THE UNIVERSITY OF HULL

Model Transformation for Multi-objective Architecture Optimisation for Dependable Systems

being a Thesis submitted for the Degree of

Doctor of Philosophy

in the University of Hull

by

Zhibao Mian BEng. MSc.

August 2014

For my parents …

Abstract

Model-based engineering (MBE) promises a number of advantages for the development of embedded systems. Model-based engineering depends on a common model of the system, which is refined as the system is developed. The use of a common model promises a consistent and systematic analysis of dependability, correctness, timing and performance properties. These benefits are potentially available early and throughout the development life cycle. An important part of model-based engineering is the use of analysis and design languages. The Architecture Analysis and Design Language (AADL) is a new modelling language which is increasingly being used for high dependability embedded systems development. AADL is ideally suited to model-based engineering but the use of new language threatens to isolate existing tools which use different languages. This is a particular problem when these tools provide an important development or analysis function, for example system optimisation. System designers seek an optimal trade-off between high dependability and low cost. For large systems, the design space of alternatives with respect to both dependability and cost is enormous and too large to investigate manually. For this reason automation is required to produce optimal or near optimal designs.

There is, however, a lack of analysis techniques and tools that can perform a dependability analysis and optimisation of AADL models. Some analysis tools are available in the literature but they are not able to accept AADL models since they use a different modelling language. A cost effective way of adding system dependability analysis and optimisation to models expressed in AADL is to exploit the capabilities of existing tools. Model transformation is a useful technique to maximise the utility of model-based engineering approaches because it provides a route for the exploitation of mature and tested tools in a new model-based engineering context. By using model transformation techniques, one can automatically translate between AADL models and other models. The advantage of this model transformation approach is that it opens a path by which AADL models may exploit existing non-AADL tools.

There is little published work which gives a comprehensive description of a method for transforming AADL models. Although transformations from AADL into other models have been reported only one comprehensive description has been published, a transformation of AADL to petri net models. There is a lack of detailed guidance for the transformation of AADL models.

This thesis investigates the transformation of AADL models into the HiP-HOPS modelling language, in order to provide dependability analysis and optimisation. HiP-HOPS is a mature, state of the art, dependability analysis and optimisation tool but it has its own model. A model

transformation is defined from the AADL model to the HiP-HOPS model. In addition to the model-to-model transformation, it is necessary to extend the AADL modelling attributes. For cost and dependability optimisation, a new AADL property set is developed for modelling component and system variability. This solves the problem of describing, within an AADL model, the design space of alternative designs. The transformation (with transformation rules written in ATLAS Transformation Language (ATL)) has been implemented as a plug-in for the AADL model development tool OSATE (Open-source AADL Tool Environment). To illustrate the method, the plug-in is used to transform some AADL model case-studies.

Keywords: MBE, AADL, HiP-HOPS, Dependability modelling, Dependability analysis, Multi-objective architecture optimisation, Model transformation, ATL.

List of abbreviations

AADL	Architecture Analysis and Design Language
ADAPT	from A**AD**L **A**rchitectural models to stochastic **P**etri nets through model **T**ransformation
ADLs	Architecture Description Languages
API	Application Programming Interface
AQOSA	Automated Quality-driven Optimisation of Software Architecture
ARP	Aerospace Recommended Practice
ATESST	Advancing Traffic Efficiency and Safety through Software Technology
ATL	ATLAS Transformation Language
AUTOSAR	AUTomotive Open System ARchitecture
BSCU	Brake System Control Unit
CFTs	Component Fault Trees
COMPASS	Correctness, Modelling and Performance of Aerospace Systems
DG	Direct Graph
EAST-ADL	Electronics Architecture and Software Technology - Architecture Description Language
EAST-EEA	Electronics Architecture and Software Technology – Embedded Electronic Architecture
EMF	Eclipse Modelling Framework
EMI	Electromagnetic Interference
FMEA	Failure Modes and Effects Analysis
FPTC	Fault Propagation and Transformation Calculus
FPTN	Failure Propagation and Transformation Notation

FT	Fault Tree
FTA	Fault Tree Analysis
GSPN	Generalised Stochastic Petri Net
HiP-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies
LABRI	Laboratoire Bordelais de Recherche en Informatique
M2M	Model to Model transformation
M2T	Model to Text transformation
MAENAD	Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles
MBE	Model-based Engineering
MCSs	Minimum Cut Sets
OSATE	Open-source AADL Tool Environment
PCS	Pre-collision System
PSM	Platform-Specific Model
RBD	Reliability Block Diagram
SAE	Society for Automotive Engineer
SEFTs	State Event Fault Trees
SEI	Software Engineering Institute
SysML	Systems Modeling Language
UML	Unified Modelling Language
WBS	Wheel Brake System
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Contents

Abstract iii

List of abbreviations v

Contents vii

List of figures: x

List of tables: xiii

Acknowledgments xiv

Author's declaration xvi

Chapter 1 Introduction 1

1.1 Research problem 1

1.2 Research contributions 3

1.3 Thesis structure 7

1.4 Publications 8

Chapter 2 Background 10

2.1 Languages for model-based engineering 10

2.1.1 Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL) 10

2.1.2 Architecture Analysis and Design Language (AADL) 12

2.2 Methods and tools for model-based dependability analysis 18

2.2.1 Methods for achieving system dependability 19

2.2.2 Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) – A tool for system dependability analysis and optimisation 24

2.3 System optimisation with dependability 26

2.4 Model transformation 29

2.4.1 Introduction to model and metamodel 29

2.4.2 Introduction to model transformation 30

2.4.3 Model transformation for model-based dependability analysis 32

2.4.4 Modularity in model transformation 35

2.4.5 Transformation languages 36

Chapter 3 Model transformation for the automatic generation of HiP-HOPS-oriented dependability analytical models from high level AADL dependable models 39

3.1 Model transformation overview 39

3.1.1 Translation of AADL component error model to HiP-HOPS failure expressions 41

3.1.2 Transformation of AADL connections to HiP-HOPS Lines 51

3.2 Model transformation method implementation using transformation rules 57

3.2.1 ATL rule implementation 58

3.3 Model transformation for AADL connections with error models 64

3.4 Model transformation design 66

3.4.1 Modularising transformation definition 67

3.4.2 Rule integration mechanisms 73

3.5 Summary 79

Chapter 4 Case study: temperature monitoring system 81

4.1 The temperature monitoring system 81

4.2 AADL modelling of temperature monitoring system 84

4.3 Model transformation from AADL to HiP-HOPS 87

4.4 Model transformation checking 92

4.5 The transformation cost estimation 93

4.6 Summary 94

Chapter 5 Multi-objective architecture optimisation for AADL dependable systems 96

5.1 Optimisation modelling for AADL dependability systems 96

5.1.1 Introduction 96

5.1.2 Method 97

5.1.3 Optimisation modelling in AADL 99

5.1.4 The definition of alternatives and optimisation properties 101

5.2 The model transformation from extended AADL system optimisation model to HiP-HOPS optimisation model 106

5.3 Summary 107

Chapter 6 Case study: aircraft wheel brake system 109

6.1 System description 109

6.2 Failure data 112

6.3 Dependability analysis of the wheel brake system model 115

6.4 Design optimisation 115

6.5 Optimisation results 116

6.6 Summary 117

Chapter 7 Conclusions and future work 119

References 124

List of figures:

Figure 1.1 The developed system dependability and optimisation analysis method between AADL and HiP-HOPS 6

Figure 2.1 Containment and reference associations in AADL metamodel (Source: SAE-AADL Meta Model/XMI V0.999, 2006) 15

Figure 2.2 The HiP-HOPS metamodel 25

Figure 2.3 An overview of model transformation process 31

Figure 3.1 An AADL system model consisting of five components, each with an associated error model, and six connections 40

Figure 3.2 HiP-HOPS model corresponding to the AADL model shown in Figure 3.1. The HiP-HOPS model consists of five components, and their associated fault tree expressions. There are four HiP-HOPS Lines (shown as small triangles) connecting the components 41

Figure 3.3 Error state machine for component A and D 42

Figure 3.4 The general form of the guard_in property 44

Figure 3.5 The fault trees generated from the guard_in property shown in Figure 3.4. The top events i.e. e1 to e5 are qualified by port pi. The trees shown above are for the port pi. There is one set of trees for each port p1 to pk in the applies to clause 46

Figure 3.6 The formal algorithm for transforming AADL guard_in (guard_out) property into HiP-HOPS fault trees 48

Figure 3.7 Left hand figure shows the AADL connection metamodel and right hand figure shows a corresponding HiP-HOPS Line metamodel 52

Figure 3.8 Left hand figure shows an AADL model with six connections, small circles. Right hand figure shows a corresponding HiP-HOPS model with four Lines, small triangles 54

Figure 3.9 The formal algorithm for transforming AADL Connections into HiP-HOPS Lines 56

Figure 3.10 Rule transforming AADL Connections to HiP-HOPS Lines 59

Figure 3.11 The helper function that returns all connections that share the same destination port 60

Figure 3.12 The helper function that returns all connections in a Line. A list of connections is constructed for each of error propagated to the destination port..... 60

Figure 3.13 Rule transforming AADL top system instance to HiP-HOPS top system..... 61

Figure 3.14 Rule transforming an AADL component instance to a HiP-HOPS component..... 62

Figure 3.15 Rule transforming AADL sub-system to HiP-HOPS component 62

Figure 3.16 The overview of transformation design, an in-memory model to model transformation is followed by conversion of the model to XML format for input into HiP-HOPS 63

Figure 3.17 Two AADL components `A` and `B` and one AADL connection `C` with associated error model..... 65

Figure 3.18 The connection `C` with associated error model shown in Figure 3.17 is transformed to two new connections and an intervening ConnectionComponent called `CC` 66

Figure 3.19 The algorithm for transforming AADL Connection with associated error model to two new AADL Connections and an intervening connection component called `CC` 66

Figure 4.1 The top level structure of the temperature monitoring system 82

Figure 4.2 The architecture of sub-system `SensorInput`..... 83

Figure 4.3 The architecture of sub-system `TempProcess`..... 83

Figure 4.4 The architecture of sub-system `TempLimLog`..... 84

Figure 4.5 Partial AADL architecture for temperature monitoring system 85

Figure 4.6 Partial AADL implementations associated with error models 86

Figure 4.7 Partial AADL error model type and implementation declaration..... 87

Figure 4.8 The result HiP-HOPS model of process `SensorInput_P1` transformed from AADL models..... 90

Figure 4.9 The screenshot of resultant FMEA of temperature monitoring system generated by HiP-HOPS from the initial AADL model after transformation: Direct effects 91

Figure 4.10 The screenshot of resultant FMEA of temperature monitoring system generated by HiP-HOPS from the initial AADL model after transformation: Further effects of component `SensorInput_P1` 92

Figure 5.1 AADL text description of subcomponents and associated optimisation properties for the pre-collision system 99

Figure 5.2 The AADL property set declaration for some example properties of a person 101

Figure 5.3 The AADL property set declaration syntax 102

Figure 5.4 The use of AADL property set to define optimisation properties 103

Figure 5.5 The use of AADL property set to define a new `Objectives` property type 103

Figure 5.6 The use of AADL property set to define the `List_of_Alternatives` property. This property specifies the alternative components for a given component 104

Figure 5.7 The association of defined optimisation properties for process type `CR_PP01` and different implementations of this type 105

Figure 5.8 The overview of the extended transformation design 106

Figure 5.9 The ATL rule for transforming AADL alternative classifier to HiP-HOPS Alternative 107

Figure 6.1 The basic system structure of aircraft wheel brake system 111

Figure 6.2 The AADL description for the aircraft wheel brake system 111

Figure 6.3 AADL error model type definition and error model implementation for component `Power`, `GreenValve` and `BSCU` 113

Figure 6.4 Associated AADL error model, guard_in and guard_out error model properties for component `BSCU` and alternative implementations of this component 114

Figure 6.5 The Pareto front optimal solutions 116

List of tables:

Table 6.1 The failure rates and costs data for the component alternatives 115

Table 6.2 The three solutions which satisfy the constraint: Risk ≤0.000015, Cost ≤120..... 117

Acknowledgments

I would like to express my immense gratitude to Dr. Leonardo Bottaci, my research supervisor, for his patient guidance, expert advice and encouragement throughout this difficult project. He taught me step-by-step from very small English grammar to difficult mathematical algorithm. My completion of this dissertation could not have been accomplished without his support and help.

I offer my sincere appreciation to other staff members involved in my research committee. I cannot express enough thanks to Prof. Yiannis Papadopoulos, my committee chair, for his introduction to the safety and reliability analysis field and for his continued support and enthusiastic encouragement. I would also like to thank Dr. Neil Gordon, my technical expert, for his valuable and useful critiques of this research work.

I would like to acknowledge and extend my heartfelt gratitude to my colleagues for their willingness to discuss with me about my premature ideas for my research: Dr. Martin Walker, Dr. David Parker, Dr. Septavera Sharvia, Dr. Nidhal Mahmud, Dr. Yan Zhang, Dr. Amer Dheedan, Dr. Shawulu Nggada, Dr. Nabil Abu Hashish, Dr. Nongnuch Poolsawad, Mr. Ernest Edifor, Mr. Luis Azevedo, Mr. Sohag Kabir, Miss Lisa Moore, Mrs. Lamis Farah Iqab Al-Qora'n, and Mr. M. Mostafizur Rahman. A special thanks to Prof. Ping Jiang, Dr. Qingde Li, Dr. Bing Wang and Dr. Hantao Liu for their friendly care to my research and life.

I am particularly grateful for the assistance given by the following support staff in my department: Mrs. Helen El-Sharkawy, Mrs. Lynn Morrell, Mrs. Amanda Millson, Mrs. Jo Clappison, Mr. Simon Grey, Mr. Adam Hird, and Mr. David Glover.

To Dr. Peter Feiler, Dr. Ana-Elena Ruguna, Dr. Matthias Biehl and Dr. Dejiu Chen who has guided me through internet and answered many research questions regard to AADL error modelling and transformation and generally bailed me out when every time I was struggled with my research.

This dissertation would have been impossible without the financial support from the Departmental Scholarship supplied by the Department of Computer Science in the University of Hull and the EU Project MAENAD (Grant 260057).

My eternal thanks to Dr. Yan Zhuang – thank you for your guidance, support and help to my study in the UK. I would never reach the step I am standing today without your kindness help and continued support.

I wish to acknowledge the help and extraordinary support provided by Zhaoyang Wang who enlightened and encouraged me to cross the most difficult study time in Hull in the UK. You deserve all my thanks.

A special thanks to my girlfriend Miss Yi Zhou for her patient love and encouragement and also for understanding what is generally not understandable. Without you, this research would have taken years off my life. My heartfelt thanks.

Most especially to my roommate Miss Marion Joassin for French teaching and English practice, and for the help in Paris.

I wish to thank my friends, classmates, colleagues and all the others who perhaps deserve a mention helped me but aren't listed here.

Finally, I wish to thank my parents, my brothers and my sisters for their continued support and encouragement throughout my study and for teaching me many useful lessons for life.

Author's declaration

I declare that the material contained in this thesis represents original work undertaken solely by the author. The various aspects of the work covered in this material have been presented in a number of international conferences and scientific publications.

The work on model transformation in Chapter 3 was presented in Mian et al. (2012) and Mian et al. (2013c). Parts of the introduction (Chapter 1) and summarised background (Chapter 2) and parts of the model transformation from AADL to HiP-HOPS shown in Chapter 3 were presented in Mian et al. (2013c). The work on the optimisation modelling for AADL dependable systems shown in Chapter 5 was presented in Mian and Bottaci (2013), Mian et al. (2013a) and Mian et al. (2013b). The case study used in Chapter 6 was presented in Mian et al. (2013c).

