facilitate more than one action. For example, F_1 appears seven times in the facilitation. The "Level Crossing Control System (LCCS) Control Unit" (F_1) is the central system in the LCCS. Being the central of computerised system, the control unit directs the operation of the system and other systems that connected to it by informing the computer's memory, arithmetic unit, logic unit, and input and output devices. Therefore, most of the functions of the LCCS, instead of govern by the respective facilities, the functions are also reliant on the control unit. The facilities of a group that facilitate an action is not necessarily have to be with the same group or even can be alone to facilitate another action.

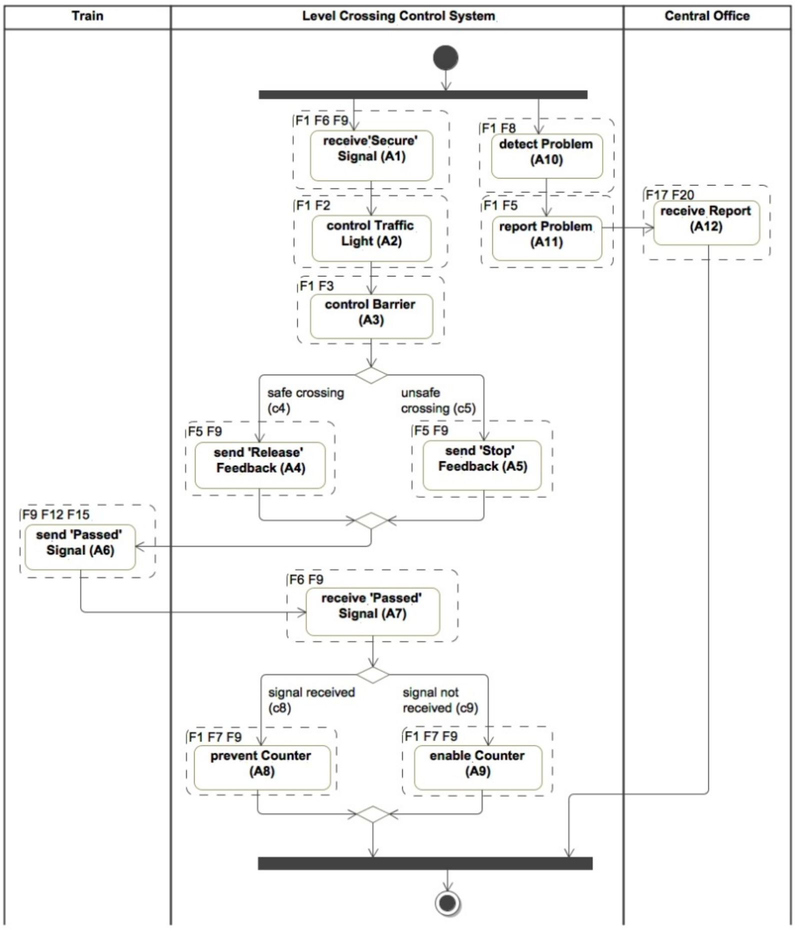


Figure 6.8: Facilitation on Activity Model of Level Crossing Control System

In the FPC, with the application of facilitation, the useful information of the identification of the system component that cause functional faults as well as inferring a fault structure is demonstrated. The FPC is embedded with functional (action) and structural (facility) fault events. Another set of fault events of facilities is defined as the negation of the proposition of facilities, $f_i \overset{\text{def}}{=} \neg c_i$, modelled in the Classes model of LCCS. The type of facilitation in the FPC is expected to be the same as in the previous facilitation on Activity model.

In the application of the concept, the contracted event in FPC is expanded for separating fault events to be in individual presentation to assist the facilitation process. This also applied to fault events of the contracted events, a_4, a_5, a_8, and a_9, which facilitated by multiple facilities. In addition, both fault events of a contracted event are facilitated by the same facilities, e.g. a_4 and a_5 are facilitated by f_5 andf_9. By using the relation in (4.2.7), the two contracted functional fault events, $a_{4,5}$ and $a_{8,9}$, can be presented as two pairs of two individual functional fault events, $a_{4,5} = a_4 \wedge a_5$ and $a_{8,9} = a_8 \wedge a_9$, in the FPC. The structure of expanded fault event must be grouped as the structure of FPC can turns into the same structure as the sequence of actions modelled from the pair of fork and join nodes, i.e. bifurcate into multiple chains. The new FPC, with expansion of contracted functional fault events and facilitation of structural fault events is depicted in Figure 6.9.

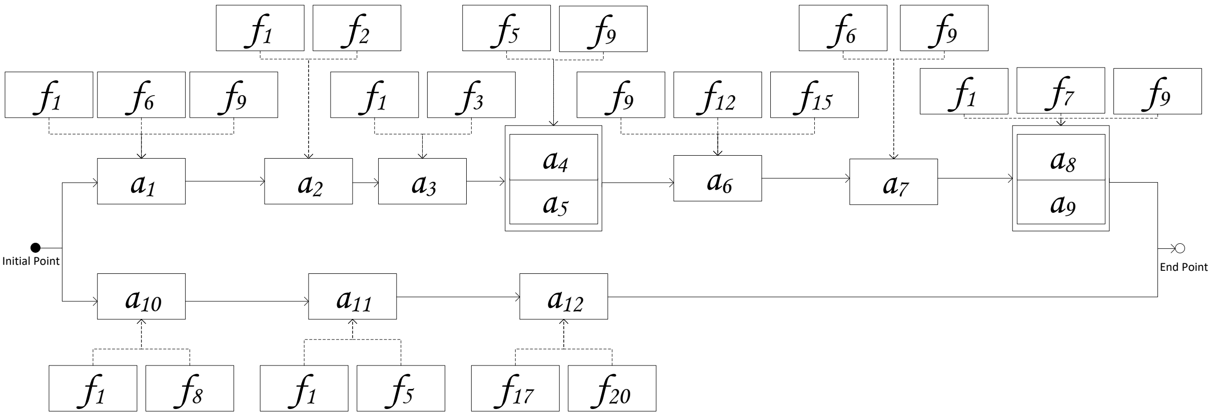


Figure 6.9: Fault Propagation Chain with Facilitation of Level Crossing Control System

The fault arrangement of functional and structural of a system in the FPC presentation is then transformed into an integrated Fault Tree of LCCS as depicted in Figure 6.10. The types of facilitation can infer the connection between facility fault event (basic event) and action fault event (intermediate event) in Fault Tree. From the observation, one can expect none of single structural fault event is directly connected to functional fault event without any logic gate as the facilitation involves none of single type of facilitation. Therefore, in the integrated Fault Tree, all of the transformed structural fault events are connected by OR-gate to the correspond intermediate event, e.g. multiple basic event of structural fault are connected together by an OR-gate to output event of functional fault. For example, basic events f_1, $f_6$and f_9 are tied together (group) with OR-gate to the intermediate a_1.

Based on the frequency of a facility involves in the facilitation, the repeated appearance of basic events of the same structural faults is reflected in the integrated Fault Tree. The basic events of structural fault can be seen once in the integrated Fault Tree if the corresponding facilities appear one time in the facilitation. For example, f_2, f_3, f_8, f_{12}, f_{15}, f_{17}, and f_{20}basic events are seen for one time in the integrated Fault Tree. As compared to structural fault evens that their corresponding facilities appear more than one time in facilitation, f_1, f_5, f_6, f_7, and f_9, after they have been transformed into basic events, they have more than one time appearance in the integrated Fault Tree. For example, the facilitation of structural fault event f_1 can be found as one of the basic events of seven functional fault events a_1, a_2, a_3, a_8, a_9, a_{10}, and a_{11}.

For simplicity, all the basic events of functions failures in the previous functional Fault Tree, based on the facilitation and integrated FPC, are transformed into intermediate events. These intermediate events are connected by the basic events of structural faults. Thus, the minimal cut sets are only consisted of the twelve structural faults, $\{f_1\}$, $\{f_2\}$, $\{f_3\}$, $\{f_5\}$, $\{f_6\}$, $\{f_7\}$, $\{f_8\}$, $\{f_9\}$, $\{f_{12}\}$, $\{f_{15}\}$, $\{f_{17}\}$, and $\{f_{20}\}$. Similar to the previous functional Fault Tree, every single structural fault event of minimal cut sets contributes to the occurrence of the top event, a_s, and this align with the allocation of system functions to system components, where the failure rate of facility will contribute to the calculation of the probability of functional failure. For example, the probability of failure of functional fault event a_1, $P(a_1)$, is contributed by the probability of failure of collectively three structural fault event f_1, f_6, and f_9, i.e. $P(a_1) = P(f_1) + P(f_6) + P(f_9)$. From the generated Fault Tree, one can analyse the relevancy of the facilities that have been modelled in UML Classes to facilitate function (action) of system. For example, “LCCS Control Unit” (F_1) is the most common facility to facilitate in the functionality of LCCS. This can be seen from the time of facilitation which in fault viewpoint, the facilitation of f_1 occurs seven time.