Chapter 1 Introduction

1.1 Research problem

Model-based engineering (MBE) is used to design systems in which models are the central artifacts through the lifecycle of a system development process. Model-based engineering, as argued in (Feiler and Gluch, 2012), allows a systematic analysis of system architecture early and throughout the development life cycle. This can provide higher confidence that the system will meet specific design goals such as dependability, timing and performance-related requirements. Furthermore, model-based engineering enables a more efficient development and system integration process.

These model-based engineering approaches need to be supported by languages and tools in order to ensure that the designed system complies with its requirements. Recent work in the area of model-based engineering approach has focused on the development of languages and notations that aim to progressively refine requirements models and design models to automatically drive the development and then verification of complex systems. These include general purpose modelling languages such as Unified Modelling Language (UML) (OMG, 2005) and SysML (OMG, 2012) which enable system architecture design, behaviour modelling and allocation of functions to software and hardware resources. Recently, Architecture Description Languages (ADLs) such as AADL (Architecture Analysis and Design Language as described in SAE-AS5506 (2006)) and EAST-ADL (Electronics Architecture and Software Technology - Architecture Description Language as described in MAENAD project (2013)) have gaining widespread acceptance in aerospace, automobile and avionics industries for model-based design of complex embedded systems.

Beyond the modelling of "normal" behaviour, these languages also include error modelling concepts to enable dependability-related analysis. Dependable systems are those systems that have high dependability (safety, reliability, availability, maintainability) requirements. For example, safety critical systems such as transport and medical engineering systems need to be dependable because their failure or malfunction may harm people or the environment. Dependability of a computing system as defined in (Avizienis et al., 2001) is the ability to deliver service that can justifiably be trusted.

System dependability is a vital factor in system quality control since it quantifies system failures. It is important to differentiate between the terms - fault, error and failure as defined in (Avizienis et al., 2001; Popic, 2005). A fault is a defect in system implementation that causes an

error. A fault is said to be active when it produces errors otherwise it is dormant. An error is a design flaw embedded in a system that when activated may cause a failure. A failure is any departure of system behaviour from user needs and it occurs when an error reaches the service interface and alters the service. The ways in which a system can fail are defined as its failure modes, e.g., timing, value failures (Avizienis et al., 2001).

The AADL Error Model Annex document (SAE-AS-5506/1, 2006), published by the Society for Automotive Engineer (SAE), focuses the capabilities for dependability modelling. One of the advantages of the Error Model Annex is that it supplies a notation used for modelling the failure information on the original AADL architecture model. This kind of error annotation enables the dependability analysis to consider both intra- and inter-component error models, which is considered important for dependability analysis (Joshi et al., 2007). An AADL architecture specification including error models supports a dependability-oriented view of the system.

Architecture description languages are ideally suited to model-based engineering; but the use of new languages threatens to isolate existing tools which use different languages. This is a particular problem when these tools provide an important development or analysis function. System optimisation is such a function.

System optimisation is an important part of system development and should benefit from a model-based engineering approach. System optimisation, as argued in Walker et al. (2013), is difficult. Optimisation requires the exploration of potentially huge design spaces. In addition, system designs typically need to balance multiple demands (e.g., dependability and low cost), which introduces the complication of multiple optimal designs. To overcome these difficulties, some automation is needed.

Architecture description languages can enable automation and consistency in a model-based development process but as described in Walker et al. (2013), architecture description languages are not yet fully developed to ensure effective assessment and satisfaction of architecture optimisation. Currently, model-based engineering lacks tools for system optimisation and tools for new modelling languages take time to develop. There is also a lack of analysis techniques and tools that can perform a dependability analysis and optimisation of AADL models. Some analysis and optimisation tools are available but they are not able to accept AADL models since they use different modelling language.

A cost effective way of adding system dependability analysis and optimisation to models expressed in AADL is to exploit these capabilities of existing tools. By using model

transformation techniques, one can automatically translate between AADL models and other models. The advantage of this model transformation approach is that it opens a path by which AADL models may exploit existing non-AADL tools.

A number of transformation approaches have been proposed to produce dependability analysis of AADL models. Joshi et al. (2007) produced a static fault tree generator prototype based on AADL models. This work has been extended by Dehlinger and Dugan (2008), so that dynamic fault trees are generated automatically from AADL models. In Rugina et al. (2008), an AADL dependability model is transformed into a Generalised Stochastic Petri Net (GSPN) by applying model transformation rules. The resulting GSPN can be processed by existing tools. Although transformations from AADL into other models have been reported only one comprehensive description has been published, a transformation of AADL to petri net models (Rugina, 2007). There is little published work which gives a comprehensive description of a method for transforming AADL models. The designers therefore face the following challenge: How to transform AADL models to other models for dependability analysis and optimisation of AADL models?

1.2 Research contributions

If AADL models could be transformed into the models used by other model-based methods and tools then it would extend the range of analysis that could be done on AADL models. Model transformation is potentially very valuable but it is also complex. Mens and Van Gorp, (2006) and Biehl, (2010) argue that the semantics between source model and target model should be unchanged during the transformation. This is called semantics-preserving program transformation (Yang et al., 1992). In order to ensure a semantics-preserving program transformation, an in depth concept mapping between the source and target models is required. Moreover, the target model often makes explicit properties that are implicit in the source model. In this situation, the transformation design may need to search from different level of source models and finally create one single element in target model. At this transformation stage, however, it may difficult to create the exact target element since a complicated algorithm or function is needed to transform multiple elements to a single one. This thesis investigates how model transformation can be used to advance model-based dependability analysis and optimisation to models expressed in AADL. The research hypothesis is that model transformation is a cost effective way to maximise the utility of model-based dependability analysis and optimisation to AADL models because it provides a route for the exploitation of mature and tested tools in the AADL context. More specifically, in order to introduce system dependability analysis and optimisation into AADL, this thesis argues for a model-based

dependability modelling and analysis and architecture optimisation method through model transformation. In order to evaluate the validity of the hypothesis, the model transformation method is implemented and integrated as a plug-in into the AADL development environment OSATE (Feiler et al., 2006). Several case studies are also developed and tested based on this plug-in so as to check the developed transformation method.

The thesis summarises the current system optimisation methods and tools for AADL models. One tool is ArcheOpterix (Aleti et al., 2009), which is based on AADL and potentially allows automatic optimisation of AADL specifications. Another tool is AQOSA (Automated Quality-driven Optimisation of Software Architecture) (Li et al., 2011, Etemaadi and Chaudron, 2012), for automated software architecture optimisation that allows multiple quality attributes (e.g. data flow latency, safety and cost). Limitations in the exploitation of current system optimisation methods are identified. A summary on model transformation is also examined. Following on from this, the thesis develops a transformation based method to introduce optimisation via an existing optimisation tool. The approach advocated in this thesis is to exploit existing dependability analysis and architecture optimisation techniques and tools. The challenge is to ensure that such tools are properly integrated into a model-based engineering process.

The method is demonstrated using the AADL language and AADL Error Model Annex. The system architecture including components and its subcomponents and connections between components are described by using the AADL language. The dependability-related data is described with AADL Error Model Annex and then associated to the components and connections. The transformation takes advantage of the existing mature dependability analysis and architecture optimisation technique HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) (Adachi et al., 2011). HiP-HOPS models the topology of a system in terms of components and data transactions among those components. HiP-HOPS supplies a fast algorithm for automatic generation of dependability analysis artifacts such as fault trees and Failure Models and Effects Analyses (FMEAs). Moreover, using genetic algorithms, a novel extension is able to solve multi-objective (cost and dependability) optimisation problems.

Although HiP-HOPS is a state-of-the-art model-based system dependability and architecture optimisation analysis technique, unfortunately, HiP-HOPS requires that the system to be optimised is expressed as a HiP-HOPS model using the HiP-HOPS modelling language. HiP-HOPS requires, as input, the local failure behaviour of the system components together with the inter-component failure propagation behavior. For optimisation, component variability information is also required. The integration of tools such as HiP-HOPS into an AADL model-based engineering environment requires that these tools have suitable access to the system

model. Without proper integration, additional system information must be input at additional cost and risk of inconsistency.

This problem can be overcome by transforming the AADL model into an equivalent HiP-HOPS model. More specifically, the AADL dependable model must be transformed into a HiP-HOPS model that captures the relevant component structure, topology and local failure information required for the HiP-HOPS analysis. The transformation design is driven by considering the needed information from the target HiP-HOPS model and the location of that information in the source AADL model. The thesis shows the integration of AADL model with HiP-HOPS model and implements the transformation between the two models by defining a set of identified transformation rules. Through the transformation design the thesis contributes as follows:

1. Model transformation method for automatic optimisation of AADL dependable models

A method that integrates system dependability design and dependability analysis and architecture optimisation based on dependability and cost is the main contribution of this thesis. The method is automated and based on model transformation.

The thesis presents rules for transforming an AADL model which contains an error state machine into a fault tree by integrating the state machine to fault tree conversion algorithms shown in Mahmud et al. (2010, 2011) to the model transformation method. The transformation rules have been automated by implementing them in the ATL language (Jouault et al., 2008). The thesis shows the feasibility of the automation by implementing the tool AADL2HiP-HOPS, which has been developed to implement the model transformation and has been integrated as a plug-in into the AADL development environment OSATE (Feiler et al., 2006).

Figure 1.1 shows the developed system dependability analysis and optimisation method in this research. In this method, once a dependable system with its optional designs is identified then it can be implemented by AADL. AADL is used as the notation for capturing the system architecture model and the possible component faults and failure modes (error model). The AADL modelling language has no direct support for optimisation of models. This has motivated the development of a new set of properties to extend AADL models (see the rectangle box with dotted lines shown in Figure 1.1). These properties allow an AADL model to capture the information required to perform model optimisation. In particular, they can represent the relevant optimisation properties and allow component alternatives to be specified.

The system architecture model annotated with error model of components and optimisation model is defined as a system dependability and optimisation model in this thesis. The model transformation from AADL to HiP-HOPS is able to use this information to produce a HiP-

HOPS optimisation model. The model which is then used for dependability analysis including automatic synthesis of fault trees, generation of FMEAs tables and multi-objective architecture optimisations analysis based on dependability and cost.

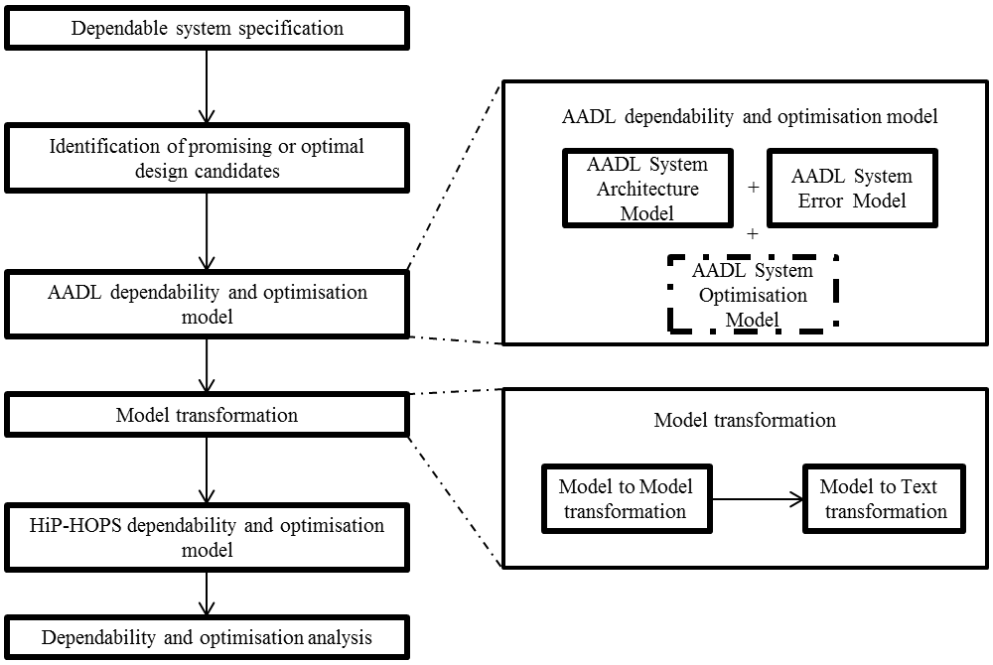

Figure 1.1 The developed system dependability and optimisation analysis method between AADL and HiP-HOPS

The model transformation method developed in this thesis is based on a synthesised view of several works (Joshi et al., 2007; Biehl et al., 2010; Rauzy, 2002; and Mahmud et al., (2010, 2011)). The method for extracting the AADL dependability model is similar as the method shown in Joshi et al. (2007). However, the thesis adopts the state machine to fault tree conversion algorithm shown in Mahmud et al. (2010, 2011) rather than using Direct Graph (DG) shown in Joshi et al. (2007). This conversion algorithm ensures that the temporal characteristics (Mahmud et al., 2010) can also be obtained. The transformation method is similar as shown in Biehl et al. (2010). The transformation concepts shown in Biehl et al. (2010) are used because their work is related to this research. The transformation in this thesis, however, is from a different model (AADL) and the scope is broader, aiming to encompass not only the dependability analysis but also the optimisation and temporal analysis (Mahmud et al., 2010) capabilities of HiP-HOPS.

2. Extending AADL modelling with optimisation attributes

Though AADL supports the description of system dependability by using the Error Model Annex, there are no AADL features that can be directly used for model optimisation. The second contribution of this thesis is to extend the AADL modelling features by adding an

optimisation modelling scheme. This includes a high level way to define the alternatives for a component and optimisation needed properties such as component cost, system optimisation objectives.

1.3 Thesis structure

The structure of the remaining chapters of this thesis is described below:

Chapter 2 introduces the relevant background to this research. It is divided into four sections. The first section introduces a few well-known modelling languages in the context of dependability analysis. More detailed modelling concepts are given to selected test language: AADL. The second section describes methods and tools used for achieving system dependability analysis. More detailed introduction is given to a well-known fault tree based dependability analysis technique - HiP-HOPS, for system dependability modelling and analysis. The third section introduces the system architecture optimisation in the context of dependability and cost has been selected as the combined optimisation objective. The third section also summarises the relevant work for system architecture optimisation. This section also identifies a key research problem of this thesis, i.e., how to integrate system architecture optimisation constructs into the AADL modelling language. Model transformation has been identified as a cost effective way to tackle this problem. The fourth section introduces concepts of model transformation and summarises the current model transformation methods used in the context of dependability analysis. Limitations in these transformation method i.e., lack of detailed report of transformation method are identified.

Chapter 3 describes methods for performing automatic model transformation from AADL models to HiP-HOPS models. This chapter focuses on model to model transformations, especially the transformation rules identified and the implementation of these transformation rules in model transformation language ATL. Chapter 3 also describes the design of model transformation considering modularity to achieve two identified design qualities: reusability and adaptability. From the transformation experience gained in this research, this chapter describes a model transformation guideline for rule-based (general standalone rule with source and target) model transformations. This guidance is illustrated by considering the transformation implemented in the model transformation language ATL. The chapter shows how adaptability and reusability in transformation designs can be achieved through modularisation. The chapter shows the rule implementations of the transformation definitions by applying three ATL rule integration mechanisms, i.e., implicit rule calls, explicit rule calls and rule inheritance. Every mechanism has its advantages and disadvantages to integrate rules together. Implicit rule calls relies on indirect rule dependencies and usually lead to a low coupling between rules. It is thus

would be chosen when the adaptability and reusability are set as the main quality attribute for the transformation definitions. For explicit rule calls, rules are integrated with explicit rule scheduling of function and method calls, where a rule may be directly invoked by another rule. This is particular useful when a transformation algorithm is needed to explicitly generate target model element from imperative code. Rule inheritance mechanism enables one rule to inherit functionalities from another rule. This allows one rule can be reused for many times. However, rule inheritance may lead to tight coupling between rules because it refers to rules by name.