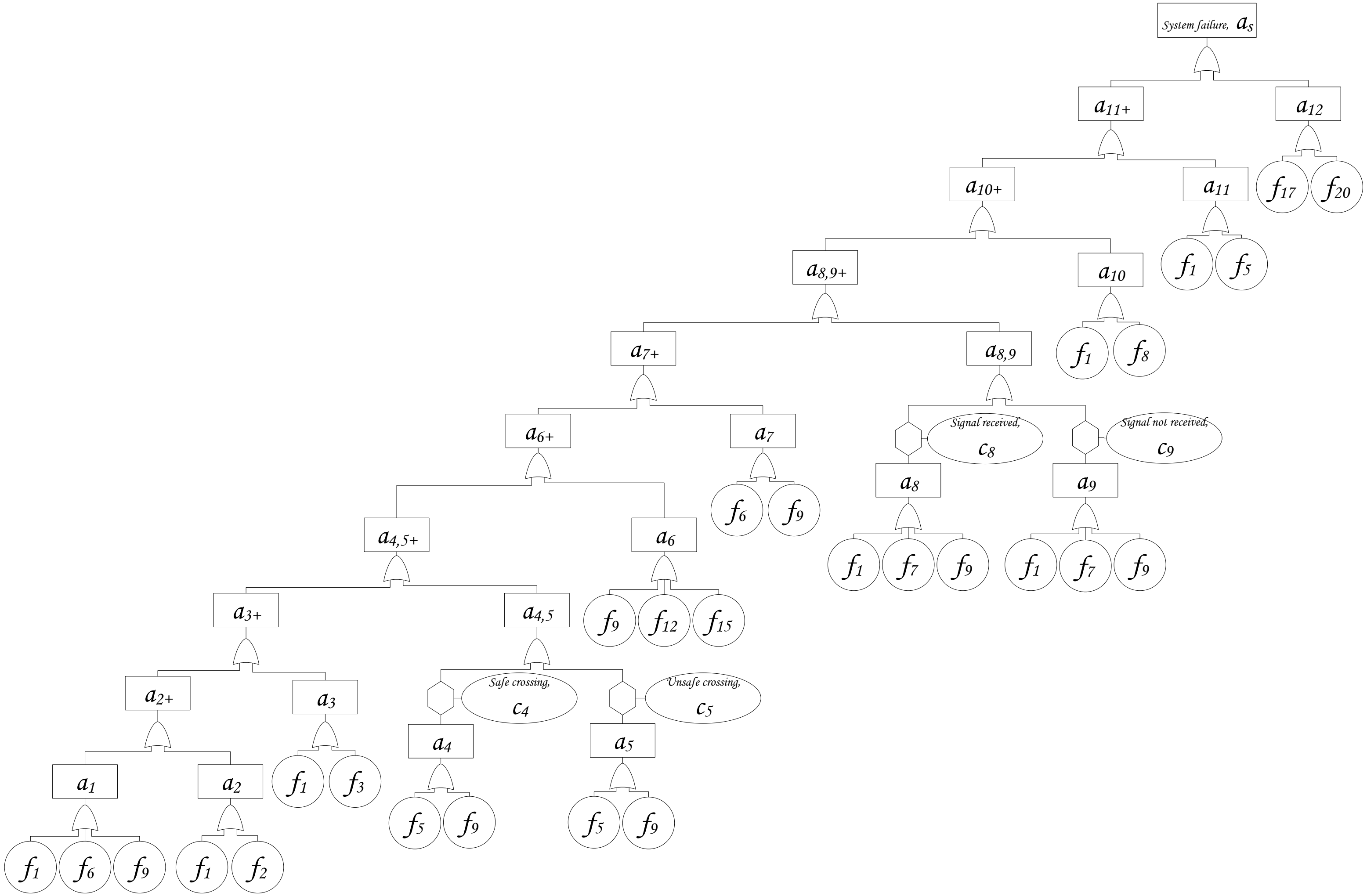


Figure 6.10: Integrated Fault Tree of Level Crossing Control System

6.4.3 Classes to Fault Tree

In this subchapter, the structural faults of the integrated Fault Tree are analysed into a lower detail based on the facilities relation in Class model of LCCS depicted in Figure 6.5. The subchapter starts with the identification of Composition relationship of Classes in the Class model to generate an extended Fault Tree which comprises component Fault Tree. This identification leads to the decomposition of basic events of the integrated Fault Tree generated in previous subchapter. As structural faults are the main concern in this subchapter, the quantity of the same facility is traced from the multiplicity of a Class which is then horizontally extended the basic events in the Fault Tree. The extended Fault Tree of LCCS is then used to analyse the system models based on design intent of LCCS.

Based on the structural fault events that have been facilitated to functional fault events in the integrated FPC and tracing back to the Classes modelled for LCCS, "LCCS Transceiver)" (F_4), "Train" (F_{11}), "Train Transceiver" (F_{12}), and "Central Office (CO)" (F_{16}), are modelled as owner Class of two part Classes with Composition relationship. By assuming that the part Classes presenting the complete Classes for the owner Class, the logical connective as in Relation (4.3.12) can be applied. For example, the material equivalence of the structural faults is presented as $f_{12} \equiv f_{13} \vee f_{14}$.

Furthermore, the number of component is also considered based on the multiplicity of Class. As referring to the Class modelled for LCCS based on the technical description, "Traffic Light" (F_2) and "Barriers" (F_3) are two Classes that represent facilities which designed with two of each facility for LCCS. The same facilities serve the same function. A pair of a traffic light and a barrier is located each of two crossroads (c.f. refer Figure 6.1). Noting the same facilities which serve the same functions are not redundant, i.e. one facility is operating at a time and another is serving as a backup.

Based on the Composition relationship and multiplicity identification, the integrated Fault Tree is extended as depicted in Figure 6.11. Two differences can be spotted between the integrated Fault Tree and extended Fault Tree. First, in the extended Fault Tree, the basic event f_{12} is transformed into an intermediate event of two basic events, f_{13} and f_{14}. This is due to the Composition relationship discussed earlier. Second, basic events f_2 and $f_3$are decomposed by two fault events of each. They are identified with f_2^1 and f_2^2 for basic events f_2 and f_3^1 and f_3^2 for basic events f_3. This is to present, for example, either traffic light 1 fails to available, f_2^1, or traffic light 2 fails to available, f_2^2, traffic light is failed to available, f_2. The extension part of the extended Fault Tree is the component Fault Tree. Therefore, the extended Fault Tree is comprised of component Fault Tree. However, based on the transformation, it is not necessary for generating an extended Fault Tree if Composition and redundancy are not applicable.

For the extended Fault Tree, all the basic events are also consisted of structural faults. The functional faults as represented as intermediate events in the tree are not affected by the Class model transformation. The minimal cut sets for the extended Fault Tree are $\{f_1\}$, $\{f_2^1\}$, $\{f_2^2\}$, $\{f_3^1\}$, $\{f_3^2\}$, $\{f_5\}$, $\{f_6\}$, $\{f_7\}$, $\{f_8\}$, $\{f_9\}$, $\{f_{13}\}$, $\{f_{14}\}$, $\{f_{15}\}$, $\{f_{17}\}$, and $\{f_{20}\}$. The basic event f_{12} is no longer in the minimal cut sets list as it has been decomposed by $\{f_{13}\}$ and $\{f_{14}\}$. The minimal cut sets in the extended Fault Tree is lengthier that minimal cut sets in integrated Fault Tree as more structural faults events are identified in the contribution to the system failure, a_s. For example, instead of a traffic light failure, the two traffic lights as in intended design are separated and its individual is considered in the contribution to the system failure. Although the occurrence of one of the basic events leads to the system failure, since all of the logic gates are OR-gate, the separation as individual when there is a redundant facility can be identified. Therefore, the occurrence of the top event will not be affected by the occurrence of fault event of redundant facility. Furthermore, the extended Fault Tree presents a lower detail of structural fault events which can be used to analyse the system architecture design. For example, if the Train is modelled with “receive ‘Passed’ Signal” together with “send ‘Passed’ Signal” (A_6) in Activity model, the generated Fault Tree will consist of basic events f_{13} and f_{14} from facilitation on FPC directly in the integrated Fault Tree instead of decomposed from f_{12} in the extended Fault Tree.

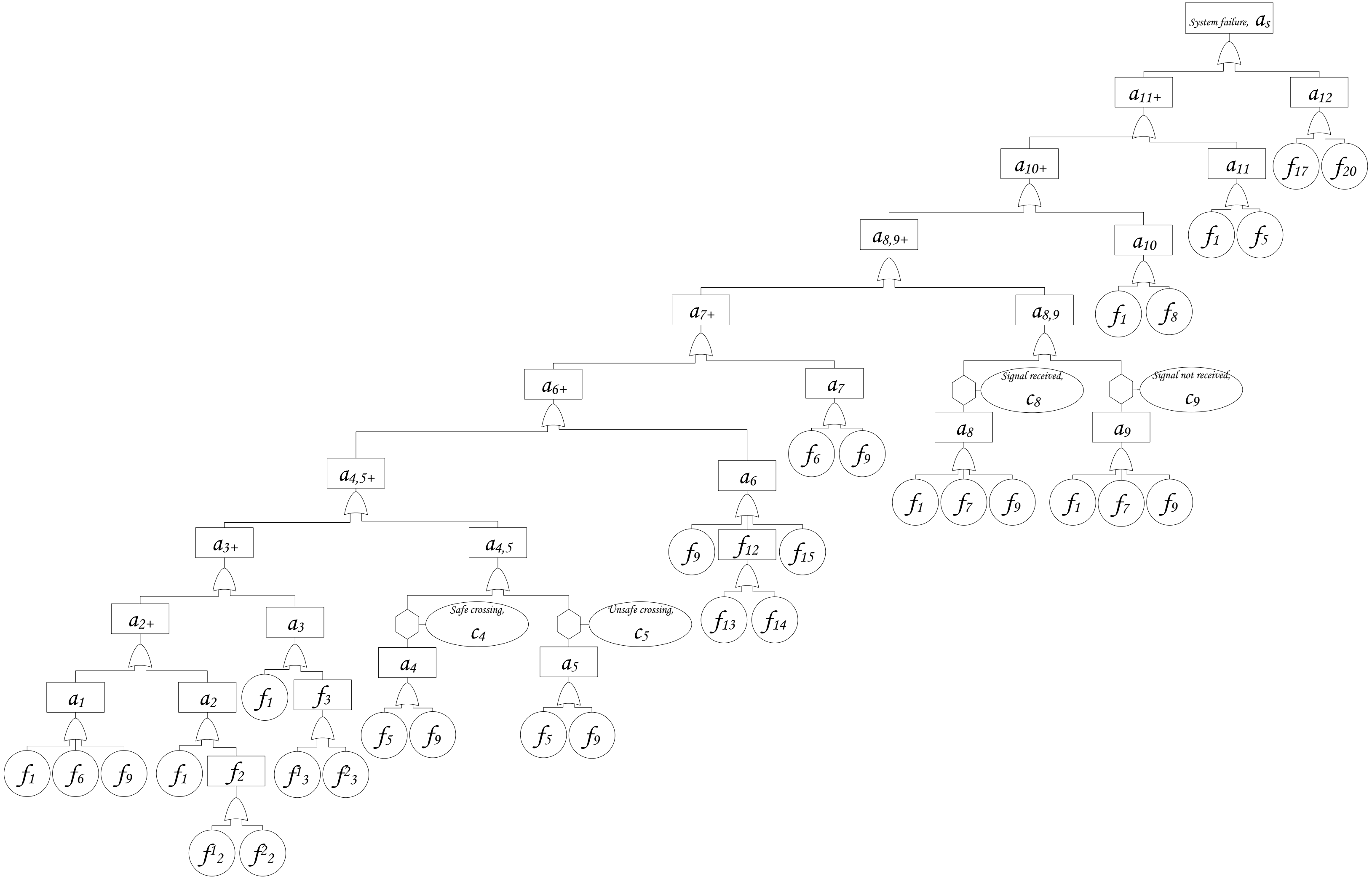


Figure 6.11: Extended Fault Tree of Level Crossing Control System

6.4.4 Functions Ownership of Class for a Complete Fault Tree Transformation

In this subchapter, ownership of LCCS facilities on specific function(s) modelled by using UML Class is identified. The identified functions are transformed into fault viewpoint for elaborating structural fault events of extended Fault Tree. The elaboration is observed as to conclude the transformation methods from system models in UML to Fault Tree.