Chapter 4 shows a case study, where the model transformation techniques described in Chapter 3 are applied to a complex temperature monitoring system. Based on the benefit of this transformation, the fault tree and FMEA analysis results are analysed. This system is designed in order to verify all transformation rules identified in Chapter 3.

Chapter 5 discusses the developed AADL optimisation modelling methods and rules for transforming AADL optimisation model to HiP-HOPS model.

Chapter 6 applies the optimisation modelling method and optimisation transformation rules to a safety critical system. Through the analysis of this case study, this chapter highlights the value of this research.

Chapter 7 concludes the thesis and lists the potential future work.

1.4 Publications

The following is a list of publications in which materials from this research work have been presented:

Mahmud N and Mian Z, 2013, Automatic Generation of Temporal Fault Trees from AADL Models, *European Safety and Reliability conference (ESREL 2013)*, Amsterdam.

Mian Z and Bottaci L, 2013, Multi-objective Architecture Optimisation Modelling for Dependable Systems, *the 4th IFAC Workshop on Dependable Control of Discrete Systems (DCDS2013)*, York University, UK.

Mian Z, Bottaci L and Papadopoulos Y, 2013b, Multi-objective Architecture Optimisation for Dependable Systems, extended abstract for the *3rd International Workshop on Model Based Safety Assessment, IWMBSA'2013*, Versailles, France.

Mian Z, Bottaci L, Papadopoulos Y and Adachi M, 2013a, Multi-objective Architecture Optimisation for Dependable Systems, *Reliability Engineering & System Safety Journal*, Elsevier, Accepted.

Mian Z, Bottaci L, Papadopoulos Y and Biehl M, 2012, System Dependability Modelling and Analysis Using AADL and HiP-HOPS, *14th IFAC Symposium on Information Control Problems in Manufacturing*, Bucharest, Romania.

Mian Z, Bottaci L, Papadopoulos Y, Sharvia S and Mahmud N, 2013c, Model Transformation for Multi-objective Architecture Optimisation of Dependable Systems, In Zamojski W (Ed.) Dependability problems of complex information systems, *Springer Verlag*.

Chapter 2 Background

Model-based engineering is a design approach in which models are the central artifacts throughout the lifecycle of the system development process. A model as described in Czarnecki and Helsen (2006) is an abstraction of a system and its environment. By the use of models, the developers and other stakeholders can effectively address concerns about the system or effects when there is a change on the system before the system is implemented. Model-based engineering depends on a common model of the system, which is refined as the system is developed. As a system model is continuously refined, the model includes finer analysis of the system and provides more detailed insights into the quality (e.g., dependability and performance) of the design. Model-based engineering, as argued in (Feiler and Gluch, 2012), supplies a systematic analyses of system architecture model early and throughout the development life cycle and can provide higher confidence that the implemented system will meet specific design goals such as dependability, timing and performance-related requirements. Furthermore, model-based engineering approach enables a more efficient system development and system integration process. For example, Biehl et al. (2010) argues that the integration (between the automotive domain to the safety domain through model transformation) help the safety engineers to perform the safety analysis early in the development process to fulfil safety goals with lower effort and cost. The transformation is automated and therefore can be repeated efficiently in the course of design iterations.

2.1 Languages for model-based engineering

Model-based engineering approaches must be supported by languages. Architecture and analysis languages as introduced in (Feiler and Gluch, 2012), have well-defined semantics that include specification of architecture and a standardized mechanism for semantically consistent extensions. It provides the basis for automatically deriving analytical models to validate non-functional requirements (such as dependability and performance) through formal analysis and simulation.

The following subsection introduces a few of the better-known architecture modelling languages and how they may be useful for dependability analysis.

2.1.1 Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL)

EAST-ADL (Walker et al., 2013; MAENAD project, 2013) is an architecture description language mainly developed for the design and analysis of embedded systems in automotive

industry. It is developed in several European research projects including the EAST-EEA project, the ATESST and ATESST2 projects and most recently the European FP7 MAENAD project. The language is currently maintained by the EAST-ADL Association (2013 (a)). This language uses concepts from SysML, UML and AADL but adapted for automotive needs including vehicle features, functions, requirements, variability, software and hardware components and communication. The language is aligned on the automotive standard AUTOSAR (2014) and is structured in five abstract levels, i.e., vehicle level, analysis level, design level, implementation level and operational level. These abstract levels cover all development information (e.g. the internal structure and external interfaces of system being modelled) from early analysis to late implementation.

In addition to the five abstract levels defining the structural view of a system being modelled (nominal model), the language also defines concepts for defining abnormal behaviours of a system. EAST-ADL contains a set of dependability packages (i.e., elements related to dependability) for defining and formalising system dependability-related requirements (EAST-ADL Association, 2013 (b)). These include ErrorModel, SafetyRequirements and SafetyConstraints. The ErrorModel sub-package defines concepts for representing abnormal behaviours (e.g. component errors and their propagations) of a system in its operation. The error behaviours are represented as a separated view but orthogonal to the nominal system model. The benefit of this separation of concern, as described in EAST-ADL Association (2013 (b)), avoids the undesired effects of error modelling e.g. the risk of mixing erroneous and nominal behaviour in term of reuse and system synthesis (e.g. code generation).

In the ErrorModel sub-package, the concepts of Fault and Failure are distinguished in terms of the perspective of the component. For example, the incoming flaw or an internal flaw is modelled as a Fault that may result in a component Failure. An output error propagation from a component is defined as a Failure. More details of how EAST-ADL ErrorModel are used to specify and model the failure behaviour of a dependability-critical system can be found in EAST-ADL Association (2013 (b)).

The SafetyRequirement package contains elements for organising standard safety requirements in accordance with ISO 26262. For instance, the SafetyGoal defines the top-level safety requirement specified in ISO 26262 in order to avoid or reduce the risk of hazardous event. The SafetyConstraints package contains constructs and elements for defining safety constraints. For example, the QuantitativeSafetyConstraint represents the quantitative integrity constraints (failure rate) on a fault or failure.

The variability modelling in EAST-ADL enables the language to define the optimisation space (Walker et al., 2013) and thus potentially extend the language to have system optimisation capability. The individual optimisation candidates are produced through configuration and variability resolution.

2.1.2 Architecture Analysis and Design Language (AADL)

2.1.2.1 Introduction to AADL

AADL (SAE-AS5506, 2006; Feiler and Gluch, 2012) is an example of an architecture and analysis language which is increasingly being used for high dependability embedded systems development. AADL is both a textual and graphical language with component-based modelling concepts designed to represent embedded software systems. The development of the AADL started in 1999 and is patterned after MetaH, a research prototype of a language and tool developed by the Honeywell Technology Centre for analysis and generation of embedded real-time systems. AADL was originally published in November 2004 and the revised standard became available in January 2009. It has been improved and published as an international standard by the Society of Automotive Engineer (SAE). Compared with other languages, the language as described in (Feiler and Gluch, 2012), provides more advanced support for analysing quality attributes. This includes the Behaviour Annex standard (SAE-AS5506/2, 2011) and the Error Model Annex standard (SAE-AS5506/1, 2006) for AADL. The Behaviour Annex standard extends the AADL to specify the component interaction behaviour with further precision to address safety aspects of the system. The Error Model Annex standard extends the AADL to specify fault behaviour and error propagation to address reliability aspects of the system architecture.

2.1.2.2 Comparison between AADL and UML, SysML and EAST-ADL

Feiler and Gluch (2012) compared the AADL, UML and SysML for model-based development of embedded systems. They highlighted that AADL solves the issue that UML has in addressing the non-functional characteristics of performance-critical systems in PSM (Platform Specific Model) by directly addressing the performance-critical aspects of software system architecture. The main difference between AADL and SysML is that AADL is a textual modelling language with graphical representations but SysML is a graphical language based on UML. SysML incorporates two new diagrams (the requirements and the parametric diagram) that are applicable to system engineering. These diagrams, however, are less effective in capturing and representing the execution of a computer's runtime environment including threads, process, and their allocation to different processors (Feiler and Gluch, 2012). For capturing the architecture

of an embedded system, AADL, focuses on the interaction between the physical system architecture, the runtime architecture of the embedded software and the computer hardware. SysML, however, focuses on the system as a whole in the context of its operational environment with the computer hardware as one component and the software implementing system functionality. Compared with SysML, AADL provides more insights into the management and control of a system in terms of timing, reliability and safety behaviour of the embedded software system (Feiler and Gluch, 2012). Furthermore, as noted in (OMG, 2012; Feiler and Gluch, 2012), SysML lacks a defined foundation for rigorous formal analysis.

Johnsen and Lundqvist (2011) compared and investigated the two ADLs - East-ADL and AADL regarding the level of support provided by these two ADLs to developers for developing dependable systems. The authors (Johnsen and Lundqvist, 2011) summarised that both EAST-ADL and AADL have their advantages and disadvantages. Compared to AADL, the authors highlighted that the metamodel (Jouault et al., 2008) of EAST-ADL has higher abstraction levels to describe systems. AADL, however, models a system using concrete system elements e.g., process and thread. This provides less freedom of the structure and designers may find it's hard to understand how the functionality is obtained in the implementation. Since EAST-ADL has a higher abstraction level, the gap between an architecture description artefact and its implementation is larger in EAST-ADL compared to AADL. EAST-ADL tends to focus on understandability and communication of systems whereas AADL tends to be more suitable for analysis tools.

In this thesis, AADL was selected as a suitable dependability modelling language to illustrate the developed model transformation method. The reasons for selecting AADL are: First, compared with other languages, AADL as described in (Feiler and Gluch, 2012), provides more advanced support for analysing quality attributes. The AADL Error Model Annex standard specifies system fault behaviour and error propagation to address reliability aspects of the system architecture. Thus, it is well fitted in the context of this research, i.e., system dependability analysis and optimisation. Second, compared with EAST-ADL, AADL does not provide a variability management mechanism that directly supports defining design alternatives (which is useful for representing the design space for system optimisation). This is taken into consideration when choosing between AADL and EAST-ADL and has motivated this research to develop a new AADL property set for modelling component and system variability (see Chapter 5). Third, though AADL is increasingly being used as an international standard for high dependability embedded systems development there is still a lack of analysis techniques and tools that can perform a dependability analysis and optimisation of AADL models (as argued in section 2.3). This motivates the thesis to select AADL as a modelling language and develops a

model transformation based method to enable AADL has the capability of system dependability analysis and optimisation.

The next sub-sections give an overview of AADL language and the dependability modelling with AADL Error Model Annex. Since system dependability is concerned as one key system attribute in this research, the introduction focuses on related architecture and error modelling and error propagation in AADL.

2.1.2.3 AADL metamodel

The SAE AADL metamodel (SAE-AADL Meta Model/XMI V0.999, 2006) defines the structure of AADL models, i.e., an object representation of AADL specifications that corresponds to a semantically decorated abstract syntax tree. This is done through a set of related class specifications using the Eclipse Modeling Framework (EMF) Ecore (Steinberg et al., 2009) notation.

Objects in an AADL metamodel are related to each other through two types of associations, a containment association, and a reference association. Figure 2.1 shows the containment and reference associations between objects in AADL model.

A containment association specifies that one object is part of another object. A containment association is represent by using a line with a diamond at the container end and an open arrowhead at the contained end (see the containment of ProcessSubcomponents in ProcessImpl in Figure 2.1).

A reference association specifies that objects are accessible from other objects either uni-directionally or bi-directionally. Unidirectional reference associations are shown as a line with an arrowhead and a label at the destination (see the reference from ProcessSubcomponent to ProcessClassifier in Figure 2.1). Bi-directional reference associations are shown as a line (without an arrowhead) with labels on both ends (see the reference between ProcessImpl and ProcessType in Figure 2.1).

If one of the labels of a bi-directional reference association is marked with a (T), then references in this direction are transient, i.e., not persistently stored in XML. This reduces the size of XML documents, while at the same time provides for efficient access to objects in the AADL model when operated on by tools.

Both containment and reference associations can have a multiplicity to specify the number of association instances. The containment association between ProcessImpl and

ProcessSubcomponents shown in Figure 2.1 indicates that a process implementation can contain zero or one process subcomponents subclause objects. The bi-directional reference association between ProcessTypes and ProcessImpl shown in Figure 2.1 indicates that the reference from the implementation to the type is required to be one and is stored persistently, while the reference from the type to the implementations is zero or more and is maintained only in-core.

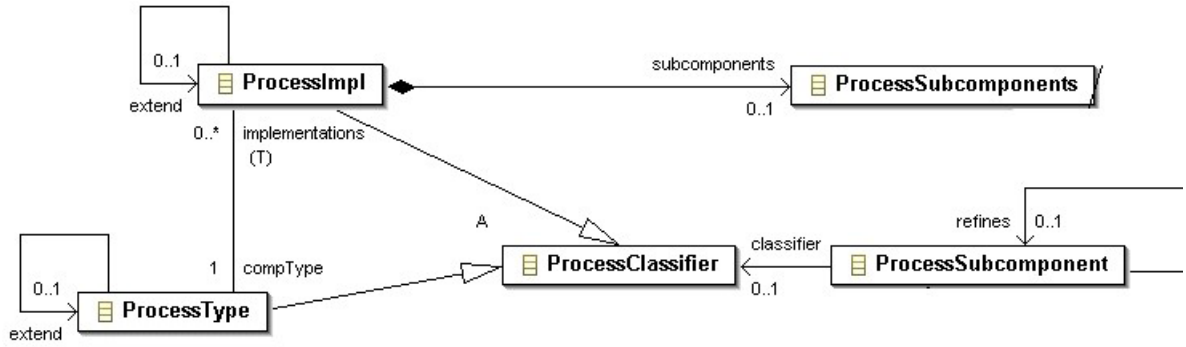

Figure 2.1 Containment and reference associations in AADL metamodel (Source: SAE-AADL Meta Model/XMI V0.999, 2006)

The AADL metamodel is the basis of a standard persistent interchange format in XML and the basis of an in-core object model that result from loading an XML-based model into memory. A detailed introduction about AADL metamodel can be found in SAE-AADL Meta Model/XMI V0.999 (2006).

2.1.2.4 AADL modelling concepts

The Architecture Analysis and Design Language (AADL) is an SAE standard for the specification and analysis of the software and hardware architecture of embedded systems. AADL uses a component-based paradigm and provides a number of modelling concepts, in a number of dimensions, which can be used for both the design and analysis of embedded systems.

Components form the central modelling vocabulary in AADL (Feiler et al., 2006; Joshi et al., 2007). There are a number of categories of component designed to model both hardware and software. Components are defined through the type and implementation declarations. A component type defines the functional interface e.g. ports of the component. A component implementation is an instance of a component type, which specifies the internal structure of the component. A component type can have several implementations but an implementation can only belong to one component type.

Composite components allow models to be structured into systems of subsystems. A system component may model a component containing hardware, software and other composite

components. AADL constructs are suitable for modelling both software and hardware components. Systems tend to be at the highest levels of the hierarchically structured models.

AADL defines five categories of application software components (i.e. process, thread, thread group, subprogram, and data) and four categories of execution platform components (i.e. processor, memory, device, and bus category). Each of these categories has its specific semantics defined in the SAE AADL standard (SAE-AS5506, 2006).

Connections are used to define interactions between components through externally visible features. A port in a component is one kind of AADL feature, which defines an interface for the directional transfer of data or events, into or out of a component. Port to port connections are pathways for such directional transfers between components.

Component properties are used to describe component types, component implementations, connections, and etc. A property contains a name, a type, and an associated value. The property type defines the set of acceptable values for a property. A property type could be one of the built-in data types e.g. string, integer, Boolean, enumeration, and component reference, or a user defined type.

For dependability analysis, one must consider the environment in which the system runs. In the context of this research, a system normally contains software, hardware and mechanical components. Model-based development tools and techniques can be used to model these components. By modelling these software and hardware components, one can create a model of the nominal (non-failure) system behaviour. Furthermore, for dependability analysis, it also requires knowledge of system fault behaviour e.g. different faults that can occur and various system component malfunctions. Thus, model-based development approaches need to incorporate system fault behaviours (models) into the development process. Model-based dependability analysis as described in Joshi et al. (2006) is an approach in which non-failure system behaviour obtained in model-based development approaches is augmented with the fault behaviour of the system.

The system nominal model augmented with fault behaviours supplies a dependability-oriented view of the system and is called the system dependability model. Given system dependability models, the dependability analysis process as described in Joshi et al. (2006) consists of defining a set of dependability requirements for the system. Then using dependability analysis techniques e.g., failure modes and effects analysis (FMEA) (IMCA, 2002), Markov chains (Trivedi, 2001; Mahmud, 2012), fault trees (Vesely et al., 2002) and reliability block diagrams

(Vesely et al., 2002; Rouvroye and Bliek, 2002), to determine whether the dependability requirements are satisfied based on the proposed system architecture.

AADL is designed to be flexible and can be extended to accommodate descriptions that the core language does not completely support. By using the extension capabilities of the language, additional models and properties can be included. In particular, an AADL Error Model Annex (SAE-AS5506/1, 2006) can be used to define components' error models, which facilitates a fault tree or Markov analysis of the system dependability.

2.1.2.5 Dependability modelling with the AADL Error Model Annex

AADL error models are specified in the AADL Error Model Annex (SAE-AS5506/1, 2006). The AADL Error Model Annex defines a sub-language of AADL that supports specification of dependability-related information e.g. fault assumptions, fault tolerance policies, error state and propagations, and stochastic parameters specifying the occurrence of fault events and propagations.

Error models, as introduced in Feiler and Rugina (2007), are based on the concept of a state machine. The error model for a component describes the behaviour of that component in the presence of local (internal) failure and repair events, and in the presence of output failures from other components that are propagated to the component's inputs. Error models may be placed in an error model annex library to be reused.

There are two kinds of reusable error model: basic and derived error models. A basic error model defines a set of error states and transitions for a component or connection. The error state transitions specify how a component changes its error state (from one state to another) when local or internal faults (represented by error events) occur or when errors propagate from other components. In a derived error model, a component's error state can be defined by using its subcomponents' error states. In this case, the error state transitions are not explicitly defined. For example, a component, because it has no internal redundancy, could be in an error state when any of its subcomponents are in an error state. Typically, the error models of the high level subsystems and overall system are used to obtain the results of system hazard analysis.

The error models for low-level components are typically used to obtain the results of failure modes and effects analysis. In the AADL Error Model Annex, error propagation rules are predefined to specify the paths through which the error propagations can propagate from one component or connection to another. For example, a process hosted on a processor can receive an error propagated from that processor. These error propagations must propagate through the data flow's direction e.g. direct port to port connections or explicit bindings as specified in the

model. They cannot propagate if the components are not connected to each other. A connection connects an output port of one component to an input port of another component. An input port of a component may have a guard_in property which can be used to:

1. Unconditionally map an incoming error state or error propagation (name) from a sender component error model into an error event name defined in the receiving error model.

2. Conditionally map a set of incoming error states and error propagations defined in sender components error models into a single input error propagation or a set of input error propagations defined in the receiving error model.

3. Conditionally mask incoming error states and error propagations.

Similarly, an output port may have a guard_out property to specify the pass-through mappings or masking of incoming error states or error propagations from sender components to output error propagations of the given component. More details of how AADL error models are used to specify and model the failure behaviour of a dependability-critical system can be found in Feiler et al. (2006), Feiler and Rugina (2007), Rugina et al. (2007), Joshi et al. (2007), Joshi and Heimdahl (2007) and Dehlinger and Dugan (2008).

Model-based engineering languages are ideally suited for high dependability embedded systems development since they supply a systematic modelling mechanism to model not only the system nominal behaviour but also the system fault behaviour. However, the use of new languages such as AADL threatens to isolate existing tools which use different languages. This is a particular problem when these tools provide an important development or analysis function. System optimisation (based on e.g. dependability and cost) is such a function. An important question faces engineering is how to do dependability analysis and system optimisation for AADL models. The following sections first introduce methods and tools for dependability analysis. It then introduces the background of system optimisation based on dependability and cost.

2.2 Methods and tools for model-based dependability analysis

For the design of embedded systems, quality requirements such as dependability and performance are important. It is beneficial to ensure that designs fulfil these quality requirements from the early architecture design stages. The later in the development a change is made the larger are the costs and delays. Model-based analysis and verification technologies can answer important questions related to these quality requirements and thus can facilitate a model-

based development process. This section describes methods and tools used for achieving system dependability analysis.

2.2.1 Methods for achieving system dependability

The development of dependable system requires techniques that can successfully detect, prevent, forecast and reduce faults. Avizienis et al. (2001) introduced four methods that can be applied to achieve system dependability:

(1) Fault prevention: to avoid the occurrences of faults,

(2) Fault tolerance: to delivery correct service regardless the presence of faults,

(3) Fault removal: to detect and eliminate faults,

(4) Fault forecast: to estimate the presence and the likely consequence of faults.

Model-based dependability analysis is one of the two means for fault forecasting (Rugina, 2007). The dependability evaluation as described in (Avizienis et al., 2001) can be:

(1) Ordinal or qualitative evaluation, which is used to identify and rank failures or the event combinations in order to avoid failures.

(2) Probabilistic or quantitative evaluation, which is used to evaluate dependability attributes in terms of probabilities.

Dependability is typically evaluated using dedicated analytical models. Some types of models are only suitable for one of the two types of analysis e.g., failure modes and effects analysis (FMEA) (IMCA, 2002) for qualitative analysis and Markov chains (Trivedi, 2001; Mahmud, 2012) for quantitative analysis. Some types of models e.g., fault trees (Vesely et al., 2002) and reliability block diagrams (Vesely et al., 2002; Rouvroye and Bliek, 2002) are suitable for both types of analysis.

During the last two decades, a considerable body of work (Rouvroye and Bliek, 2002; Papadopoulos and Maruhn, 2001; Papadopoulos et al., 2004; Bieber et al., 2002; Bozzano and Villafiorita, 2003; Giese et al., 2004; Joshi and Heimdahl, 2005; Grunske and Kaiser, 2005; Joshi et al., 2005; Joshi et al., 2006) has been developed to ensure that dependability concerns are satisfied. A large body of this work, classed broadly as dependability analysis (Vesely et al., 2002; Joshi et al., 2006), is concerned with understanding system failures and their causes and

the relationship between them. It is then concerned with reducing the probability of failures by modifying the system architecture (design) to make them less likely.

Basically, these techniques as introduced in Rouvroye and Bliek (2002), can be divided into two categories: one is quantitative techniques such as reliability block diagram (RBD), Markov analysis, hybrid techniques and etc. The other is qualitative techniques such as fault tree analysis (FTA) and failure modes and effects analysis (FMEA). Quantitative techniques are probabilistic analysis approach which aim to predict the probability of hazardous failure of the system from statistical data about the failure rate of its components. Qualitative techniques use logical analysis in which causes and effects of unwanted system behaviour are investigated.

Papadopoulos and Maruhn (2001) and Papadopoulos et al. (2004) provide a model-based synthesis of fault trees and failure modes and effects analysis. Bieber et al. (2002) provide a formal verification technique focusing on automated dependability analysis of systems represented as state automata. In this approach, model-checking (Bozzano and Villafiorita, 2003) is used to verify the satisfaction of dependability requirements or detect violations of requirements in normal or faulty conditions. Giese et al. (2004) present an approach supporting the compositional hazard analysis of UML models. The approach permits to systematically identify and prioritise the hazards and failures. Joshi and Heimdahl (2005) present a model-based safety analysis approach for automating the safety analysis process using executable Simulink models. Similarly, Grunske and Kaiser (2005) provide a technique for generating an analyzable failure propagation model for a system by annotating components with modular failure mode assumptions. The modular failure mode is described in the failure propagation transformation notation (FPTN). By using this technique a model-based safety evaluation is possible. A model-based safety evaluation enables the automatic generation of safety-related models from system models. Joshi et al. (2005, 2006) present a proposal for model-based safety analysis.

Nicol et al. (2004) is a survey of existing model-based techniques for evaluating system dependability and Mahmud (2012) provides an introduction and classification in terms of static and dynamic approaches to safety analysis. For this research, the fault tree and FMEA based dependability analysis was selected as a typical example of dependability analysis that would be applied to a system. HiP-HOPS is a fault tree and FMEA based dependability analysis and optimisation technique and hence HiP-HOPS has been selected as a suitable dependability analysis and optimisation technique to demonstrate the transformation approach. The following sections introduce some concepts from FMEA and fault tree.

2.2.1.1 Failure mode and effects analysis

The failure mode and effects analysis (FMEA), as defined in (IMCA, 2002), is "A systematic analysis of the systems to whatever level of detail is required to demonstrate that no single failure will cause an undesired event". FMEA starts from the known causes and uses the bottom-up analysis method in order to investigate the immediate failure mode and its hazards. FMEA shows the relationship between component failures and its possible effects that can have on the system. Thus, it enables the designers to understand how a failure (and how likely that failure is) for a component affects the system. The objectives of FMEA summarised in (IMCA, 2002) are to:

(1) identify each of potential failure mode and its causes,

(2) evaluate the system effects based on each failure mode,

(3) identify measures to eliminate or reduce the risks related to each failure mode,

(4) provide information to the designers to ensure they understand the limitations of the system.

FMEA as described in Mahmud (2011) can be useful in many ways: first it serves as a basis for probabilistic reliability and availability analysis. Second, it provides future references for design changes so as to avoid or minimise the effects of the most critical failures identified. Third, it helps to show (e.g., in HiP-HOPS) how some design alternatives can represent optimal trade-offs based on maximum dependability and minimum cost.

2.2.1.2 Fault tree and fault tree analysis

A fault tree (FT), as described in Vesely et al. (2002), "is constructed as a logical expression of the events and their relationships that are necessary and sufficient to result in the undesired event, known as top event". The symbols used in a fault tree indicate the type of events and the type of relationships that are involved. The fault tree can be considered as qualitative, when it provides extremely useful information on the causes of the undesired event, and also quantitative, when it provides useful information on the probability of the top event occurring and the severity of all the causes and events modelled in the fault tree.

Fault tree analysis (FTA), as described in Vesely et al. (2002), is a failure-based deductive approach. Fault tree analysis begins with a system failure (e.g. an unwanted event), and then deduces its causes by using a systematic, backward-stepping process. Fault tree analysis supplies key information that can be used to prioritise the importance of the causes to the unwanted event and those causes should be the focus of any dependability activity. As a result,

fault tree analysis is very useful to identify weaknesses of the systems and to evaluate possible upgrades.

One important concept in fault tree analysis is cut sets and minimal cut sets (MCSs). A cut set is a set of basic events so that if all these events occur together then the top event will occur. A minimal cut set is a cut set so that if any of the basic events is removed then the remaining set does not cause the top event (i.e., no longer a cut set). Minimal cut sets are important because it relates the top event directly to the basic event causes (Vesely et al., 2002). The minimal cut sets indicate all the ways that the occurrence of the various basic events can cause the top event.

Another important concept is the order of a cut set, which is the number of events that compose the cut set. Cut sets with order 1 indicate a single point of failure that will immediately cause the top event. These single failures are often weak links and could be the failures of critical components or unhandled deviations of critical system inputs. By identifying the critical failures and components the designers then can focus the design on those critical components to meet the reliability concerns. Especially, those appearing in minimal cut set of low order and those included in most cut sets are good candidate to be critical for the system dependability analysis.

Till now, this section introduces different methods for dependability analysis. For large systems, tools are needed to automatically generate and synthesise fault trees. There are also tools for these different methods for dependability analysis. The Failure Propagation and Transformation Notation (FPTN), as described in (Fenelon and McDermid, 1993; Fenelon et al., 1994) is a graphical dependability modelling tool for the analysis of failure behaviour of a system. It is an abstraction of a technique for supporting both Fault Tree Analysis and Failure Modes Effects Analysis. This technique, however, has a key deficiency: The failure model is sensitive to changes of system components and as a result the defined failure model can be easily impacted by the changes of the system model (Parker, 2010; Wallace, 2005). This can result in a complete reanalysis of the entire system when changes are required to a component in the system model. Furthermore, FPTN is limited to a process that cannot be automated (Parker, 2010).

The Fault Propagation and Transformation Calculus (FPTC) is a technique developed by Wallace (2005) in an attempt to overcome the identified limitations with FPTN. This is achieved by integrating the failure behaviour of the system more closely into the system model. The result of this, as argued in Parker (2010), is that not only transmit failure but also all potentially important dependencies are identified and recorded. Compared with FPTN, FPTC can be automated (Parker, 2010). Moreover, since the failure model is tied to system architectural model, changes made to components can be localised and not require a complete reanalysis to update these changes. However, the FPTC as argued in Parker (2010), has

limitations in the analysis of the effect of a failure, i.e. the failure must be injected into the system. This injection process has to be repeated for each different failure.

The State Event Fault Trees (SEFTs) as described in (Kaiser et al., 2007), were developed to overcome of the inability of standard fault trees to model temporal event ordering. This is done by separating states (that last a period of time) from events (that are instantaneous and typically trigger state changes). The use of the concepts of states and transitions in the state event fault trees makes it unable to use standard fault tree analysis algorithms. Instead, the model is converted into a Petri Net model. Once a Petri Net model exists, a quantitative analysis may be performed automatically by a suitable tool, such as COMPASS (COMPASS project, 2013). Since the SEFTs is a state-based analysis technique, this could lead to a state-space explosion for larger models and thus as argued in (Parker, 2010), may affect the scalability of the technique.

More detailed introduction of various dependability analysis techniques and tools can be found in Paker (2010) and Mahmud (2012). However, these tools only support system dependability analysis and none of them can supply system architecture optimisation based on dependability and cost. HiP-HOPS is a model based dependability analysis and architecture optimisation technique. Recently, HiP-HOPS has combined with meta-heuristics (Pareto-based Genetic Algorithms (Walker et al., 2013)) to enable the design models to meet the desired cost and dependability requirements. By using genetic algorithms, HiP-HOPS is able to explore the space of variations of a model and by evaluating the dependability and cost of the various model variations, HiP-HOPS is able to solve multi-objective (cost and dependability) optimisation problems. Dependability analysis and optimisation tools that are not AADL based cannot be used directly on AADL models. Section 2.3 summarises techniques and associated tools for system architecture optimisation. By comparing the advantages and limitations of these related work, this section also draws forth the research problem in the context of this research, i.e., how best to integrate system architecture optimisation analysis into AADL. A model transformation between AADL and HiP-HOPS has been identified as a cost effective way to tackle this problem. Section 2.4.3 summarises the current model transformation methods used in the context of dependability analysis and also gives the motivation for choosing HiP-HOPS as a target model for AADL dependability and optimisation analysis. Details of the HiP-HOPS modelling concept for dependability analysis will be introduced in the following section.

2.2.2 Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) – A tool for system dependability analysis and optimisation

HiP-HOPS is a well-known fault tree based dependability analysis technique for system dependability modelling and analysis. HiP-HOPS is a state-of-the-art system dependability (i.e. availability, maintainability, reliability and safety) analysis technique. HiP-HOPS uses the topology of a system together with reusable local failure specifications at component level to automatically produce a network of interconnected fault trees and a Failure Modes and Effects Analysis (FMEA) for the system. It offers a significant degree of automation and reuse thereby countering problems arising from the increasing complexity of systems. The technique is supported by a software tool.