There are 15 basic events in the minimal cut sets, $\{f_1\}$, $\{f_2^1\}$, $\{f_2^2\}$, $\{f_3^1\}$, $\{f_3^2\}$, $\{f_5\}$, $\{f_6\}$, $\{f_7\}$, $\{f_8\}$, $\{f_9\}$, $\{f_{13}\}$, $\{f_{14}\}$, $\{f_{15}\}$, $\{f_{17}\}$, and $\{f_{20}\}$, are identified from the extended Fault Tree of LCCS from previous subchapter. These basic events represent LCCS structural fault. As the LCCS structural faults in the Fault Tree are traceable in the LCCS Class model, the function(s) (Operation) of each facility can be identified from the corresponding Classes. Therefore, each facility is associated with function(s). For example, the functions of a traffic light which used to give indicator to vehicle and pedestrian in the LCCS as specified in "Traffic Light" (F_1) Class are switch and display. However, in fault viewpoint, the failure of traffic light to function as intended leads to the failure of the traffic light f_2. The basic events f_2^1 and f_2^2 can be elaborated by inserting transfer events (transfer out) of the failure functionality of the traffic lights. This applies to all the basic events of extended Fault Tree for the elaborated Fault Tree as depicted in Figure 6.12.

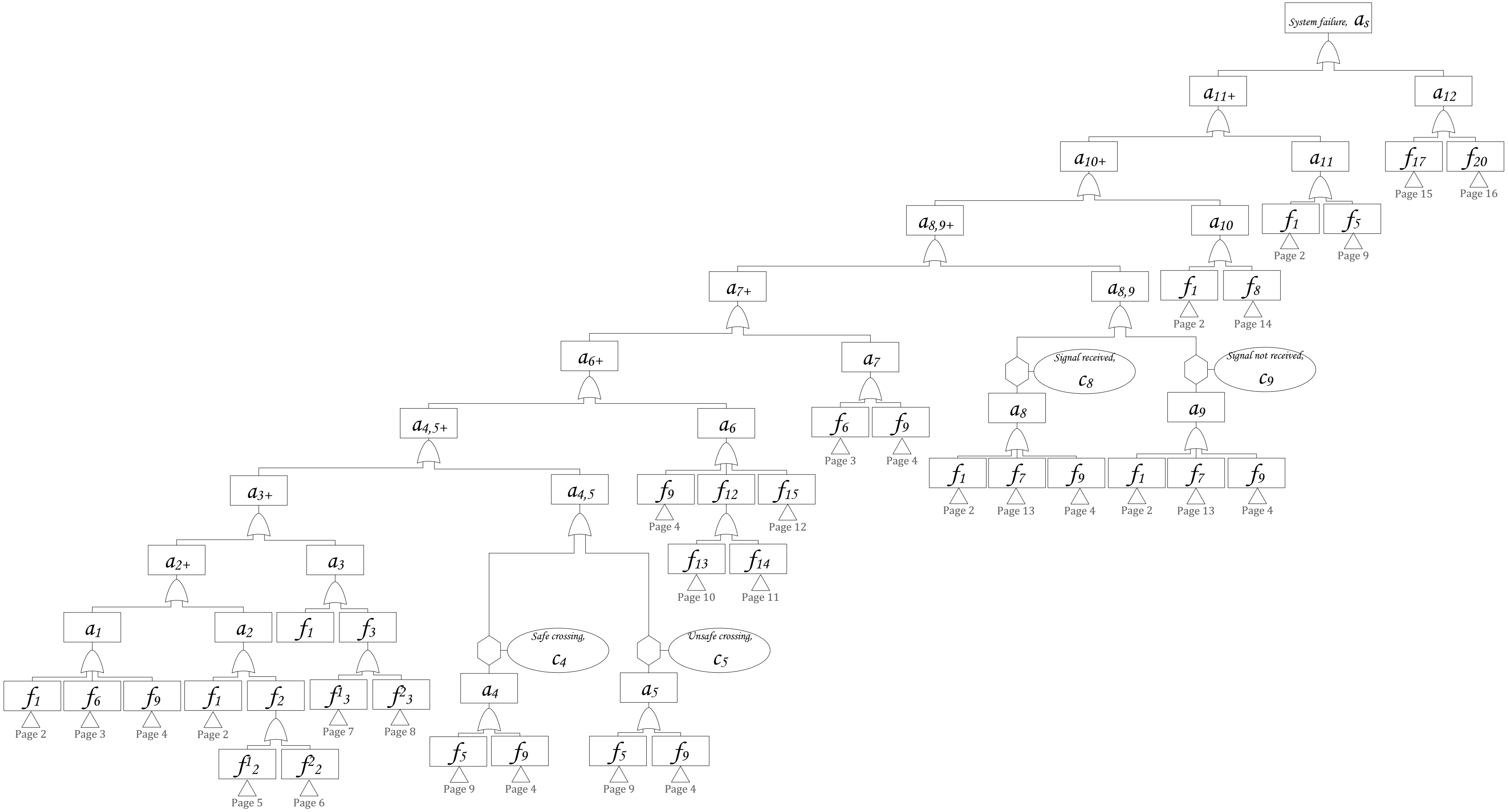


Figure 6.12: Elaborated Fault Tree of Level Crossing Control System

The basic events are replaced by transfer out events by connecting the top of transfer out event (triangle) to the bottom of output event (rectangle), i.e. a pair of each basic event. Under the transfer out events, an indicator of the location of the transfer is specified. For example, under the transfer out event of f_2^1, an indicator of "Page 5" is marked. This means the lower details of f_2^1 can be found in page 5. Assuming the elaborated Fault Tree is the on Page 1 as it is the first constructed tree. In the page 3, the same intermediate event of f_2^1 with a connected transfer in event (triangle) next to each other is presented. Under the transfer in event, an indicator of "Page 1" is specified to link back to the original Fault Tree. The transfer in event of f_2^1 is depicted as in Figure 6.13.

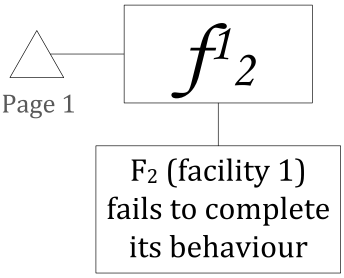


Figure 6.13: Fault Tree of f_2^1 Transfer In Event

6.5 Comparative Analysis

In this subchapter, the developed transformation methods are evaluated by conducting a qualitative comparison against established transformation methods. Technically, modelling and problem solving concerned in the establishment of the transformation methods can be related to system methodology suggested by Klir in 1980's. According to the Klir's system methodology, an abstraction of system that interpret into a model of a system is an approach of discovering the whole system (Zeigler 1985). A model that abstracts problem of a system the model can demonstrate solution to the problem. The problem solving is also an approach of concept of structured analysis. The concept supports separation of problems from a system scale before solves the problem. This can reduce the complexity of problem solving process by focusing on particular concerns within the system.

For this comparison, the established transformation methods are categorised into two sets. The first set concerns with transforming systems models developed in standardised modelling languages such as UML (Kim et al. 2012), (Hu et al. 2011), (Zhao & Petriu 2015), (Kim et al. 2010) and SysML (Mhenni et al. 2014), (Xiang & Yanoo 2010), (Yakymets et al. 2013) to Fault Trees; while the second set concerns with formalism of the transformation methods. These comparisons can be referred to the MBSA tools categories (fault modelling, FLM, and behavioural fault simulation, BFS) in Table 2.2.

The four proposed transformation methods from system models modelled in UML to Fault Tree generation covers: (i) A transformation of UML Activity model into a Fault Tree, (ii) A transformation of Classes with Composition relationship model into a Fault Tree, (iii) A transformation of integrated behavioural and structural from UML Activity and Class models to extend the Fault Tree. A comparison to other transformation to Fault Tree from information modelled in other techniques such as diagraphs (Wang et al. 2003), (Vemuri 1999), component-based modelling (Majdara & Wakabayashi 2009), (Bhagavatula et al. 2016), knowledge-based approach (G 2002), and architecture description language (Joshi et al. 2007) would require a comparison between the modelling techniques, which would be beyond the scope of this research.

The proposed methods are unique in two ways: firstly, one of the method is the first attempt in transforming an Activity model with a focus on system behaviour and fault propagation; secondly, the methods gather two single perspectives, i.e. functional and structural, and integrate both perspectives. In comparison, on the UML side, similar to the proposed methods, there are methods developed based on system model presented in a single type of diagram: Use Case in (Hu et al. 2011); and State Machine in (Kim et al. 2010). In both research, rules and algorithm are the two proposed practices for analysing system safety based on system model. However, both research apply different category of MBSA tools. Rules and algorithm in (Hu et al. 2011) applies FLM as they are used to analyse safety on the modelled system. While rules and algorithm used in (Kim et al. 2010) extend State Machine with hazard from Fault Tree to develop primary fault events which applies BFS of MBSA method. Some other methods developed consider multiple diagrams: for instance, (Zhao & Petriu 2015) has a wider focuses by utilising UML Composite Structure, Sequence, and Use Case Diagrams. The information from each UML diagrams is transformed to the different level of fault events in Fault Tree. On the SysML side, methods developed in (Mhenni et al. 2014) transform information captured in SysML Internal Block Diagrams; and the method developed in (Yakymets et al. 2013) utilises both Block and Internal Block Diagrams. Mhenni et. al. utilise directed graph to perform graph traversal as intermediate processes from the structural diagram to Fault Tree generation. Different patterns are identified through the processes as to provide safety analysis artefacts. The transformation performed by Mhenni et. al. is classified as FLM. Dissimilar to BFS category proposed by Yakymets et. al. onto their transformation activites. The structural diagrams are further annotated with failure behaviour to represent deviation from internal failures of the diagrams. Although many of the above methods have considered system behaviours, none of them emphasises the concept of fault propagation that is manifested from control flows as modelled in UML or SysML Activities.

In addition, to facilitate the transformation methods developed in most of these works, additional stereotype are introduced to the standard UML and SysML models; whereas the proposed methods do not. As such, the advantage of the proposed method, i.e. transformation of UML Activity model into a Fault Tree, that reflect exactly the reliability of system behaviour architecture is claimed; while the limitation of the method is that it only concerns one type of fault, which is functional failure. The method is extended with facilitation to integrate structural failure in the developed Fault Tree. The developed formal transformation methods in this thesis can be classified as FLM of MBSA tools. The methods transform system models without any failure mode models extension to generate Fault Tree. This presents the safety analyses are done based on what have been modelled for the system.

In comparison, on the formalism side, similar to the developed methods, there is a method developed based on conjunctions of predicates for constructing a formal Fault Tree (Xiang et al. 2005). The conjunctions of predicates are applied to states transition of a SoS as transition rule that identify a normal state (not failure state) such as barriers are open could also lead to system failure (top event of the tree) which in this case is collision between crossing train and vehicles (c.f. Railway System case study). However, the system analysis is grounded to security of the system in which the system can be disturbed in any course such as train is hijacked. Furthermore, the root (top event) of the tree for analysing the safety of the system is an unintended situation (collision between crossing train and vehicles) instead of failure of LCCS in serving its function.