A HiP-HOPS model abstracts a system into components and connections between those components. The input and output ports of components provide the anchor points for connections between components. HiP-HOPS defines a language for the description of failure behaviour at the component level. In the basic version of this language, the failure behaviour of a component can be specified as a function of internal failure modes of that component (internal malfunctions) and deviations or omission of parameters as they can be observed at the outputs of connected components (output malfunctions). Each internal malfunction is optionally accompanied by quantitative data, for example a failure rate or a repair rate. Output malfunctions are associated with Boolean expressions which describe their causes as a logical combination of internal malfunctions of the component and the malfunctions of connected components manifest as deviations and omission of parameters at the component inputs. It is this component failure data that is used by HiP-HOPS to automatically produce a network of interconnected fault trees and a Failure Modes and Effects Analysis (FMEA) for the system. To synthesise a system fault tree, HiP-HOPS examines the local fault tree expression of each component and the propagation of failure data between components. In HiP-HOPS, the Line objects specify how failure events propagate between components.

2.2.2.1 HiP-HOPS metamodel

The metamodel of HiP-HOPS describes the structure of the model. Figure 2.2 shows the simplified HiP-HOPS metamodel. The top element of system consists of components and lines. Each component contains ports, a default implementation, and alternative implementations (alternatives). Each implementation can have failure data (see fData shown in Figure 2.2) to indicate how this component can fail and the probability between component failures and system failures. The failure data consists of basic events and output deviations. Each basic event contains a name and an optional unavailability formula that defines the quantitative failure

information of this basic event. The basic events are the basic failure modes of an individual component. Basic events can include things like wear, environmental factors (e.g. temperature), faults in the component (e.g. overheating, short circuits), or anything else that cannot be decomposed further.

The output deviations represent errors or faults propagated from the outputs of a component. At minimum, an output deviation needs a name and a failure expression (see defaultString attribute, defined as an Estring type under object OutputDeviation shown in Figure 2.2). The failure expression is a string representing the logic of the output deviation. This is an expression in Boolean logic that specifies how causes – either basic events, common cause failures, input deviations, or deviations imported via allocation links – to the fault propagated from the output. It may also possess a flag indicating whether this is a "system output port" (see systemOutPort attribute). The system output port represents system level failures. Output deviations with this flag set to true become the top events of fault trees.

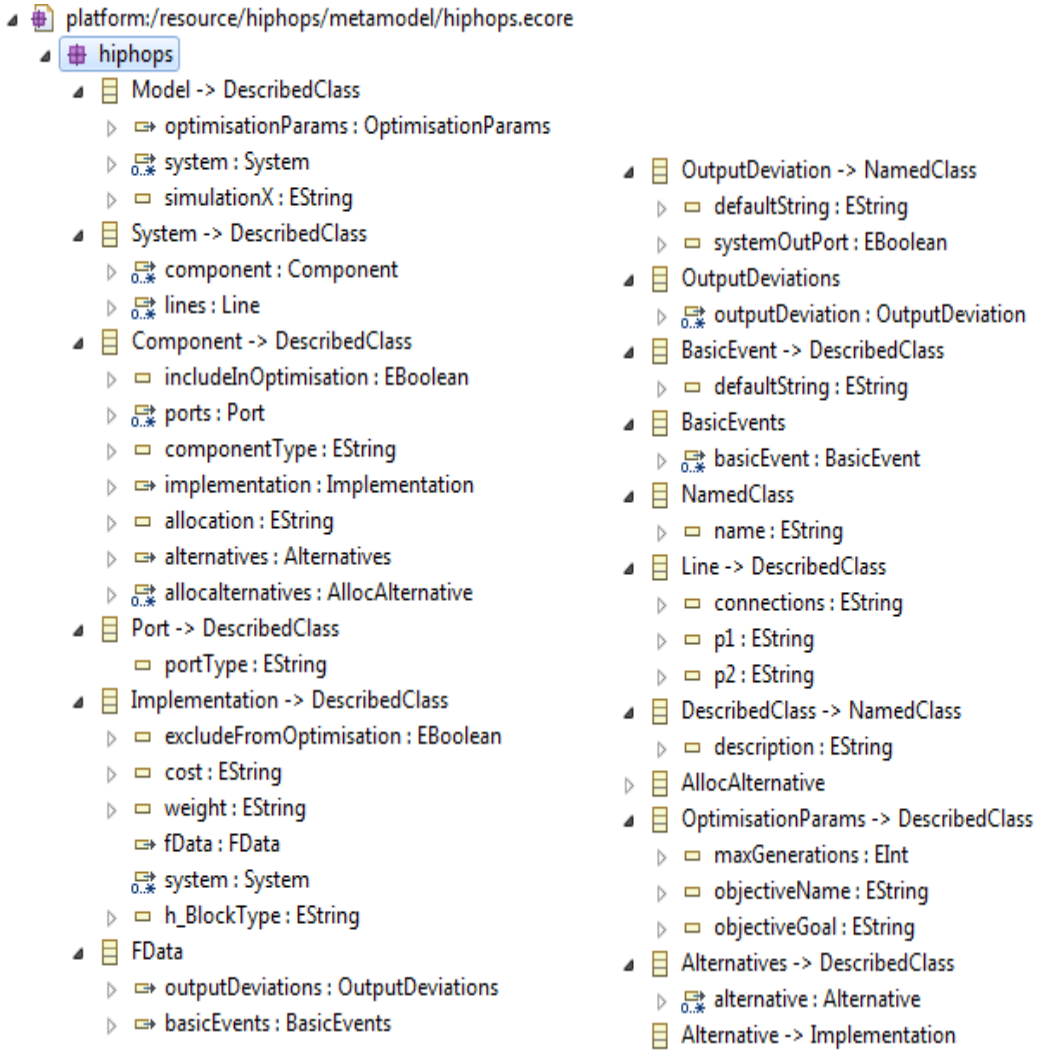

Figure 2.2 The HiP-HOPS metamodel

2.3 System optimisation with dependability

For the development of dependable systems, designers must address both dependability and cost concerns. For example, the cost of cars can be reduced by developing distributed flexible subsystems such as car control systems that can be based on reusable and standard interchangeable modules and architectures (Papadopoulos and Grante, 2005). Such control systems, although of a lower cost, are required to perform reliably.

The designers must carefully address the safety issues since the car control subsystems provide important vehicle functions such as steering and braking. Development processes are needed to address both these safety and economic concerns and they should be applicable at an early stage of the design process so as to avoid unnecessary and expensive design iterations.

However, to consider both dependability and cost design objectives, the following two situations need to be addressed (Adachi et al., 2011; Walker et al., 2013). On one hand, a large number of different designs with different architectures can potentially meet the dependable requirement both technically and economically. In such architectures, shared information and hardware resources allow a large number of configurations options. This results in more difficulties in design since potential design spaces are largely expanded. The system designers must find the architecture that needs minimal development costs. On the other hand, if no design solutions can fulfil all dependability-related requirements, then the designer must find the architectures that achieve the key requirements with best possible trade-offs between dependability and cost. This is called the trade-off between multi-objectives.

Aleti et al. (2009) and Walker et al. (2013) argue that it is the conflicting quality requirements (e.g., minimum cost and maximum dependability) and the increasing complexity of today's systems, which adds more challenges for the development of dependable systems. With a number of different designs with different architectures that can potentially supply the functions of a system, the designers then face a difficult multi-objective optimisation problems (Walker et al., 2013). Multi-objective optimisation problems, as described in Adachi et al. (2011) and Walker et al. (2013), can effectively be addressed by the use of optimisation techniques and computerised algorithms. These algorithms can efficiently search for optimal solutions in large potential design spaces. Grunske et al. (2007) is an introduction to the model-based optimisation field and Aleti et al. (2012) is a wider survey of literature on architectural optimisation techniques.

As argued in Walker et al. (2013), to find a suitable or optimal architecture design is difficult and some automation is needed. One key issue facing system designers is how to optimise

system architectures throughout the whole system development lifecycle with respect to dependability and cost. Modelling languages and emerging ADLs offer multiple qualities analysis and evaluation functions. Thus, they could benefit from concepts and technical support of ADLs that enable this type of optimisation.

Some work has been done for solving multi-objective architecture optimisation problems in dependable systems. This includes the work that based on Reliability Block Diagrams (RBDs) model (Konak et al., 2006) and the HiP-HOPS (Papadopoulos and Grante, 2005, Hamann et al., 2008, Adachi et al., 2011). HiP-HOPS is a model based dependability analysis and architecture optimisation technique. Recently, HiP-HOPS has combined with meta-heuristics (Pareto-based Genetic Algorithms (Walker et al., 2013)) to enable the design models to meet the desired cost and dependability requirements. By using genetic algorithms, HiP-HOPS is able to explore the space of variations of a model and by evaluating the dependability and cost of the various model variations, HiP-HOPS is able to solve multi-objective (cost and dependability) optimisation problems.

Some related work has been done in the context of ADLs. Walker et al. (2013) presented a multi-objective optimisation approach based on EAST-ADL. In this approach, three objectives, i.e., dependability, timing and cost were evaluated. Typically, the system dependability is evaluated by HiP-HOPS where the system dependability characteristics are obtained via transforming EAST-ADL model to HiP-HOPS model. One important aspect of this approach is that they used EAST-ADL's variability management mechanism to define the alternative implementations and thus benefiting to the ease of defining design space. AADL, however, has no such scheme to define the search space of alternatives. A scheme for representing component variability is needed for optimisation AADL models. This motivates the thesis to extend the AADL model by defining sets of optimisation related properties to represent variability. The details of defined optimisation properties are given in Chapter 5.

Some other work has been done on the development of tools for multi-objective optimisation of software architectures. One tool is ArcheOpterix (Aleti et al., 2009), which is based on AADL and potentially allows automatic optimisation of AADL specifications. Two quality metrics, i.e., data transmission reliability and communication overhead were evaluated. The tool was extended to enable reliability, cost and response time optimisation of AADL models shown in Meedeniya et al. (2009). In this extension, only simple component redundancy allocation was used as a reliability improvement. The design space of a set of component alternatives, however, was not considered, which is considered important to create the possible design space in this research for optimisation. Furthermore, in order to perform the optimisation, it is necessary to transform from other models to AADL model since the tool is tightly integrated in AADL.

Another tool is AQOSA (Automated Quality-driven Optimisation of Software Architecture) (Li et al., 2011, Etemaadi and Chaudron, 2012), for automated software architecture optimisation that allows multiple quality attributes (processor utilisation, response time, data flow latency, safety and cost). The tool uses model transformation technology to transform AADL input models into an intermediate AQOSA-IR model and this model is then used as the basis of the optimisation process. AQOSA is developed independently for any domain specific languages and hence needs model transformation technology to generate analysis models from other architecture models to perform the optimisation. There is, however, no detailed work shown how the variability of alternative AADL components are represented and how the AADL dependable model can be transformed to AQOSA for AADL architecture optimisation based on dependability and cost.

There is still a lack, however, of analysis techniques and tools that can perform a dependability analysis and optimisation of AADL models. HiP-HOPS models the topology of a system in terms of components and data transactions among those components. HiP-HOPS supplies a fast algorithm for automatic generation of dependability analysis artifacts such as fault trees and Failure Models and Effects Analyses (FMEAs). Moreover, using genetic algorithms, a novel HiP-HOPS extension is able to explore the space of variations of a model. By evaluating the dependability and cost of the various model variations, HiP-HOPS is able to solve difficult multi-objective (cost and dependability) optimisation problems.

A cost effective way of adding system dependability analysis and optimisation to models expressed in AADL is to exploit the capability of HiP-HOPS. This can be done by transforming the AADL model into an equivalent HiP-HOPS model. The purpose of this research is to develop a method which combines the multi-objective optimisation techniques (e.g. HiP-HOPS) and variability capabilities (to be developed in this thesis) and quality attribute such as dependability transformed from AADL. The benefit of this transformation is that it opens a path that will enable the AADL language to take advantage of some of the unique capabilities of HiP-HOPS, i.e. the synthesis of multiple failure mode FMEAs, temporal fault tree analysis and evolutionary architecture optimisation with respect to dependability and cost. This in turn brings some of the benefits of this type of analysis – rationalisation, automation, consistency between design models and analyses and the ability to iterate analyses - in the context of an emerging paradigm for model-based design.

To enable the design of model transformation from one model (e.g. AADL model) to another (e.g., HiP-HOPS), it is necessary to understand the concepts used in each of the model and identify the mappings between the two models. The rest of this Chapter gives an introduction of model transformation and developed model transformation techniques to tackle the identified

limitations and challenges, i.e., lack of techniques for system optimisation of AADL dependable systems, discussed in this thesis.

2.4 Model transformation

Different models and modelling languages are used for different kinds of development and analysis and to describe a system on different abstraction levels and from different views. It is not always possible or best to develop and analyse systems in a single model. Different models are implemented in different languages and hence have different domains. These include data program code, data schemas, UML models and interface specifications etc. (Czarnecki and Helsen, 2006).

Due to the use of varied models, transformations between models are necessary (Czarnecki and Helsen, 2006; Biehl et al., 2010). Czarnecki and Helsen (2006) argue that model transformations are very important in model-based engineering and they are necessary in application including: generation of lower-level models code from high-level models, synchronisation and manipulation of models at different (or same) levels of abstraction, the creation of a system based on query views, model refactoring in model revolution tasks, and etc. Biehl et al. (2010) also argue that the transformation between the automotive domain to the safety domain is required to feed the growing demand of integrating safety analysis techniques into the model-based embedded systems development process. They also argue that the integration (through model transformation) help the safety engineers to perform the safety analysis early in the development process to fulfil safety goals with lower effort and cost.

In the context of this research, AADL supplies a dependability-oriented view of system by providing support for capturing all the dependability-related information through AADL architecture modelling and AADL Error Model Annex. To aid automated dependability analysis (e.g. automated generating fault trees and FMEAs) for AADL, the AADL dependability model need to be transformed to other model-based engineering approaches such as HiP-HOPS.

2.4.1 Introduction to model and metamodel

A model as described in Jouault et al. (2008) is a representation of a system, which contains characteristics and knowledge of the system. In model-based engineering, models must be described in accurate modelling languages. When the conceptual foundation of a modelling language is expressed as a model, then it is called metamodel (Jouault et al., 2008). For example, the metamodel and interchange formats annex (SAE-AADL Meta Model/XMI V0.999, 2006) specifies the AADL metamodel, i.e., the structure of AADL models. This is done through

a set of related class specifications using the Eclipse Modeling Framework (EMF) Ecore (Steinberg et al., 2009) notation.

The AADL metamodel is useful since it can be manipulated programmatically through an API, and can also be stored in a standard interchange format e.g. in XML. This enables different tools to interoperate on AADL models if they support AADL XMI (XMI, 2003) metamodel specification or XML schema (XML, 2001).

The AADL metamodel is the basis of a standard persistent interchange format in XML and the basis of an in-core object model that results from loading an XML-based model into memory. A detailed introduction about the AADL metamodel can be found in SAE-AADL Meta Model/XMI V0.999 (2006).

A model usually conforms to its metamodel (the relation between a model expressed in a language and the metamodel of this language is called conforms to). Similarly, metamodels are in turn expressed in a modelling language specified in a metametamodel. A detailed definition of these concepts can be seen in Jouault et al. (2008).

2.4.2 Introduction to model transformation

Mens and Van Gorp (2006) define model transformation as the "automatic generation of one or multiple target models from one or multiple source models, according to a transformation description". A model transformation definition, as introduced in (Biehl, 2010), expresses how source models are transformed into target models.

The transformation should also preserve some properties. The semantics between source model and target model should be unchanged during the transformation (Mens and Van Gorp, 2006; Biehl, 2010). This is called semantics-preserving program transformation (Yang et al., 1992), which is defined as the values computed are unchanging while the way computations are performed is changing. Although, a transformation between source and target models is semantics preserving, the target model often makes explicit properties that are implicit in the source model. An example of this is the HiP-HOPS Line object which models the propagation of failure events. Failure propagation information is available in the AADL source model but not so explicitly represented. If source metamodels assumptions are basically different from target metamodels, it might be difficult to preserve all the semantics (Biehl, 2010).

Transformations may be defined in a model transformation language. If the model transformation language is rule-based (general standalone rule with source and target) then the transformation definition is a set of model transformation rules (Biehl, 2010). Each rule defines

a source pattern and a target pattern and specifies how an element in the source pattern can be converted into an element in the target pattern. The running result of each rule is that a target pattern (specified in the target model) is created for any source pattern appears in the source model.

Model transformations as introduced in (Biehl, 2010), enable an automatic generation of target models from required information obtained from the source models. This automation (through model transformation) allows consistency between different models and enables systematic reuse of information.

Generally, the goal of the model transformation is to generate a target model e.g. `Mb`, from a source model e.g. `Ma` through a model transformation model definition defined as a model e.g. `Mt`. These models (i.e. `Ma`, `Mb` and `Mt`) have to conform to their own metamodel respectively.

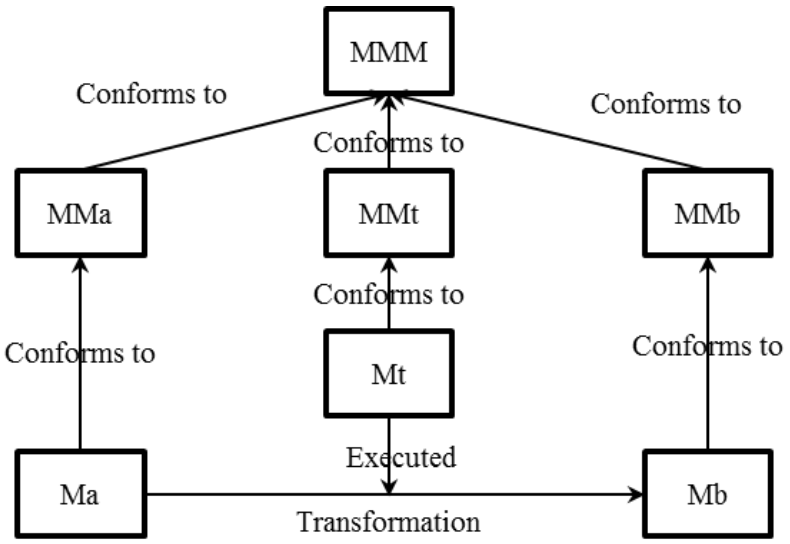

Figure 2.3 An overview of model transformation process

Figure 2.3 shows the complete model transformation pattern. A target model `Mb` that conforms to a metamodel `MMb` is created from a source model `Ma` that conforms to a metamodel `MMa` through the execution of a model transformation model `Mt that` conforms to a model transformation metamodel `MMt`. These metamodels conform to a metametamodel `MMM`, an example of which is Ecore (Steinberg et al., 2009). The execution of `Mt` results in automatic generation of `Mb` from `Ma`.

In the context of transforming AADL to HiP-HOPS, the AADL dependable model (`Ma` in Figure 2.3) conforming to AADL metamodel (`MMa` in Figure 2.3) is the source model. The HiP-HOPS file (`Mb` in Figure 2.3) is the final output of the transformation, which conforms to the HiP-

HOPS schema definition (MMb in Figure 2.3). The transformation rule (sematic mapping, Mt in Figure 2.3) written in the ATL language is a model conforming to the ATL metamodel (MMt in Figure 2.3). All metamodels conforms to the Ecore (Steinberg et al., 2009). More details of AADL Ecore metamodels and HiP-HOPS Ecore metamodel can be found in SAE (2006) and Biehl et al. (2010).

2.4.3 Model transformation for model-based dependability analysis

A number of model transformation approaches have been proposed for the dependability analysis of AADL models. One approach consists of developing a framework for dependability analysis of AADL error models by transforming AADL models to Generalised Stochastic Petri Nets (GSPNs) as described in (Rugina, 2007; Rugina et al., 2008). The GSPN obtained by transforming the AADL dependability model is constructed by applying a set of transformation rules. All transformation rules are presented and formalised using the notations related to Petri Nets. There are two subnets, the component subnets and the dependency subnets are structured in order to obtain GSPN model from AADL model. The component subnets model the behaviour of independent components based on their own faults and repair events. Components' error models including error states and error transitions triggered by error events are processed to create the component subnets. The corresponding transformation rules are identified for transforming independent components. For example, the AADL error state is transformed to the place in GSPN. The AADL initial error state is transformed to the place with token in GSPN. The AADL error event is transformed to GSPN transition. The dependency subnets model the error behaviour associated with dependencies described by name-matching, i.e. input and output error propagations. The input and output error propagations are the basic mechanisms for presenting interactions between AADL components. For each identified output error propagation, the input error propagations that could occur as effects of the output error propagation are searched through the AADL architecture model. The name-matching input and output error propagations are then transformed, according to defined rules, into dependency subnets that are connected to the component subnets.

There are also rules for transforming AADL guard_in and guard_out error properties. These error properties provide a mechanism for error propagation filtering and masking. For each of such error property, the components owning the propagations and states named in the Boolean expressions or the property are searched and obtained. The resulting dependency subnets are connected to the component subnets. Furthermore, there are similar defined rules for other mechanisms, e.g., for connecting error states to modes (Rugina, 2007), i.e., guard_event and guard_transion.

This approach focuses mainly on quantitative (probabilistic) analysis. This approach, however, as argued in (Walker et al., 2013), does not allow qualitative analysis. Qualitative analysis such as FMEA is important when probabilistic data is not available. A more suitable model for qualitative analysis is the fault tree and an alternative approach for AADL dependability analysis has been described in (Joshi et al., 2007), where the AADL dependable model is transformed to static fault trees in three steps. First, the AADL system instance error model is extracted based on the AADL system instance model and the individual error models referenced in the system instance model. The extracted instance error model is stored in the form of nodes of a Directed Graph (DG) (Joshi et al., 2007). The DG is used for identifying error propagation sources e.g. the directed edge points to all associated components that could be the sources of their input error propagations. Second, once the system instance error model data has been stored in the form of a DG, a fault tree generation algorithm is used to recursively generate an intermediate fault tree with the top event being the error state or propagation listed in an AADL `report` property. Finally the intermediate fault tree is formatted for a specific analysis tool. There is, however, no published implementation work showing how each step of the transformation is done and how the static fault tree is generated from an AADL error model.

There are also drawbacks, however, in Joshi and others' conversion, particularly, as argued in (Walker et al., 2013) that the temporal characteristics (Walker and Papadopoulos, 2009; Mahmud et al., 2010; Mahmud, 2012; Mahmud and Mian, 2013) of the AADL error model are lost through the transformation to static fault trees. As a consequence, this loss could cause serious errors when analysing dynamic systems. To overcome the above drawbacks, HiP-HOPS integrated temporal logic called Pandora (as described in Walker and Papadopoulos, 2006) to achieve automated synthesis and analysis of dynamic fault trees. This is one reason that HiP-HOPS was chosen as the dependability analysis tool in this research and also motivates the work to convert AADL models to HiP-HOPS models.

A very similar work to Joshi et al. (2007) is the work reported in Dehlinger and Dugan (2008). In this work, Joshi and others' conversion is extended to allow dynamic fault trees (Dugan et al., 2007) to be generated from an AADL model. There is, however, no detailed implementation work show how the dynamic fault tree is generated from an AADL error model.

Several groups have been reported to have an integration of dependability related analysis capabilities based on AADL. The aerospace corporation toolset, as described in Hecht et al. (2009, 2010) and Feiler (2010), is integrated with OSATE and generates Generalised Stochastic Petri Nets (GSPN) from AADL models. From this GSPN (XML file generated by the ADAPT tool (Rugina, 2007; Rugina et al., 2008), a translator generates a Stochastic Activity Network

model in the Mobius toolset format. Mobius (2014) is a model-based environment for validating reliability and availability of systems. A second translator generates an FMEA representation from the Petri Net of the error model for FMEA analysis. The COMPASS project (Correctness, Modelling and Performance of Aerospace Systems) (2013) is an integration toolchain for verification and validation of AADL models. The toolset support safety analysis such as fault tree analysis and failure mode and effects analysis (Feiler, 2010). Under ASSERT project (2014), an effort has been started to develop a transformation between AADL dependability model and AltaRica (Bieber et al., 2002). The AltaRica project (2014) is a modelling language designed by LABRI (2014) and industry companies for formally specify the behaviour of systems when faults occur.

Biehl and others (2010) reported a model transformation method for automated transformation from EAST-ADL to HiP-HOPS. They assumed a model based development process through which the automotive concepts (e.g. vehicle features, functions, requirements, variability, software and hardware components and communication) are represented by the EAST-ADL architecture description language. The concepts of the error analysis are represented by the safety analysis tool HiP-HOPS, and the automated transformation from EAST-ADL to HiP-HOPS integrates the development and analysis domains together. Their transformation method is divided into two phases. The first phase is designed for conceptual mapping between the two domains, i.e., an abstract model to model transformation from source model (EAST-ADL) to intermediate target model (intermediate HiP-HOPS model). The purpose of the first phase transformation is to preserve the semantics of the source model through mapping concepts between EAST-ADL and HiP-HOPS. The second phase is designed for representing the output of first phase transformation (the intermediate HiP-HOPS model) in the concrete syntax of target model (HiP-HOPS model). In this work, only concept mapping between EAST-ADL and HiP-HOPS is reported. For example, the EAST-ADL `ErrorModelType.errorConnector` object is mapped to the HiP-HOPS `System.Lines` object. There is, however, no published work showing why the two identified objects should be mapped and how this mapping is transformed and implemented in the selected transformation language, ATL.

Grunske and Han (2008) compared the AADL error annex to some of the existing model-base dependability analysis techniques according to its ability in modelling and tool support. Five existing model-based evaluation methods, i.e., Failure Propagation and Transformation Notation (FPTN), as described in (Fenelon and McDermid, 1993; Fenelon et al., 1994), Component Fault Trees (CFTs), as described in (Kaiser et al., 2003), State Event Fault Trees (SEFTs) as described in (Kaiser et al., 2007), Fault Propagation and Transformation Calculus (FPTC) as described in (Wallace, 2005) and HiP-HOPS as described in (Papadopoulos and Maruhn, 2001;

Papadopoulos et al., 2004; Papadopoulos and Grante, 2005) are compared. The author highlighted that one weakness of most of these techniques is they cannot generate FMEA tables. Only HiP-HOPS provides ways to obtain FMEA tables by analysing minimal cut sets of the generated fault trees. The author also highlighted that the adoption of the generation of FMEA tables would be a good extension to the support of AADL's error model. The integration of AADL and HiP-HOPS allows AADL exploring some of the advantages of HiP-HOPS such as the ability to generate FMEA, temporal fault trees and architecture optimisation. This is also one motivation of the selection of using HiP-HOPS as the dependability analysis and optimisation technique in this research.

The above early work shows the transformation approaches for dependability analysis of ADLs models, however, there is little published work which gives a comprehensive description of a method for transforming AADL models. Although transformations from AADL into other models have been reported only one comprehensive description has been published, a transformation of AADL to Petri Net models. There is a lack of detailed guidance for the transformation of AADL models. This motivates the thesis to give a detailed transformation from AADL model to HiP-HOPS model shown in Chapter 3. Furthermore, none of the above early work shows in detail how the AADL model can be optimised based on dependability. Section 2.3 has introduced the concept of system optimisation and a summary of the attempt to optimise architecture models. This includes some tools such as ArcheOpterix, AQOSA and HiP-HOPS for multi-objective optimisation of software architectures. Based on the identified limitations in the exploitation of current system optimisation methods, Chapter 5 introduces the transformation approach advocated in this thesis so as to tackle these limitations.

2.4.4 Modularity in model transformation

Modularity in the context of a programming language means the ability to compose programs as a set of smaller units, called modules. It is common to impose a set of requirements to the modules, e.g., for software development, modularity means to achieve two qualities: reusability and adaptability.

In addition to correctness, the design of model transformation should fulfil certain quality properties, such as adaptability, modifiability and reusability. In the context of model transformation, a suitable modular structure may help designers to fulfil these quality properties. Kurtev et al. (2007) analysed several modular features of rule-based model transformation languages in order to produce adaptable and reusable transformation designs. Two quality properties, i.e., adaptability and reusability of transformation designs, are studied. This work draws developers' attention to two main considerations: First, to achieve a proper

modularisation, more than one decomposition in the source and target metamodels should be considered. Second, an integration mechanism that ensures loose coupling between modules should be used in order to achieve an adaptable and reusable transformation design.

In the context of model transformation, most of current transformation languages as introduced in (Kurtive et al., 2007; Czarnecki and Helsen, 2006), are rule-based. Rules defined in a transformation language are either matched rules (i.e. clearly separated source and target pattern) or are imperative rules (i.e. procedures units that similar to the imperative languages). The transformation rule as introduced in Kurtive et al. (2007) and Jouault and Kurtev (2005), is the basic module. Modularity as described in Kurtive et al. (2007) requires decomposition of transformation definitions (rules). This will help the transformation designers to reduce the complexity when designing model transformation and thus may help in promoting reusability and adaptability. Rules as described in Kurtive et al. (2007) should also be designed to meet the criterion for composability. This means that a rule may use the functionality of other rules to combine them together.

Generally, the declarative languages (Kurtev et al., 2007) with implicit rule calls enable more adaptable transformations. This is because implicit rule calls are not explicitly referenced to rule names. An implicit rule call is triggered when one rule needs the functionality produced by another rule. This ensures loose coupling between rules since implicit rule call relies on indirect rule dependencies. The evaluation, however, presented in Kurtev et al. (2007) does not use formal case studies.

2.4.5 Transformation languages

The model transformation language is important in model transformation. Software engineers should be supported in performing model transformation by mature model-based engineering tools and techniques (Jouault et al., 2008). Model transformation languages are designed and developed to provide such a support so as to solve common model transformation tasks (Czarnechi and Helsen, 2006). The model transformation technique e.g. the declarative approach offers a number of advantages compared to the use of a general purpose programming language such as Java. In any model transformation, two tasks must be accomplished. Firstly, the source model must be navigated to locate the elements that contain the information required in the target model. Secondly, the collected information must be processed and used to construct the appropriate elements of the target model. If a general purpose language is used to implement a model transformation, considerable programming effort can be devoted to the navigation of the source model. The use of a model transformation language abstracts away from explicit source model navigation. Instead, the transformation language programmer specifies, using a

pattern, the structure of the information required from the source model and the transformation language pattern matcher will systematically traverse the source model instantiating pattern matches wherever they are found.

Model transformation languages and engines have been classified in Czarnechi and Helsen (2003, 2006) and Biehl et al. (2010). Different model transformation languages have their advantages and disadvantages in solving specific types of tasks. To choose which model transformation approach is based on specific domain models. For example, as described in Czarnecki and Helsen (2006), most graph-transformation-based approaches are suited for transforming UML models.

For this research the model-to-model transformation languages are well suited for the semantic mapping transformation, since both input and output are models. The ATLAS Transformation Language (ATL) (Jouault et al., 2008), a hybrid language containing a mixture of declarative and imperative constructs is chosen in this research. The criteria for selecting ATL are simplicity (i.e. hiding the complexity e.g. explicit source model navigation, of model transformation), suitability (i.e. suitable to illustrate the developed transformation method) and integratability (i.e. can be easily integrated into the growing toolset surrounding AADL).

The use of a declarative language such as ATL abstracts away from explicit source model navigation. Instead, the ATL programmer, specifies, using a pattern, the structure of the information required from the source model and the ATL pattern matcher will systematically traverse the source model instantiating pattern matches. Each match of a pattern in the source model identifies the source model elements required to construct an element of the target model. Matched patterns trigger rules which specify how the information matched in the pattern should be processed to create an element of the target model. The ATL rules also simplify the processing of the information necessary to construct target model elements. The right-hand-side of an ATL rule is essentially a template for the target element to be constructed and variables in the template are instantiated when the left-hand-side pattern of the rule is matched.