In addition, the concept of negation is being used in many Fault Tree research from computer science background for identifying fault or failure from system design. For example, as Fault Tree is a deductive safety analysis technique, Xiang et. al. identified the negation of safety regulation and requirement of a system as the top level of the tree (Xiang et al. 2004). Although, the logical reasoning for the top event has been formally verified with CafeOBJ, the engineering practices and system viewpoint are leaving behind.

6.6 Summary of Verification of the Developed Formal Transformation Methods

In this chapter, a trial run of the developed transformation methods is conducted as a part of verifying the methods that developed in this research. The developed transformation methods including metamodels have been applied to an authorised Railway System case study. The technical description of the system is gathered from several published papers to acquire a comprehensive understanding of the system for the application of the methods. The Activity and Class models of the Railway System are developed in UML as the sources for the Fault Trees transformation. For the purpose of this chapter, LCCS is selected as the main CS of the Railway System together with three other CSs: (i) Train, (ii) Central Office, and (iii) Vehicle. The system architecture is designed from the viewpoint of the LCCS. The methods developed in this research are observed to be able to apply to complex system.

The overview of the transformations from UML system models to Fault Tree is depicted as in **Figure 6.14**. In the application of the transformation methods, four types of Fault Tree are generated. First, functional Fault Tree is generated from Activity model of the LCCS with the supporting methods: semantic mapping rules and FPC. In the functional Fault Tree, a top event is defined as system failure and the intermediate fault events are composed of functional fault events (top event and functional events at level 1). Taking the functional Fault Tree as the backbone of further Fault Tree generation and Class model as presentation of facilities of LCCS, the concept of facilitation is applied. As a result, an integration of functional and structural failures (structural fault events at level 1) is reflected in the generated integrated Fault Tree. The extended Fault Tree is then generated by transforming Classes in Composition relationship which has material equivalence structure. The extended Fault Tree is comprised of component Fault Tree at the structural faults decomposition (structural fault event at level 2). Finally, the elaborated Fault Tree is generated by applying the concept of ownership of facility to function. In this thesis, the basic event of structural event is transformed from facilities that have been designed in UML Class. The Class that presents facility could be hardware or software. When the facility is not atomic level, its failure can be caused by fault or error from other element. Therefore, the transformation methods allow decomposition of fault events in a hierarchy structure. When necessary, the fault events can be decomposed into the next level, i.e. decomposition of basic event of structural.

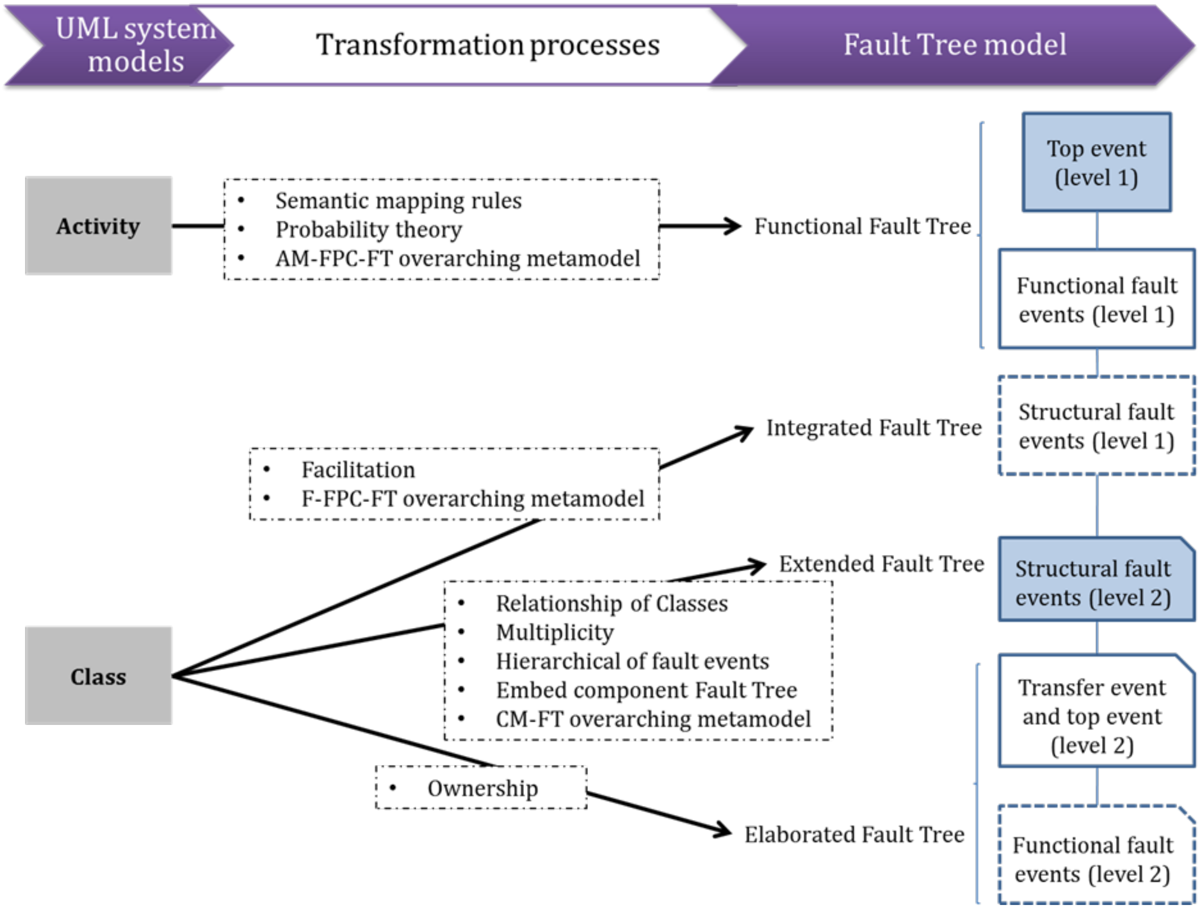

UML system models
Transformation processes
Fault Tree model
Activity
Semantic mapping rules
Probability theory
AM-FPC-FT overarching metamodel
Functional Fault Tree
Top event (level 1)
Functional fault events (level 1)
Integrated Fault Tree
Structural fault events (level 1)
Facilitation
F-FPC-FT overarching metamodel
Extended Fault Tree
Structural fault events (level 2)
Class
Relationship of Classes
Multiplicity
Hierarchical of fault events
Embed component Fault Tree
CM-FT overarching metamodel
Transfer event and top event (level 2)
Ownership
Elaborated Fault Tree
Functional fault events (level 2)

Figure 6.14: Overview of Transformation from UML System Models to Fault Tree Generation

Technically, the transformation methods are established based on two key features model-based approach and separation of concerns. These key features can be reflected to system methodology suggested by Klir and the concept of structured analysis. The establishment of the transformation methods are compared according the transformation from system models to Fault Tree approach and formality of the concerned in the transformation methods. The transformation is the first attempt of transforming Activity model to Fault Tree. The formality concerned in the transformation methods enables preservation of the relational structure of Activity model and Classes in compositional relationship in the Fault Trees. In addition, the transformation methods lead to the decomposition of fault events from behavioural failure to structural failure and elaboration of the basic events.

To develop a Fault Tree by using traditional way after waiting the architecture design completed is time consuming. The application of the developed methods can leverage the safety analysis by generating Fault Tree from the UML system models. The application of the developed methods enables analysing safety of system design which is a step of closing the gap between SE and SSE.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Four transformations methods have been developed based on propositional logic and probability theory to allow a system modelled in UML to be transformed semantically into its equivalent Fault Tree. The methods developed align with current industrial practices in early stage system assurance (Zeller et al. 2016) and advance existing approaches in terms of accommodating system model availability (Majdara & Wakabayashi 2009), (Bhagavatula et al. 2016) while incorporating mathematical rigour (Mhenni et al. 2014), (Yakymets et al. 2013). A few concepts have been introduced to assist the transformation methods; FPC as an intermediate step and facilitation as an extension step. Three overarching metamodels were then developed; the AM-FPC-FT overarching metamodel bridging the metamodels of Activity model, FPC, and Fault Tree, the CM-FT overarching metamodel bridging the metamodels of Classes in Composition relationship and Fault Tree; and the F-FPC-FT overarching metamodel bridging the metamodels of FPC with facilitation and Fault Tree. The formal basis of the transformation methods, together with the overarching metamodels, assist automated Fault Tree generation. The main contribution of the formal transformation methods to the Fault Tree generation is the relational structure mapping. Furthermore, the transformation methods developed in this thesis support formal approach for generating Fault Trees from system architecture models. As discussed by Xiang et. al., the relational structure mapping which can be seen as a practical way of helping engineers in the line to understand the developing Fault Tree by using formal methods (Xiang et al. 2004). The relational structure mapping can potentially add new knowledge in the discovery of faults and fault structure (i.e. fault logic such as teo faults associated with AND-gate) which are not manifested from a conventional entity-to-entity mapping.

To demonstrate and evaluate the applicability of these developed transformation methods, they were applied to the RMS case study previously studied in (Ingram et al. 2014) and the LCCS case study previously studied in (Reif et al. 2000), (Schellhorn et al.

2002), (Xiang et al. 2004), and (Xiang et al. 2005). The application of the formal transformation methods to these case studies demonstrates their compatibility with SoS problems. The transformation methods developed in this thesis utilise system models in UML to generate Fault Trees. The elements in the generated Fault Trees are traced back to the elements of the corresponding system models. Additionally, the methods enable the preservation in the transformed Fault Tree of the relational structure between system behaviour and system structure. Qualitative analyses of the Fault Trees were then carried out to examine the quality of the system architecture designs. The designer will be informed by the safety engineer of safety and reliability features based on the outcomes of the analyses. Furthermore, the developed overarching metamodels bridging the metamodels of UML, FPC, and Fault Tree can be used as the basis of a software automation tool.

Despite the successful application of the methods to the cases studies, there are limitations of the approaches presented in this thesis. For instance, loops are particularly useful in modelling control systems, but have been deliberately avoided in control flows as they do not naturally fit into the structure of a standard Fault Tree. Majdara and Wakabayashi in (Majdara & Wakabayashi 2009) also suggested that events that are encountered repeatedly can lead to an infinite loop structure in a Fault Tree. In addition, the configuration of components in a system supports the identification of component redundancy structure and error propagation that leads to system failure (Pai & Dugan 2002). Work needs to be undertaken in future, therefore, to resolve these issues by using of UML Structured Activity Nodes such as LoopNode, as well as, Connector in the UML Composite Structure.

Furthermore, because the current state of the approach is concerned only with control flows of behavioural model and the Composition relationship, the transformation methods are not applicable to object flows and interaction of components. In this context, four pathways are envisioned to extend the current work and to make the methods more widely applicable. These are extensions of the transformation of control flows, facilitation, and ownership to:

1) object flows;

2) state machines;

3) interactions;

4) non-functional aspect;

The following discusses each pathway in more detail. With these extensions, it should then be possible to evaluate the reliability of the transformed Fault Tree in great depth by comparing it to similar work.

Object flow, which was not included in the scope of this thesis, is the other important aspect of UML Activities. Errors presented in an Object, e.g. incorrect data that are passed between system components, may lead to undesired system functions such as malfunctions, in addition to the fault events that this work is concerned with. Developing a formal representation of Objects and their associated faults can be challenging. It is anticipated that the use of Predicate Calculus to capture the states of an object can provide insights into the inclusion of object flows.

In practice, the modelling of system behaviour is also concerned with the modelling of system states. In UML, these system states are modelled using State Machines. Since the executions of system functions is closely related to the changing of system states, the transformation between the two types of behavioural models could provide insights in elaborating current transformations to further consume State Machines as system behavioural models. It would also be useful to utilise and elaborate other existing works, such as that of (Kim et al. 2010). In addition, an overarching metamodel of both behavioural diagrams can be developed by abstracting related elements owned between Activity and State Machine.

The behaviour of a system can be modelled according to the execution of system functions. In UML, other than Use Case, Activity and State Machine, the system functions are modelled at varying levels of detail by using Interactions. Sequence, one of these Interactions, is used to model functions precisely up to the level of inter-processes according to sequence of occurrence. The Operation of a Class that addresses the functions of the modelled system can be modelled using Sequence. Since the Fault Tree generated by using the developed methods concluded with failure of system components to complete its behaviour, the methods can be explored to include Fault Tree transformation from functions that are designed by using non-Activity model.

Lastly, the non-functional aspect of a system can also be considered in the generation of Fault Trees based on the transformation from UML system models. For example, a failure in the aircraft Traffic Alert and Collision Avoidance System (TCAS) can lead to incorrect colours and shapes being displayed on the Resolution Advisory display which can in turn lead to inappropriate aircraft manoeuvres, potentially putting the plane at risk (Department of Transportation Federal Aviation 2011). A failure of a system analysed in a Fault Tree can be triggered by an error or malfunction. State machine and Object can therefore offer positive insights by analysing non-functional aspects of a system.

The presented approach in this thesis to transform Activity Diagram and Class Diagram into Fault Trees is rooted in the use of propositional calculus to capture failure semantics and failure logics (relational structure). Extension of this approach to other models such as State Machines and Interactions would therefore imply the use of propositions to capture different failure semantics and failure logics. If the thesis scope failure semantics to be only complete failure (e.g. “message is unavailable”) similar to the approach of this thesis, capturing these failure semantics involved in additional UML diagrams would be straightforward. However, capturing the different failure logics embodied in the additional UML diagrams would be much more challenging. For instance, in State Machines, transitions between states can be triggered by various mechanisms such as pre-planned events, operator actions, and environment changes. These mechanisms involving both discrete and continuous dynamical behaviour, would likely lead to the need of different failure logics to be captured using advanced and complex inference in propositional calculus.

In addition to the above, as already mentioned in Subchapter 4.3.2, extension of the current work also means going beyond complete failure to include other failure semantics such as malfunction and erroneous outputs. Therefore, this brings additional challenge in using solely propositional calculus to capture different semantics with various types of failure logics between them in the additional UML diagrams.

The second challenge is concerned with creating an integrated picture of the transformed Fault Trees. Extension of the method does not stop at transform an additional diagram into a Fault Tree, but it is also necessary to integrate the resultant Fault Tree with the hierarchical Fault Tree that is transformed from an integrated Activity and Class diagrams as presented in Chapter 5, with using the concept of facilitation. Therefore, it is important that an integration mechanism, similar to facilitation, needs to be developed in a formal language, such as propositional calculus, to properly integrate the diagrams, and the transformations of these diagrams such that the resultant Fault Tree is also an integrated one.

Last but not least, defining probability calculation for looping mechanism is certainly required. This is especially true for Interactions and State Machines where various types of looping (e.g. iterative, alternative, and breaking) are captured. This thesis has currently deliberately avoided transforming loop structure due to Fault Trees being incapable of handling it using the classical probability theory and Boolean logic. Therefore, extension of the current method would mean to find an alternative to allow transformation of loop mechanism into a proper combination of failure semantics and logics that adhere to the metamodel needs to be redesigned to allow modelling and analysis of loops often used in system behaviour.

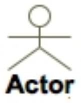
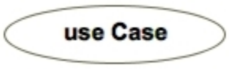
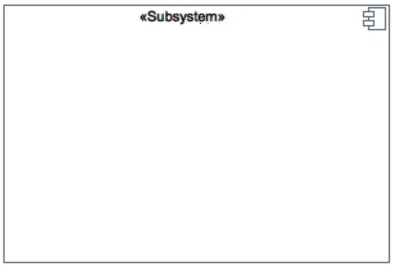
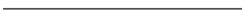
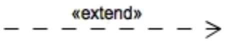
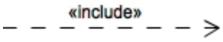
APPENDIX A

Use Case Diagram as a Behaviour Model

The behavioural view of a system that emphasises on the highest level of system functionality can be realised by using Use Cases. In the Use Cases presentation, three levels of requirements can be observed. The Use Case requirement levels include operational requirement, functional requirement, and elaborated functional requirement (listed from highest to lower details of Use Case). The highest level of Use Case is expanded for a lower level Use Case to capture lower level of system details. The level of the system details can be marked and compared amongst the Use Cases. The most detailed Use Case can be very helpful to system engineers as Use Case provides technical information to develop a system. The system behaviour can be effectively expressed by use case description in which showing the relationships (Some 2007). The comprehensive information of the system behaviour from the elaborated functional requirement Use Case is proposed for Use Case description as presented in Table A.1. The coherent system function for Use Case description can be used as a system test case to check the system function flow. The performance and interaction between a system and its boundary are also presented as part of the system functionality. Requirements and context of a system is primarily modelled by using Use Case model that emphasise missions and stakeholder goals. Use Case presents an abstraction of a system function including an overview of SoS. The abstraction presented is sufficient for the least understanding of the system function to the stakeholders. The Use Case can be presented in several levels of details which derived from the highest level of Use Case. The Use Cases presentation emphasise what systems are supposed to do.

A Use Case diagram is made up of three key concepts which are subject, Use Cases, and Actors (Object Management Group 2017d). The subject is defined as the system boundary that presented in rectangle shape. It is used to describe the context of the system. Specifically, a behaviour of the system of interest is specified by a Use Case presented in oval shape. Usually, the content of a Use Case is written in a verb noun phrase with small letters except first letter of the noun e.g. use Case and control Vehicle Flow (refer to Figure 3.9). However, this has not been specified by OMG. The functions of a system is specified within Use Cases are defined at the level which Actor, presented in 'stick man' icon, of the system can achieve. A role or a user including the environment that interacts with the system of interest is modelled using an Actor. The interaction between an Actor to the specific behaviour of the system interest is modelled by Association. An Actor can interacts with more than one use cases. In the case where the behaviour of a system (at the highest level of information) can be further defined, <<extend>> and <<include>> relationships are applied between Use Cases. Both of them are represented by directed dashed arrows. They are differentiated by pointing opposite direction to or from first (base) level Use Case. A dashed-directed arrow pointing from extending Use Case into extended (base) Use Case labelled with <<extend>> as a relationship. The <<extend>> relationship is used to define optional behaviour of the extending Use Case to the extended Use Case. For example, 'Help', denoted as extending Use Case, is an optional behaviour (function) of 'Bank Automated Teller Machine (ATM) Transaction', denoted as extended (base) Use Case. The optional function is not necessary for a user to make a transaction from the ATM. Another further defined relationship <<include>>, can be defined as an extraction behaviour of the including (base) Use Case behaviour. A dashed-directed arrow, labelled with <<include>>, is pointed away from including Use Case into included Use Case. The <<include>> relationship shows the included Use Case as an independent behaviour and it is a necessary to the including (base) Use Case. For example, an including Use Case of 'Bank ATM Transaction' has an included Use Case of 'Customer Authentication'. This means, the user needs to insert a bank card and pin number as for proving identification prior to do a transaction. The basic symbols of a Use Case Diagram are listed as in Table A.1.

Table A.1: Use Case Model Symbols

Symbol	Description
	**Actor** – Usually, an Actor is represented by a “stick man” icon labelled with the name of the Actor.
	**Use Case –** An ellipse shape with verb-noun written in the shape.
	**Subject –** A rectangle shape that holds a set of use case behaviours labelled.
	**Association –** A line that connects Actor and Use Case.
	**Extend –** Shows the included Use Case as an extraction of the including (base) Use Case
	**Include** – Shows optional behaviour of the extended (base) Use Case by the extending Use case

Use Case Metamodel

Within the scope of this thesis, a metamodel for Use Cases modelling is derived by simplifying OMG UML Specification. The Use Cases Metamodel is depicted in Figure A.1. Actor and UseCase are classified as BehaviouredClassifier that declares a set of offered behaviours (Object Management Group 2017d) Actor and UseCase are associated between each other and can be involved in multiple associations. The Use Case is owned by Classifier which represents the *subject* holding the performed behaviours. For the relationships between Use Cases, both Extend and Include are classified as DirectedRelationship. Metaclasses such as ExtensionPoint and Constraint are discarded to limit the scope of the research.

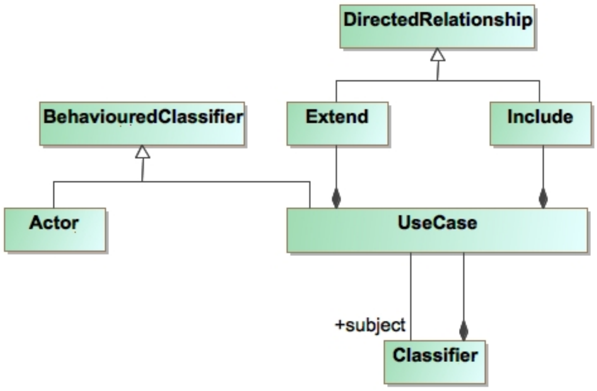


Figure A.1: Use Case Metamodel

APPENDIX B

OR-Gate and Minimal Cut Set

In a Fault Tree which merely has OR-Gate connecting all of the events, all of the basic events including any event at the bottom of the Fault Tree are the minimal cut sets in the tree by themselves. The events at the bottom of the tree can be comprised of the event that cannot be developed further for some reason such as undeveloped event and external event. As depicted in Figure B.1, in the particular structure, the occurrence of top event of the tree is contributed by at least by one the events. However, when minimal cut set applied to the Fault Tree, some of the tree especially the middle part of the tree structure is missing.