In isolation, each rule is relatively straightforward. It is the ATL rule matching algorithm which “brings the rules together” to achieve a complete transformation. The higher level of abstraction of the declarative approach not only reduces the workload of creating a model transformation it also makes the transformation less rigid and less dependent on the precise structure of the source model.

In this research, the criteria for selecting a transformation language is only for illustration purpose i.e. for showing how the identified transformation method can be implemented in a

transformation language. In the case that maintainability and efficiency is a main concern for the transformation, other model transformation languages and tools e.g. Tefkat (Lawley et al., 2004) may be used.

ATL is selected in this research not only because it hides the complexity of model transformation behind the simple syntax but also because:

1. It has been shown to provide effective means of achieving similar model transformations (Biehl et al., 2010).

2. ATL is a hybrid model-to-model transformation language that uses declarative (matched) rules and imperative (called) rules (Kurtev et al., 2007), which makes the language suitable to illustrate the transformations method.

3. ATL can be integrated into the growing toolset surrounding AADL. The Open-Source AADL Tool Environment (OSATE) developed by SEI (2004) is a set of plug-ins based on Eclipse and the Eclipse Modelling Framework (EMF) (Steinberg et al., 2009). OSATE is used to parse and semantically validate textual AADL specifications into models for input to ATL transformations.

ATL transformation is unidirectional. The transformation process operates on read-only source models and generating write-only target models i.e. the source model can be navigated but cannot be changed; the target model cannot be navigated. More detailed ATL features such as modular, helper, matched rules, and etc. can be found in Jouault et al. (2008) and ATLAS group (2005).

Chapter 3 Model transformation for the automatic generation of HiP-HOPS-oriented dependability analytical models from high level AADL dependable models

This Chapter outlines a new model transformation method (AADL2HiP-HOPS) for the automatic generation of HiP-HOPS-oriented dependability analytical models from high level AADL dependable models. AADL is used as the notation for capturing the system architecture model and the AADL Error Model Annex is used to capture the component faults and failure modes. The method transforms this AADL dependability model to a HiP-HOPS model which is then used for the synthesis of fault trees, FMEAs and other analyses to automatically generate the fault tree and FMEA table of a system for further dependability analysis.

3.1 Model transformation overview

AADL and HiP-HOPS use different modelling concepts and hence have different models and different modelling languages. To perform a HiP-HOPS dependability analysis of an AADL model it is necessary to create a HiP-HOPS model from the AADL model. There are similarities and differences in the AADL and HiP-HOPS modelling concepts. Both languages use the concepts of component, port and connection although detailed semantics differ. The remaining part of this section discusses the main similarities and differences between AADL and HiP-HOPS.

AADL component error models, as described in Feiler and Rugina (2007), are effectively state machines. Figure 3.1 illustrates how error models (i.e. error state machines) relate to AADL components and how components are connected to each other through AADL connections. In Figure 3.1, each component has a number of ports and an associated error model, e.g., component `D` has one input port called `in` and one output port called `out` and an associated error model (see dotted circle shown in Figure 3.1). The error model is a state machine which describes how the state of the component changes in response to events or the state of other components as observed at input ports. An omission of input or an internal component failure is an example of an event that might cause a transition to a component-failed state.

The connections between components form the main paths of error propagation through the system. In Figure 3.1, the connections between components are shown as small circles. AADL connections can also have associated error models although this is not typical. If there is no error model associated with the connection, error propagations defined in the connected component will propagate through the connection. To describe the transformation, the thesis

first discusses the case in which there are no error models associated with connections. The case in which connections have error models is discussed at the end of this chapter.

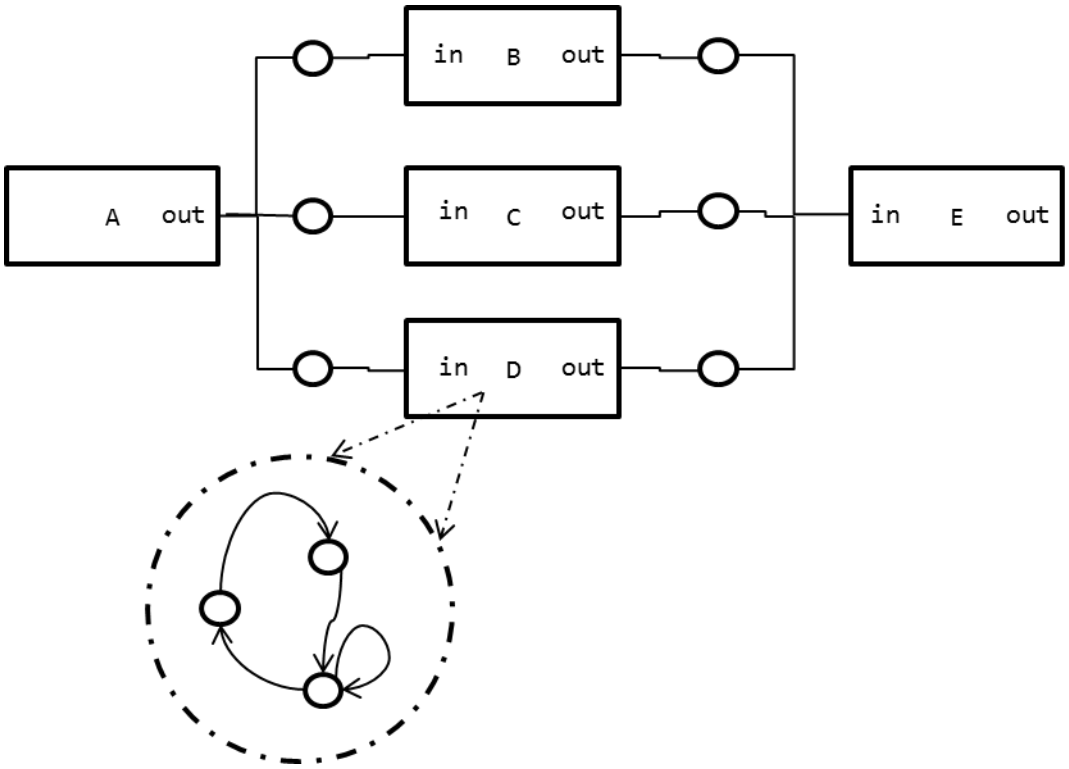

Figure 3.1 An AADL system model consisting of five components, each with an associated error model, and six connections

Figure 3.2 illustrates the HiP-HOPS model corresponding to the AADL model shown in Figure 3.1. Instead of state machines, HiP-HOPS uses local Boolean failure expressions to describe how each component may fail based on its internal malfunctions or input error deviations. The failure expression is a Boolean logic expression, which is equivalent to a local fault tree for that component. For example, in Figure 3.2, the fault tree in the dotted circle emanating from component D specifies the various ways in which component D can fail.

The HiP-HOPS Line element describes how events, typically error events, propagate from one component to another. A Line element is associated with each input port. In Figure 3.2, there are four HiP-HOPS Lines (shown as small triangles) connecting the components, one for each input port. HiP-HOPS can automatically generate a system fault tree from the locally defined component fault trees and the propagation information contained in the HiP-HOPS Line elements.

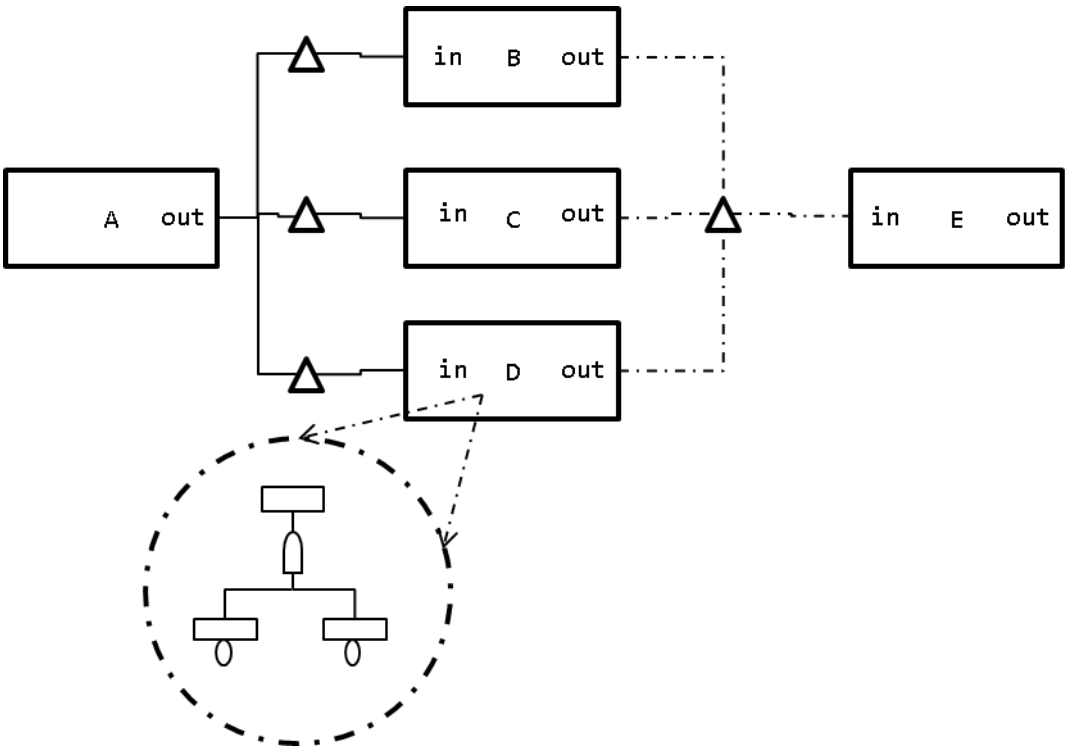

Figure 3.2 HiP-HOPS model corresponding to the AADL model shown in Figure 3.1. The HiP-HOPS model consists of five components, and their associated fault tree expressions. There are four HiP-HOPS Lines (shown as small triangles) connecting the components

3.1.1 Translation of AADL component error model to HiP-HOPS failure expressions

At the highest level of abstraction, the transformation consists of two parts. One part is concerned with the component specific error behaviour and the other part is concerned with inter-component error propagation. Structurally, the model transformation transforms AADL components into HiP-HOPS components and constructs HiP-HOPS Line objects from information in AADL components and connections. More specifically, for a given component, the HiP-HOPS failure expression (local fault tree) can be derived from the AADL error state machine, guard_in and guard_out expressions. The HiP-HOPS Line elements can be derived from the AADL connections.

An AADL component error model is a state machine in which component behaviour is described in terms of states and transitions between states caused by error events. Figure 3.3 shows example error state machines for components `A` and `D`. Component `A` is initially in the `ErrorFree` state. If component `A` fails then its state changes to `Failed`. Once in the `Failed` error state it propagates the event `LossData` from `A.out`. When component `D`, which is initially in the `ErrorFree` state receives an input error event called `LossData1` it changes to the error state `Failed`. This error state can be also reached when the component itself fails (a

`Fail` error event is detected). Once in this error state, component `D` propagates output error event `LossData2`. This event may in turn cause a transition in the error model of some other connected component.

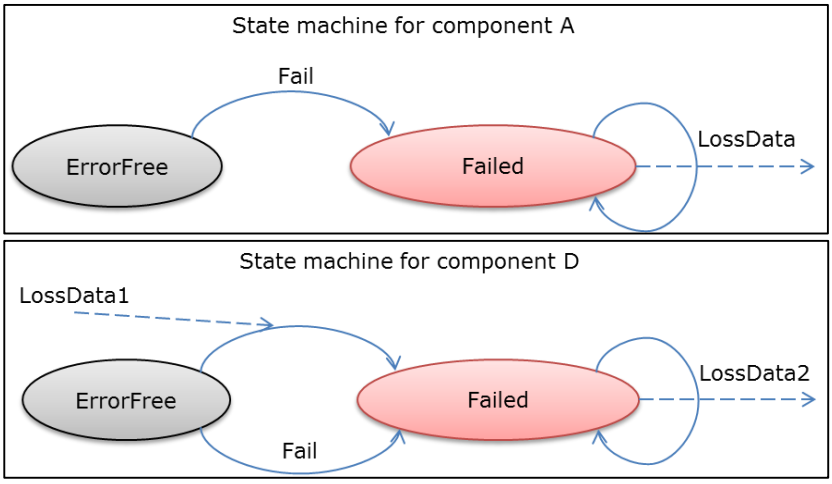

Figure 3.3 Error state machine for component `A` and `D`

The first stage of the transformation is to transform the component error models of each AADL component into a HiP-HOPS component fault tree expression. Each possible output of the error state machine is a top event for a fault tree. For example, in the component `D` shown in Figure 3.3, if we set the output propagation (`LossData2`) as the top event then the equivalent failure expression or fault tree is:

```
LossData2 = LossData1 OR Fail                                    (1)
```

This failure expression is constructed by identifying paths from the initial state to the final state corresponding to the top event in the state machine error model. The introduction to the formal algorithm is given in the following section.

3.1.1.1 Translation of error model state machine to fault tree

The error model associated with an AADL component or connection is essentially a state machine (Feiler and Rugina, 2007). The component error behaviour is described in terms of logical error states in the presence of faults. To construct the Boolean failure expression from an error state machine, a conversion algorithm, presented in (Rauzy, 2002; Mahmud et al., (2010, 2011)), is used. Essentially, the algorithm generates a fault tree for each output propagated from the state machine with that output as the top event. Below the top event is an OR-gate. There is one input to the OR-gate for each path from the initial state to the final state corresponding to the top event. This is because each path represents an alternative way of reaching the final state.

To traverse a path to the final state, each event that controls a state transition on that path must occur hence a path is represented by an AND-gate in which the inputs are the events that occur on the path. By using the above conversion algorithm, one can transform an AADL component error state machine to the corresponding component's local fault tree.

Note that the conversion algorithm, presented in Mahmud et al., (2010, 2011) can be used only for producing general Boolean failure expression (i.e. producing equation (1)) from an error state machine. Thus, the conversion algorithm needs to be adapted for this research in order to produce a HiP-HOPS specific failure expression from AADL state machines. To produce a HiP-HOPS-specific Boolean failure expression i.e. the notation <FailureClass>-<PortName>, further transformation algorithms are required. This includes the algorithms for mapping error states and events to component ports, transforming guard_in(out) expressions to fault trees, and transforming AADL connections error model to HiP-HOPS fault trees. These algorithms are discussed in the following sections.

3.1.1.2 Mapping error states and error events to component ports

Returning to the example of Figure 3.3, notice the absence of port names in the expression (1) above. Error events enter and leave components via ports. To represent this information in HiP-HOPS, expression (1) would be written as

```
LossData2-out = LossData1-in OR Fail                    (2)
```

Equation (2) specifies that when the component receives an input error propagation `LossData1` through its input port `in`, the component will propagate the error `LossData2` through its output port `out`. In general, the HiP-HOPS name for an error event consists of a basic event name, known also as a failure class or generic error followed by a port identifier. The notation <FailureClass>-<PortName> indicates the type of failure and the port from which it propagates.

The port and event name information required to construct the HiP-HOPS, expression (2) may be obtained from the guard_in error property of the AADL model. The guard_in property associates a local error event with events from other components that may propagate along a connection to an input port. The guard_in property can be used to rename error events or states or to define new events in terms of other events and states. For example, the AADL guard_in property at the input port `in` of `D` might be

```
guard_in => LossData1 when in[LossData],
            mask when others
            applies to in;
```

This expression means that the propagation of the error event `LossData` to the input port `in` of `D` will trigger the `LossData1` error event in `D`. Other error events from `A` that arrive at the input port are "masked" i.e. do not enter the port. The port name that the guard_in property applies to can be used to translate the AADL error `LossData1` into a HiP-HOPS failure class, i.e.

```
LossData1-in = LossData-in                                    (3)
```

Figure 3.4 shows the general form of the guard_in expression. In more detail, `e1` is the name of an incoming (in or in-out) propagation declared in the error model and `x1` is the expression to be evaluated to determine whether the `e1` event occurs.