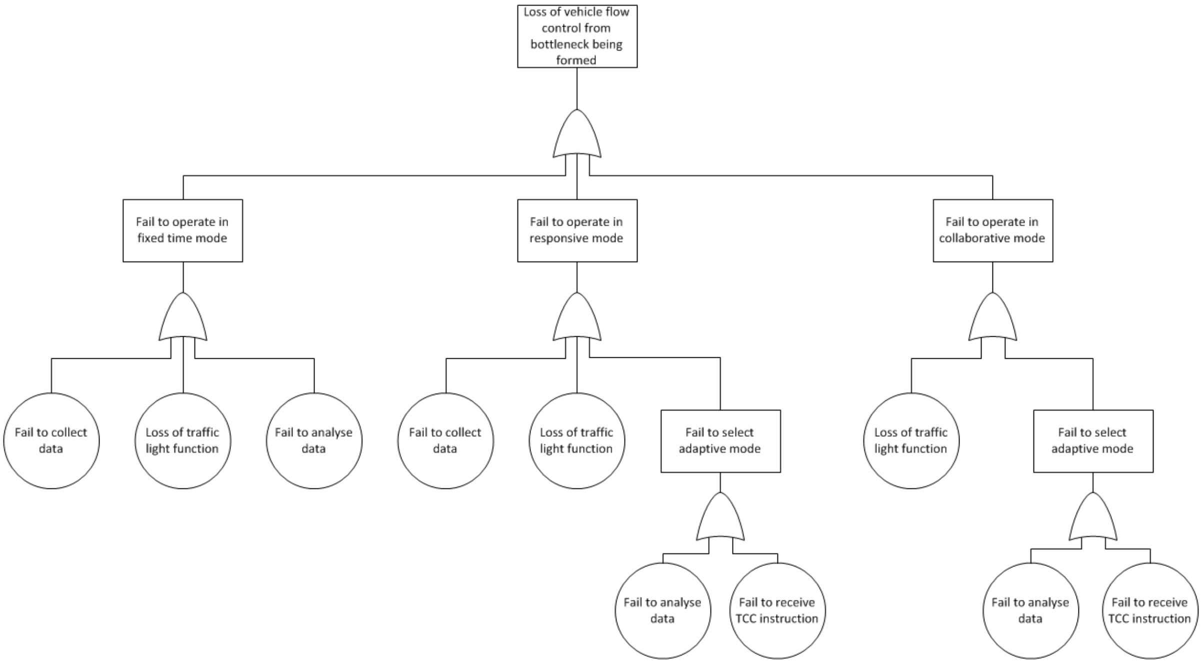

Loss of vehicle flow control from bottleneck being formed
Fail to operate in fixed time mode
Fail to operate in responsive mode
Fail to operate in collaborative mode
Fail to collect data
Loss of traffic light function
Fail to analyse data
Fail to collect data
Loss of traffic light function
Fail to select adaptive mode
Loss of traffic light function
Fail to select adaptive mode
Fail to analyse data
Fail to receive TCC instruction
Fail to analyse data
Fail to receive TCC instruction

Figure B.1: Fault Tree consisted of only OR-Gates

REFERENCES

Adler, R. et al., 2011. Integration of Component Fault Trees into the UML. In J. Dingel & A. Solberg, eds. *Models in Software Engineering. MODELS 2010. Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, pp. 312–327.

ALD, Fault Tree Analyser. Available at: http://www.fault-tree-analysis-software.com/contact-ald-engineering [Accessed January 6, 2019].

Amissah, M. et al., 2018. Towards a framework for executable systems modelling: An executable Systems Modelling Language (SysML). In *Proceedings of the Model-driven Approaches for Simulation Engineering Symposium, Mod4Sim*.

Andrews, Z., Bryans, J., Payne, R., Dider, A., et al., 2014. *COMPASS Advanced Modelling and Analysis*,

Andrews, Z., Bryans, J., Payne, R. & Kristensen, K., 2014. Fault modelling on system of systems contracts. In *Proceedings of the Workshop on Engineering Dependable Systems of Systems, EDSoS*.

Andrews, Z., Ingram, C., et al., 2014. Traceable engineering of fault - tolerant SoSs. *INCOSE International Symposium*, 24(1), pp.258–273.

Ashton, C. et al., 2015. The search for MH370. *Journal of Navigation*, 68(1), pp.1–22.

Auvation, OpenFTA. Available at: http://www.openfta.com/ [Accessed January 6, 2019].

Avižienis, A. et al., 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), pp.11–33.

Baheti, R. & Gill, H., 2011. Cyber-physical systems. *The Impact of Control Technology*, 12(1), pp.161–166.

Bahr, N.J., 2014. *System safety engineering and risk assessment: A practical approach* 2nd ed.,

Baker, P., Loh, S. & Weil, F., 2005. Model-driven engineering in a large industrial context—

Motorola case study. *Model Driven Engineering Languages and Systems*, pp.476–491.

Banhesse, E.L., Salviano, C.F. & Jino, M., 2012. Towards a metamodel for integrating multiple models for process improvement. In *38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA*. pp. 315–318.

Berramla, K., Deba, E.A. & Benhamamouch, D., 2016. Model transformation generation a survey of the state-of-the-art. In *International Conference on Information Technology for Organizations Development, IT4OD*.

Berry, D.M., 2002. Formal methods: The very idea, some thoughts about why they work when they work. In *Science of Computer Programming*. pp. 11–27.

Bhagavatula, A. et al., 2016. A new methodology for automatic fault tree construction based on Component and Mark Libraries. *Safety and Reliability Society*, 36, pp.62–76.

Blanchard, B.S. & Fabrycky, W.J., 2013. *Systems Engineering and Analysis* 5th ed., Pearson Education Limited.

Boardman, J. & Sauser, B., 2006. System of systems - the meaning of of. In *IEEE/SMC International Conference on System of Systems Engineering*. pp. 118–123.

Boehm, B.W., 1988. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 21(5), pp.61–72.

Bondavalli, A. et al., 2001. Dependability analysis in the early phases of UML-based system design. *International Journal of Computer Systems Science & Engineering*, 16(5), pp.265–275.

Bozzano, M. et al., 2015. Efficient anytime techniques for model-based safety analysis. In *International Conference on Computer Aided Verification*. pp. 603–621.

Bozzano, M. et al., 2003. ESACS : An integrated methodology for design and safety analysis of. In *Proceedings ESREL*. Balkema.

Bozzano, M. & Villafiorita, A., 2003. Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform. *Computer Safety, Reliability, and Security*, 2788, pp.49–62.

Brosgol, B.M., 2011. Do-178c: The next avionics safety standard. In *SIGAda*. pp. 5–6.

Bryans, J. et al., 2014. SysML contracts for systems of systems. In *9th International Conference on System of Systems Engineering: The Socio-Technical Perspective*. pp. 73–78.

Burgueño, L. et al., 2015. Static fault localization in model transformations. *IEEE Transactions on Software Engineering*, 41(5), pp.490–506.

Cepin, M. & Mavko, B., 1999. Fault tree developed by an object-based method improves requirements specification for safety-related systems. *Reliability Engineering & System Safety*, 63(2), pp.111–125.

Čepin, M. & Mavko, B., 2002. A dynamic fault tree. *Reliability Engineering and System Safety*, 75(1), pp.83–91.

Chanda, J. et al., 2009. Traceability of requirements and consistency verification of UML UseCase, Activity and Class Diagram: A formal approach. *International Conference on Methods and Models in Computer Science*, pp.1–4.

Clarke, E.M. & Wing, J.M., 2002. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4), pp.626–643.

Craciun, F., Motogna, S. & Lazar, I., 2013. Towards better testing of fUML models. In *6th International Conference on Software Testing, Verification and Validation*. pp. 485–486.

Czarnecki, K. & Helsen, S., 2006. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), pp.621–645.

Dai, W. et al., 2018. A cloud-based decision support system for self-healing in distributed automation systems using fault tree analysis. *IEEE Transactions on Industrial Informatics*, 14(3), pp.989–1000.

David Long, 2016. Time to drop MBSE? Available at:

http://community.vitechcorp.com/home/post/Time-to-Drop-MBSE.aspx#continue [Accessed November 1, 2018].

Dehlinger, J. & Lutz, R.R., 2006. PLFaultCAT: A product-line software fault tree analysis tool. *Automated Software Engineering*, 13(1), pp.169–193.

Department of Defense United States of America, 2012. *MIL-STD-882E Department of Defense Standard Practice System Safety*,

Department of Transportation Federal Aviation, 2011. *Introduction to TCAS II Version 7.1*,

Dirksen, J., Ten Veldhuis, J.A.E. & Schilperoort, R.P.S., 2009. Fault tree analysis for data-loss in long-term monitoring networks. *Water Science and Technology*, 60(4), pp.909–915.

Diskin, Z., 2003. *Mathematics of UML: Making the odysseys of UML less dramatic*, Springer, Dordrecht.

Dixon, J., 2017. Fault tree analysis for system safety. *John Wiley & Sons*.

Espinoza, H. et al., 2009. Challenges in combining SysML and MARTE for model-based design of embedded systems. In R. F. Paige, A. Hartman, & A. Rensink, eds. *Model Driven Architecture - Foundations and Application, ECMDA-FA*. Sringer, Berlin, Heidelberg, pp. 98–113.

Estefan, J.A., 2007. Survey of model-based systems engineering (MBSE) Methodologies. *INCOSE MBSR Focus group*, 25(8), pp.1–12.

Eti, M.C., Ogaji, S.O.T. & Probert, S.D., 2007. Integrating reliability, availability, maintainability and supportability with risk analysis for improved operation of the Afam thermal power-station. *Applied Energy*, 84(2), pp.202–221.

Felderer, M. & Herrmann, A., 2018. Comprehensibility of system models during test design: A controlled experiment comparing UML activity diagrams and state machines. *Software Quality Journal*, pp.1–23.

Fenelon, P. et al., 1994. Towards integrated safety analysis and design. *ACM SIGAPP*

Applied Computing Review, 2(1), pp.21–32.

Fitzgerald, J., Larsen, P.G. & Woodcock, J., 2013. Foundations for model-based engineering of systems of systems. In *Complex Systems Design and Management - 4th International Conference on Complex Systems Design and Management, CSD and M*. pp. 1–19.

Forsberg, K. & Mooz, H., 1991. The relationship of systems engineering to the project cycle. In *National Council On Systems Engineering (NCOSE) and American Society for Engineering Management (ASEM)*.

Fraser, M.D., Kumar, K. & Vaishnavi, V.K., 1994. Strategies for incorporating formal specifications in software development. *Communication of the ACM*, 37(10), pp.74–86.

Friedenthal, S., Moore, A. & Steiner, R., 2014. *A practical guide to SysML: The systems modeling language*, Morgan Kaufmann.

G, L.-S., 2002. Comparing selected knowledge-based fault tree construction tools. In *IASTED International Conference*.

Gogolla, M., Favre, J.-M. & Buttner, F., 2005. On squeezing M0, M1, M2, and M3 into a single Object Diagram. In *MoDELS Workshop on Tool Support for OCL and Related Formalisms*. pp. 1–14.

Gorry, B., 2015. Explanation of international standards for aircraft development.

Gough, K.M. & Phojanamongkolkij, N., 2018. Employing model-based systems engineering (MBSE) on NASA aeronautics research project: A case study. In *Aviation Technology , Integration, and Operation Conference*.

Guduric, P., Puder, A. & Todtenhoefer, R., 2009. A comparison between relational and operational QVT mappings. In *6th International Conference on Information Technology: New Generations*. pp. 266–271.

Gullo, L.J. & Dixon, J., 2017. System safety program planning and management. *John Wiley*

& Sons.

Hadian, S. & Madani, K., 2015. A system of systems approach to energy sustainability assessment: Are all renewables really green? *Ecological Indicators*, 52, pp.194–206.

Holden, T. & Dickerson, C., 2013. A ROSETTA framework for live/ snthetic aviation tradeoffs: Preliminary report. In *8th International Conference on System of Systems Engineering, SoSE: Cloud Computing and Emerging Information Technology Applications*. pp. 218–223.

Holt, J., 2001. *UML for systems engineering: Watching the wheels* P. Thomas & R. Macredie, eds., The Institution of Electrical Engineers.

Hu, W., Deng, Z. & Hong, Y., 2011. A method of FTA base on UML Use Case Diagram. In *9th International Conference on Reliability, Maintainability and Safety*. pp. 757–759.

Hutchinson, J. et al., 2011. Empirical assessment of MDE in industry. In *33rd international conference on Software engineering - ICSE*. p. 471.

Ibrahim, M. & Ahmad, R., 2010. Class diagram extraction from textual requirements using natural language processing (NLP) techniques. In *2nd International Conference on Computer Research and Development, ICCRD*. pp. 200–204.

Ingram, C., 2014. COMPASS Roadmap for research in model-based SoS engineering. , (1.0), p.121.

Ingram, C. et al., 2014. SysML fault modelling in a traffic management system of systems. In *9th International Conference on System of Systems Engineering, SoSE: The Socio-Technical Perspective*. pp. 124–129.

International Council on Systems Engineering, 2007. *INCOSE Systems Engineering Vision 2020*, San Diego.

International Council on Systems Engineering, 2015. *Systems engineering handbook: A guide for system life cycle processes and activities* 4th ed. D. D. Walden et al., eds., John Wiley& Sons, Inc.

International Electrotechnical Commission, 2006. IEC 61025 Fault Tree Analysis.

International Electrotechnical Commission, 2010. *IEC 61508-1 Functional safety of electrical/electronic/programmable electronic safety-related systems*, Available at: https://webstore.iec.ch/preview/info_iec61508-1%7Bed2.0%7Db.pdf.

International Society of Automotive Engineers, 1996. *ARP 4761 Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment* 12th ed., Society of Automotive Engineers.

ISO/IEC, 2015. *15288:2015 - Systems and software engineering - System life cycle processes*, IEEE.

Isograph, Fault Tree+. Available at: https://www.isograph.com/software/reliability-workbench/fault-tree-analysis-software/ [Accessed January 6, 2019].

Jacobson, I., Grady, B. & Rumbaugh, J., 1999. *The unified software development process*, Addision-Wesley.

Jamshidi, M., 2008. System of Systems - Innovations for 21st Century. In *IEEE Region 10 and the 3rd International Conference on Industrial and Information Systems*. pp. 6–7.

Jensen, J.C., Chang, D.H. & Lee, E.A., 2011. A model-based design methodology for cyber-physical systems. In *7th International Wireless Communications and Mobile Computing Conference*. pp. 1666–1671.

Jilani, A.A.A., Usman, M. & Halim, Z., 2010. Model transformations in model driven architecture. *Universal Journal of Computer Science and Engineering Technology*, 1(1), pp.50–54.

Joshi, A. et al., 2006. Model-based safety analysis. , p.60.

Joshi, A. & Heimdahl, M.P.E., 2005. Model-based safety analysis of simulink models using SCADE design verifier. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. pp. 122–135.

Joshi, A., Vestal, S. & Binns, P., 2007. Automatic generation of static fault trees from AADL. In *Workshop on Architecting Dependable Systems, DSN*.

Joshi, A., Whalen, M. & Heimdahl, M.P.E., 2005. *Model-based safety analysis final report*, NASA Techreport.

Jouault, F. et al., 2008. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2), pp.31–39.

Kabir, S., 2017. An overview of fault tree analysis and its application in model based dependability analysis. *Expert Systems with Applications*, 77, pp.114–135.

Keating, C. et al., 2003. System of Systems Engineering. *Engineering Management Journal*, 15(3), pp.36–45.

Kelly, J. et al., 1998. Formal methods specification and verification guidebook for Software and computer systems Volume I: Planning and technology insertion. *Nasa*, I(July).

Kim, H.J. et al., 2010. Bridging the gap between fault trees and UML State Machine Diagrams for safety analysis. *Asia Pacific Software Engineering Conference, APSEC*, pp.196–205.

Kim, J., Ghang, S. & Lee, E., 2012. Run-time fault detection using automatically generated fault tree based on UML. In *Communications in Computer and Information Science*.

Kirstan, S. & Zimmermann, J., 2010. Evaluating costs and benefits of model-based development of embedded software systems in the car industry–Results of a qualitative Case Study. In *ECMFA Workshop C2M: EEMDD- from An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases*. pp. 18–29.

Knight, J.C., 2002. Safety critical systems: Challenges and directions. In *24th International Conference on Software Engineering, ICSE*. pp. 547–550.

Kurniawan, A., Harefa, B.B. & Sujarwo, S., 2014. Unified modeling language tools collaboration for use case, class and activity diagram implemented with html 5 and javascript framework. *Journal of Computer Science*, 10(9), pp.1440–1446.

Kuske, S. & Gogolla, M., 2002. An integrated semantics for UML class, object and state diagrams based on graph transformation. *Integrated Formal Methods Lecture Notes in Computer Science*, 2335, pp.11–28.

De La Vara, J.L. et al., 2016. Model-based specification of safety compliance needs for critical systems: A holistic generic metamodel. *Information and Software Technology*, 72, pp.16–30.

Laprie, J.C. et al., 1990. Definition and analysis of hardware and software fault-tolerant architectures. *Computer*, 23(7), pp.39–51.

Latsou, C., Dunnet, S.J. & Jackson, L.M., 2017. Automated generation of a petri net model: application to an en of life manufacturing process. In *27th European Safety and Reliability Conference, ESREL*. Portoroz, Slovenia.

Lazăr, C.L. et al., 2010. Tool support for fUML models. *International Journal of Computers, Communications and Control*, 5(5), pp.775–782.

Lemmon, E.J., *Beginning logic* 2nd ed., Chapman & Hall.

Leveson, N.G., 2012. *Engineering a safer world: Systems thinking applied to safety*,

Leveson, N.G. & Stolzy, J.L., 1987. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, SE-13(3), pp.386–397.

Liebel, G. et al., 2018. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software and Systems Modeling*, 17(1), pp.91–113.

Liggesmeyer, P. & Rothfelder, M., 1998. Improving system reliability with automatic fault tree generation. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. pp. 90–99.

Lisagor, O., Kelly, T. & Niu, R., 2011. Model-based safety assessment: Review of the discipline and its challenges. In *9th International Conference on Reliability,* *Maintainability and Safety, ICRMS*. pp. 625–632.

Liu, Q., 2010. Metamodel evolution through metamodel inference. In *ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. pp. 209–210.

Lowe, C. & Lowe, M., 2015. Using system architecture considerations to analyze allocation of functions. *Procedia Manufacturing*, 3, pp.1273–1280.

Majdara, A. & Wakabayashi, T., 2009. Component-based modeling of systems for automated fault tree generation. *Reliability Engineering and System Safety*, 94(6), pp.1076–1086.

Martins, L.A.L., Bastian, F.L. & Netto, T.A., 2012. Structural and functional failure pressure of filament wound composite tubes. *Materials and Design*, 36, pp.779–787.

Marvin, J.W. & Garrett, R.K., 2014. Quantitative SoS architecture modeling. In *Procedia Computer Science*. pp. 41–48.

Mhenni, F., Nguyen, N. & Choley, J.-Y., 2018. SafeSysE : A safety analysis integration in systems engineering approach. *IEEE Systems Journal*, 12(1), pp.161–172.

Mhenni, F., Nguyen, N. & Choley, J.Y., 2014. Automatic fault tree generation from SysML system models. In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. pp. 715–720.

Miguel, M.A. de et al., 2008. Integration of safety analysis in model-driven software development. *Software, IET*, 2(3), pp.260–280.

Mohagheghi, P. & Dehlen, V., 2008. Where ss the proof? - A review of experiences from applying MDE in industry. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. pp. 432–443.

Moir, I. & Seabridge, A., 2008. *Aircraft systems: Mechanical, electrical, and avionics subsystems integration* 3rd ed., John Wiley & Sons.

Murali, R., Ireland, A. & Grov, G., 2016. UC-B: Use case modelling with event-B. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. pp. 297–302.

Nakatani, T. et al., 2001. A requirements description metamodel for use cases. *8th Asia-Pacific Software Engineering Conference*, pp.251–258.

National Aeronautics and Space Administration, 2007. NASA systems engineering handbook. , NASA/SP-20(December), p.360. Available at: http://adsabs.harvard.edu/full/1995NASSP6105.....S [Accessed July 4, 2018].

National Instruments, 2008. Redundant system basic concepts. Available at: http://www.ni.com/en-gb/innovations/white-papers/08/redundant-system-basic-concepts.html [Accessed May 5, 2018].

Nativi, S. et al., 2015. Big data challenges in building the global earth observation system of systems. *Environmental Modelling and Software*, 68, pp.1–26.

Nieuwhof, G.W.E., 1975. An introduction to fault tree analysis with emphasis on failure rate evaluation. *Microelectronics Reliability*, 14(2), pp.105–119. Available at: https://www.sciencedirect.com/science/article/pii/0026271475900244 [Accessed April 22, 2018].

Object Management Group, 2007. OMG Systems Modeling Language (OMG SysML™) V1.0.

Object Management Group, 2017a. OMG Systems Modeling Language ™ Version 1.5. Available at: http://www.omg.org/spec/SysML/1.2/PDF/ [Accessed January 1, 2019].

Object Management Group, 2017b. OMG Unified Modeling Language (OMG UML) Version 2.5.1. Available at: https://www.omg.org/spec/UML/About-UML/ [Accessed January 1, 2019].

Object Management Group, 2017c. *Safety and reliability for UML - Request for proposal*,

Object Management Group, 2017d. Unified Modeling Language Version 2.5.1. *Object*

Management Group. Available at: https://www.omg.org/spec/UML/2.5.1 [Accessed June 2, 2018].

Oliveira, A.L. De et al., 2016. Model-based safety analysis of software product lines. *International Journal of Embedded Systems*, 8(5/6), pp.412–426.

Olmo, J. del et al., 2018. Model-based fault analysis for railway traction systems. *Modern Railway Engineering, IntechOpen*. Available at: https://www.intechopen.com/books/modern-railway-engineering/model-based-fault-analysis-for-railway-traction-systems [Accessed July 23, 2018].

Olmo, J. del et al., 2017. Model driven hardware-in-the-loop fault analysis of railway traction systems. In *IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics, ECMSM 2017*.

Oren, T., Mittal, S. & Durak, U., 2018. A shift from model-based to simulation-based paradigm: Timeliness and usefulness for many disciplines. *International Journal of Computer and Software Engineering*, 3(126).

Oren, T.I. & Zeigler, B.P., 2012. System theoretic foundations of modeling and simulation: A historic perspective and the legacy of A Wayne Wymore. *SIMULATION*, 88(9), pp.1033–1046.

Ortmeier, F. & Schellhorn, G., 2007. Formal Fault Tree Analysis - Practical Experiences. *Electronic Notes in Theoretical Computer Science*, 185(SPEC. ISS.), pp.139–151.

Pai, G.J. & Dugan, J.B., 2002. Automatic synthesis of dynamic fault trees from UML system models. *13th International Symposium on Software Reliability Engineering, ISSRE*, pp.243–254.

Paiboonkasemsut, P. & Limpiyakorn, Y., 2016. Reliability tests for process flow with fault tree analysis. In *IEEE 2nd International Conference on InformationScience and Security, ICISS*.

Papadopoulos, Y. et al., 2011. Engineering failure analysis and design optimisation with HiP-HOPS. *Engineering Failure Analysis*, 18(2), pp.590–608.

Papadopoulos, Y., HiP-HOPS. Available at: http://www.hip-hops.eu/index.php/contact-us [Accessed January 6, 2019].

Patel, P.E. & Patil, N.N., 2013. Testcases Formation Using UML Activity Diagram. *International Conference on Communication Systems and Network Technologies, CSNT*, pp.884–889.

Pietrantuono, R. & Russo, S., 2013. Introduction to safety critical systems. In *Innovative Technologies for Dependable OTS-Based Critical Systems*. Springer Milan, pp. 17–27.

Price, H.E., 1985. Allocation of functions in systems. *Human Factors*, 27(1), pp.33–45.

Prosvirnova, T. et al., 2013. The AltaRica 3.0 Project for Model-Based Safety Assessment. In *4th Workshop on Dependable Control of Discrete Systems The International Federation of Automatic Control, IFAC*. IFAC, pp. 127–132. Available at: https://linkinghub.elsevier.com/retrieve/pii/S1474667015339999.

Purba, J.H. et al., 2015. Fuzzy probability based fault tree analysis to propagate and quantify epistemic uncertainty. *Annals of Nuclear Energy*, 85, pp.1189–1199.

Radio Technical Commission for Aeronautics, 2011. *RTCA DO-178 Software Considerations in Airborne Systems and Equipment Certification*,

Rauzy, A., 2002. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78(1), pp.1–12.

Redmill, F., 1999. An introduction to the safety standard IEC 61508. *System Safety Society*, 35(1).

Reif, W., Schellhorn, G. & Thums, A., 2000. Safety Analysis of a Radio-based Crossing Control System Using Formal Methods. In *9th IFAC Symposium Control in Transportations Systems*.

Roland, H.E. & Moriarty, B., 1990. *System safety engineering and management* 2nd ed., John Wiley& Sons, Inc.

Royce, W.W., 1970. Managing the development of large software systems. In *IEEE*

WESCON. pp. 328–338.

Ruijters, E. et al., 2017. Uniform analysis of fault trees through model transformations. In *Annual Reliability and Maintainability Symposium*.

Saeki, M. & Kaiya, H., 2006. On relationships among models, meta models and ontologies. *6th OOPSLA Workshop on Domain-Specific Modeling, DSM*, p.140.

Saraoğlu, M. et al., 2017. ErrorSim: A tool for error propagation analysis of Simulink models. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. pp. 245–254.

Saraswat, S. & Yadava, G.S., 2008. An overview on reliability, availability, maintainability and supportability (RAMS) engineering. *International Journal of Quality and Reliability Management*, 25(3), pp.330–344.

Schellhorn, G., Thums, A. & Reif, W., 2002. Formal fault tree semantics. In *Integrated Design and Process Technology*.

Schmitt, P.H., 2003. UML and its meaning. *Vorlesungsskript, Universität Karlsruhe*.

Seidewitz, E., 2014. UML with meaning: Executable modeling in foundational UML and the alf action language. *ACM SIGAda's Annual International Conference High Integrity Language Technology, HILT*, pp.61–68.

Selic, B. & Gerard, S., 2014. *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*, Morgan Kaufmann.

Sharvia, S. & Papadopoulos, Y., 2015. Integrating model checking with HiP-HOPS in model-based safety analysis. *Reliability Engineering and System Safety*, 135, pp.64–80.

Simha Pilot, 2002. What is fault tree analysis? *Quality Progress*, 35, p.120.

Some, S.S., 2007. Specifying use case sequencing constraints using description elements. In *ICSE 2007 Workshops: 6th International Workshop on Scenarios and State Machines, SCESM*.

Souri, A., Ali Sharifloo, M. & Norouzi, M., 2011. Formalizing class diagram in UML. In *IEEE 2nd International Conference on Software Engineering and Service Science, ICSESS*. pp. 524–527.

Tajarrod, F. & Latif-Shabgahi, G., 2008. A novel methodology for synthesis of fault trees from MATLAB-Simulink Model. *World Academy of Science, Engineering and Technology*, 41(July), pp.630–636.

Thoma, A., Kormann, B. & Vogel-Heuser, B., 2012. Fault-centric system modeling using SysML for reliability testing. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*.

Thomas, N.L., 1968. *Modern Logic*,

Umeda, Y. et al., 1990. Function, behaviour, and structure. *Applications of Artificial Intelligence in Engineering V*, 1, pp.177–193. Available at: http://diyhpl.us/~bryan/papers2/Function, behaviour, and structure - Umeda - 1984.pdf.

Vemuri, K.K., 1999. Automatic synthesis of fault trees for computer-based systems. *IEEE Transactions on Reliability*, 48(4), pp.394–402.

Vesely, W.E. et al., 1981. *Fault Tree Handbook*,

Volk, M., Junges, S. & Katoen, J.-P., 2018. Fast dynamic fault tree analysis by model checking techniques. *IEEE Transactions on Industrial Informatics*, 14(1), pp.370–379.

Volkanovski, A., Čepin, M. & Mavko, B., 2009. Application of the fault tree analysis for assessment of power system reliability. *Reliability Engineering & System Safety*, 94(6), pp.1116–1127.

Waldecker, B. & Garg, V.K., 1991. Detection of strong predicates in distributed programs. In *3rd IEEE Symposium on Parallel and Distributed Processing*. pp. 692–699.

Wang, Y. et al., 2003. Algorithmic fault tree synthesis for control loops. *Journal of Loss Prevention in the Process Industries*, 16(5), pp.427–441.

Weiss, N.A., 2006. *A Course in Probability*, Pearson Education International.

Woodcock, J. et al., 2009. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), pp.1–36. Available at: http://portal.acm.org/citation.cfm?doid=1592434.1592436.

Woodward, D.M., 2018. *Space launch vehicle design: Conceptual design of rocket powered, vertical takeoff, fully expandable, and first stage boostback space launch vehicles*. University of Texas Arlington.

Wylie, M., Harvey, D. & Liddy, T., 2016. Model-based conceptual design through a system implementation - Lesson from a structured yet agile approach. In *Systems Engineering Test and Evaluation Conference*.

Wymore, A.W., 1993. *Model-based systems engineering* 1st Editio. A. T. Bahill, ed., Florida.

Xiang, J. et al., 2011. Automatic synthesis of static fault trees from system models. In *5th International Conference on Secure Software Integration and Reliability Improvement, SSIRI*. pp. 127–136.

Xiang, J., Futatsugi, K. & He, Y., 2004. Fault tree and formal methods in system safety analysis. In *4th International Conference on Computer and Information Technology, CIT*. pp. 1108–1115.

Xiang, J., Ogata, K. & Futatsugi, K., 2005. Formal fault tree analysis of state transition systems. *5th International Conference on Quality Software, QSIC*, pp.124–134.

Xiang, J. & Yanoo, K., 2010. Automatic static fault tree analysis from system models. In *Pacific Rim International Symposium on Dependable Computing*. pp. 241–242.

Xiaoxun, L. et al., 2011. A comparison of SAE ARP 4754A and ARP 4754. In *2nd International Symposium on Aircraft Airworthiness (ISAA)*. pp. 400–406.

Yakymets, N., Jaber, H. & Lanusse, A., 2013. Model-based system engineering for fault tree generation and analysis. *1st International Conference on Model-Driven Engineering and Software Development*.

Yuhua, D. & Datao, Y., 2005. Estimation of failure probability of oil and gas transmission pipelines by Fuzzy Fault Tree Analysis. *Journal of Loss Prevention in the Process Industries*, 18(2), pp.83–88.

Zeigler, B.P., 1985. The architecture of systems problem solving by George J. Klir. *International Journal of General System*, 13(1), pp.83–84.

Zeller, M., Ratiu, D. & Kai, H., 2016. Towards the adoption of model-based engineering for the development of safety-critical systems in industrial practice. In *International Conference Computer Safety, Reliability, and Security, SAFECOMP*. Springer, Cham, pp. 322–333. Available at: http://link.springer.com/10.1007/978-3-319-45480-1.

Zhang, J., Yao, H. & Rizzoni, G., 2017. Fault diagnosis for electric drive systems of electrified vehicles based on structural analysis. *IEEE Transactions on Vehicular Technology*, 66(2), pp.1027–1039.

Zhao, Z. & Petriu, D.C., 2015. UML Model to Fault Tree Model Transformation for Dependability Analysis. *Proceedings of the International Conference on Computer and Information Science and Technology*, (127), pp.1–9.

Zoughbi, G., Briand, L. & Labiche, Y., 2011. Modeling safety and airworthiness (RTCA DO-178B) information: Conceptual model and UML profile. *Software and Systems Modeling*, 10(3), pp.337–367.

Zwolinski, M., Yang, Z.R. & Kazmierski, T.J., 2000. Applying mutual information theory to behavioural analogue fault modelling. *International Journal of Electronics*, 87(12), pp.1461–1471.