The form of the expression can

- be a single in or in-out port.
- reference one or more outgoing propagations or error stats of the error model of some components connected to the given component through some ports of the given component.
- reference specific error states in the error model of the given component.
- contain Boolean operators, not, or, and, ormore and orless.

The `port` names that appear in the expressions refer to component's input port names through which the error propagations or error states will be propagated. The input port names i.e. `p1, p2,…, pk` that are listed in the `applies to` clause specifies that the defined guard_in property is only applied to these input ports. These input ports are used to qualify the defined input error propagations.

```
guard_in => e1 when x1,
            e2 when x2,
            e3 when x3,
            [mask when em],
            e4 when x4,
            [e5 when others]
            applies to p1, p2,…,pk;
```

Figure 3.4 The general form of the guard_in property

The guard_in property consists of a sequence of clauses as shown in Figure 3.4. Some clauses, shown in [], are optional. The `when` clauses are evaluated in the order of the declaration until the first one (whose Boolean error expression `xi`) is evaluated to be logically true. The incoming error propagation in that clause is considered to have occurred and causes the event named on the left of the corresponding `when` clause. The last `when` clause might specify

`others`, which means that the event on the left of `when` clause occurs if none of the previous clauses are true.

The guard_in property can also be used to specify the conditions under which the input error propagations are masked (ignored). If a `mask` clause expression is evaluated to true, the events appearing on the left of the `when` clauses do not occur and they will not have any impact on the given component.

Generally, each when clause in a guard_in property applied to an input port can be transformed into a fault tree. The top event is the event on the left of the `when` clause and the tree is defined by the expression of the right of the `when` clause. All the clauses in the guard_in property can be transformed into a list of fault trees. To produce HiP-HOPS failure classes, the top events are qualified by the input port name.

The top events of the fault tree are qualified as:

`ei - pj,`

where `i` indexes an event and `j` indexes a port and `pj` is a port name appearing in the `applies to` clause. Figure 3.5 shows partial fault trees generated from transforming the guard_in property shown in Figure 3.4.

Under each top event is an `AND` gate. The input error propagations `ei` are evaluated in the order of declaration until the first `when` clause (i.e. Boolean error expression) is evaluated to be logically true. This means that, for example, if the third when clause is evaluated to be logically true then both the first and the second when clause must have been firstly evaluated to be logically false. Thus, a `NOT` gate should be added for each `when` clause fault tree to specify that all the clauses before the current clause must be evaluated to be false. Furthermore, when the `mask` clause is evaluated to be true then all the errors i.e. input error propagations `ei` defined in the guard_in property are masked.

The top events i.e. `e1` to `e5` are qualified by port `p1.` The fault tree for the top event of `e2-p1:`

```
e2-p1 = NOT (x1) AND x2
```

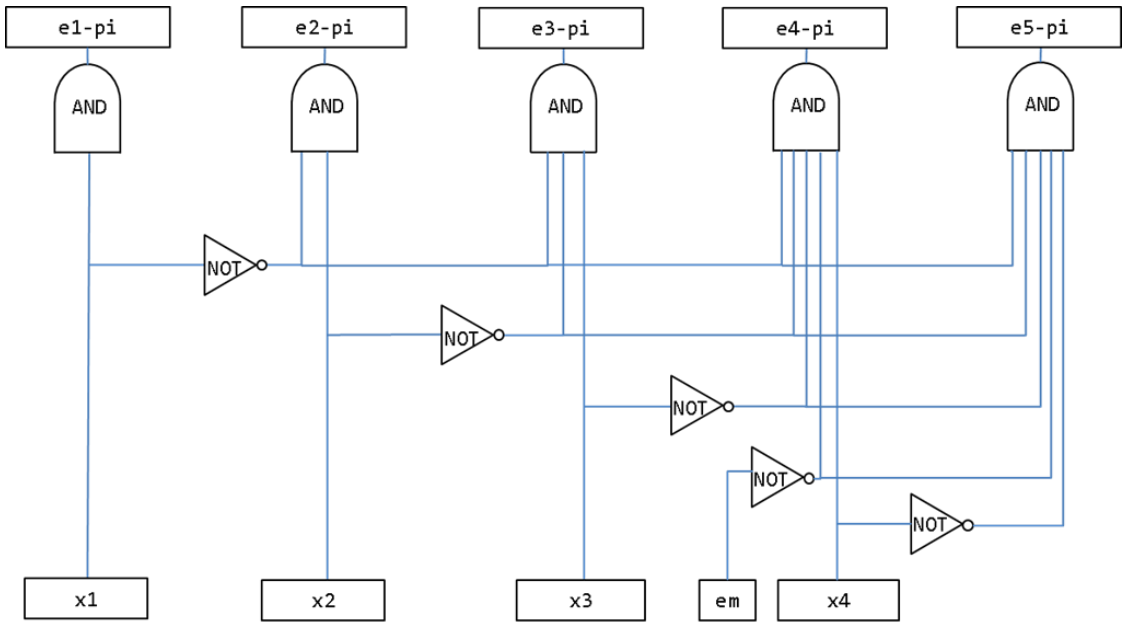

Figure 3.5 The fault trees generated from the guard_in property shown in Figure 3.4. The top events i.e. e1 to e5 are qualified by port pi. The trees shown above are for the port pi. There is one set of trees for each port p1 to pk in the applies to clause

Before give a formal algorithm, a small worked example of how the transformation algorithm from guard_in to fault tree works is given. Assume there are 5 `when` clauses in the guard_in (see Figure 3.4), listed in a sequence. The left-hand-side of the `when` clause is an event `ei`, the right-hand-side is an expression, `xi`. The `when` clause `m = em` is a mask clause. The list of `when` clauses is

```
<e1 = x1, e2 = x2, e3 = x3, m = em, e4 = x4, e5 = others>
```

The first step of the algorithm is to collect all sub-sequences of increasing length into a sequence.

```
<<e1 = x1>,
 <e1 = x1, e2 = x2>,
 <e1 = x1, e2 = x2, e3 = x3>,
 <e1 = x1, e2 = x2, e3 = x3, m = em>,
 <e1 = x1, e2 = x2, e3 = x3, m = em, e4 = x4>,
 <e1 = x1, e2 = x2, e3 = x3, m = em, e4 = x4, e5 = others>>
```

Delete the sequence with the last element equal to the mask expression.

```
<<e1 = x1>,
 <e1 = x1, e2 = x2>,
 <e1 = x1, e2 = x2, e3 = x3>,
 <e1 = x1, e2 = x2, e3 = x3, m = em, e4 = x4>,
 <e1 = x1, e2 = x2, e3 = x3, m = em, e4 = x4, e5 = others>>
```

Map each sequence into a pair, the first element of the pair is the sequence less the last element, the second element of the pair is the last element of the sequence.

```
<(<>, e1 = x1),
 (<e1 = x1>, e2 = x2),
 (<e1 = x1, e2 = x2>, e3 = x3),
 (<e1 = x1, e2 = x2, e3 = x3, m = em>, e4 = x4),
 (<e1 = x1, e2 = x2, e3 = x3, m = em, e4 = x4>, e5 = others)>
```

Map each pair (call it the source pair) to a pair (call it the target pair). The first element of each source pair is a sequence. From this sequence, construct the sequence containing only the `when` clause right-hand-side expressions. This is the first element of the target pair to be constructed. The second element of the target constructed pair is the same as the second element of the source pair.

```
<(<>, e1 = x1),
 (<x1>, e2 = x2),
 (<x1, x2>, e3 = x3),
 (<x1, x2, x3, em>, e4 = x4),
 (<x1, x2, x3, em, x4>, e5 = others)>
```

Map the logical `not` function to each expression in each sequence that is the first element of every source pair.

```
<(<>, e1 = x1),
 (<not x1>, e2 = x2),
 (<not x1, not x2>, e3 = x3),
 (<not x1, not x2, not x3, not em>, e4 = x4),
 (<not x1, not x2, not x3, not em, not x4>, e5 = others)>
```

Map each source pair to a target pair as follows. The first element of the target pair is produced by extending the sequence that is the source pair first element by adding the expression that is the right-hand-side of the `when` clause that is the second element of the pair. The second element of the target pair is produced from the left-hand-side event in the `when` clause that is the source pair second element.

```
<(<x1>, e1),
 (<not x1, x2>, e2),
 (<not x1, not x2, x3>, e3),
 (<not x1, not x2, not x3, not em, x4>, e4),
 (<not x1, not x2, not x3, not em, not x4, true>, e5)>
```

Map each pair into a fault tree constructed as follows. The expression of the fault true is constructed from folding the logical and function into the expressions in the first element

sequence. The top event of each fault tree is the second element of the pair hyphenated with the port indexed by i.

```
<e1-pi = x1,
 e2-pi = not x1 and x2,
 e3-pi = not x1 and not x2 and x3,
 e4-pi = not x1 and not x2 and not x3 and not em and x4,
 e5-pi = not x1 and not x2 and not x3 and not em and not x4>
```

This step is repeated for i=1 to k, i.e. for each port in the "applies to" clause.

The formal algorithm for transforming AADL guard_in property to HiP-HOPS fault trees is given in Figure 3.6 below. The algorithm codes each of the steps shown in the above example. In each step, a function is applied to a source sequence and produces a target sequence. In the next step, the target sequence of the previous step becomes the source sequence and so on.

```
Let Sequence G = com.ErrorProperty.inErrorPropagationGuard.list;
Let Sequence GR = <>;
for (i = 1 to G.length) {
  GR.Add(take(G, i));  // collect sub-sequences
}
G = {s : GR | s.last.lhs != mask}  // delete mask sub-seqc
// split sequences at last element and construct pairs
G = {s : G • pair({take(s, s.length – 1)}, s.last)}
// change each 'when' clause of form e = x to x
G = {p : G • pair({when: p.first • when.rhs}, p.second)}
// insert 'not' for each expression
G = {p : G • pair({x: p.first • not x}, p.second)}
// extend sequence with 'when' expression
G = {p : G • pair(p.first.add(p.second.rhs), p.second)}
// insert 'and' between expns of sequence and produce FT expns
for (i = 1 to k){
  Gi = {p : G • tree(p.second-pi, fold(and, p.first))}
}
```

Figure 3.6 The formal algorithm for transforming AADL guard_in (guard_out) property into HiP-HOPS fault trees

The algorithm first collects all the input error propagation clauses i.e. all guard clauses defined in the guard_in property to a sequence `G`. The sequence `GR` is a set of set guard clauses. Each element of `GR` is a sequence of guard clauses obtained by taking partial elements (guard clauses) in sequence `G`. This is done by `take(G, i),` which gives the first `i` elements of sequence `G`. Each element of `GR` contains all the required Boolean error expressions (defined in guard

clauses) to create a fault tree for an input error propagation identifier specified in the guard_in property.

The algorithm makes use of the filter function (the | in the algorithm denotes a filter function) which tests each element of a sequence with a Boolean function and collects all the elements that are true.

The variable `G` is then defined to be the set generated (• denotes generator operator) by collecting the value of each `pair` function.

The `pair` function creates a pair. The first element of the pair is `pair.first`. The second part element of the pair is `pair.second`. A `when` clause event is `when.lhs and` a `when` clause expression is `when.rhs`.

The `fold` function inserts a two argument function between each element of a collection to produce a single expression. The `tree` function creates a fault tree from a top event and a Boolean expression. The result fault tree expression is in the form <e> - <p>, where `p` is the port name specified in the `applies to` clause of a guard_in property.

Note that if an input error propagation (e.g. `e`) is defined in more than one guard_in property applied to multiple ports e.g. `p1` and `p2`. In this case, it may guard the situation that different types of incoming error propagations through different input ports can be mapped into the same incoming propagation `e`. Thus, this input error propagation `e1` is transformed as a disjunction (OR) of port qualified names:

```
e = e-p1 OR e-p2                                                    (4)
```

Note that if there is no guard_in error property defined for a locally defined input error propagation, then this means that no input error name mapping is required i.e., input and output error propagations have the same name. Note also that in the situation in which a component has more than one input port and no guard_in error property is defined for a given error that may propagate to those input ports then the error may propagate through any of the input ports. To translate this situation into a HiP-HOPS failure expression, an event name of the form <FailureClass>-<PortName> is created for each port. The event which is the propagation of the error to any port can then be represented as a disjunction (OR) of port qualified names. In general, each locally defined input error propagation `e` that appears in a state machine is transformed into a disjunction (OR) of the names constructed by appending each of the input ports to the input error propagation:

```
e = e-in1 OR e-in2 OR ... e-inN                                    (4)’
```

where `in1, in2, ... e-inN` are the input ports through which `e` may propagate to the component.

The HiP-HOPS names of error events that propagate out of a component may be constructed in an analogous manner. In the presence of a guard_out property at a port, the error name mapping at that port can be used to create the HiP-HOPS name. In the current example, suppose that there is the following guard_out property at the output port `out` of `D`

```
guard_out => LossData2
             when self [Failed],
             mask when others
             applies to out;
```

which means that the component will propagate an output error propagation called `LossData2` through output port `out` when it is in the state `Failed`. The other error propagations propagate through this output port are “masked” i.e. not propagated out. Again, the fact that this guard_out property is associated with the port `out` may be used to qualify the `LossData2` event, i.e.

```
LossData2 = LossData2-out                                          (5)
```

The guard_out also allows the mapping of the component `Failed` state to the output propagation `LossData2` (HiP-HOPS failure class `LossData2-out`), i.e.

```
LossData2-out = Failed                                             (6)
```

In the absence of any guard_out error property, the output error propagations defined for a component will propagate through each output port of that component. For a given set of errors that propagate out of a component, a HiP-HOPS failure class is created for each error. For a given set of output ports, each port is used to qualify the failure class. More formally, suppose that there are number n of output ports (n >= 1), i.e., `out1`, `out2, . . . , outn`, then we obtain:

```
OutputErr = OutputErr-Out1 = OutputErr-Out2 = ... = OutputErr-Outn (7)
```

For component `D`, there is only one output port `out`. Thus, in the absence of any guard_out error property, based on the Boolean logic shown in (7), we obtain:

```
LossData2 = LossData2-out                                          (8)
```

For component `D`, from Boolean logic (1), (2) and (3) we now obtain:

```
LossData2-out = LossData-in OR Fail                    (9)
```

This is the HiP-HOPS Boolean failure expression (fault tree) for the component `D` and all event names are expressed as <failure class>-<port name>.

3.1.2 Transformation of AADL connections to HiP-HOPS Lines

Using the AADL state machine and guard_in(out) expression to HiP-HOPS fault tree transformation described in the previous section, we can obtain a local fault tree for each component in the system. To create a whole system fault tree, HiP-HOPS needs information about how errors propagate between components. This information is represented using HiP-HOPS Lines. The HiP-HOPS Line element describes how events, typically error events, propagate from one component to another. The HiP-HOPS Line concept describes a set of connected ports. The Line contains a set of HiP-HOPS Connection objects (see the right hand figures shown in Figure 3.7). Each Connection describes the propagation of event to a specific port from other ports. A Line connecting two ports will have two Connections if events flow in both directions.

Figure 3.7 presents a high-level overview of the transformation in the form of mappings between constructs of the two metamodels. In Figure 3.7 the left hand figure shows the AADL connection metamodel and the right hand figure shows a corresponding HiP-HOPS Line metamodel. For example, an AADL EventConnection object (with type of Connection) that contains dst (destination port) and src (source port) is mapped as a HiP-HOPS Line objects. A HiP-HOPS Line type contains a list of ConncetionsType and each ConncetionsType has three attributes: failureClass, port and portExpression. The details of the transformation mappings i.e. how AADL connections can be transformed to HiP-HOPS Line objects are discussed as follows.

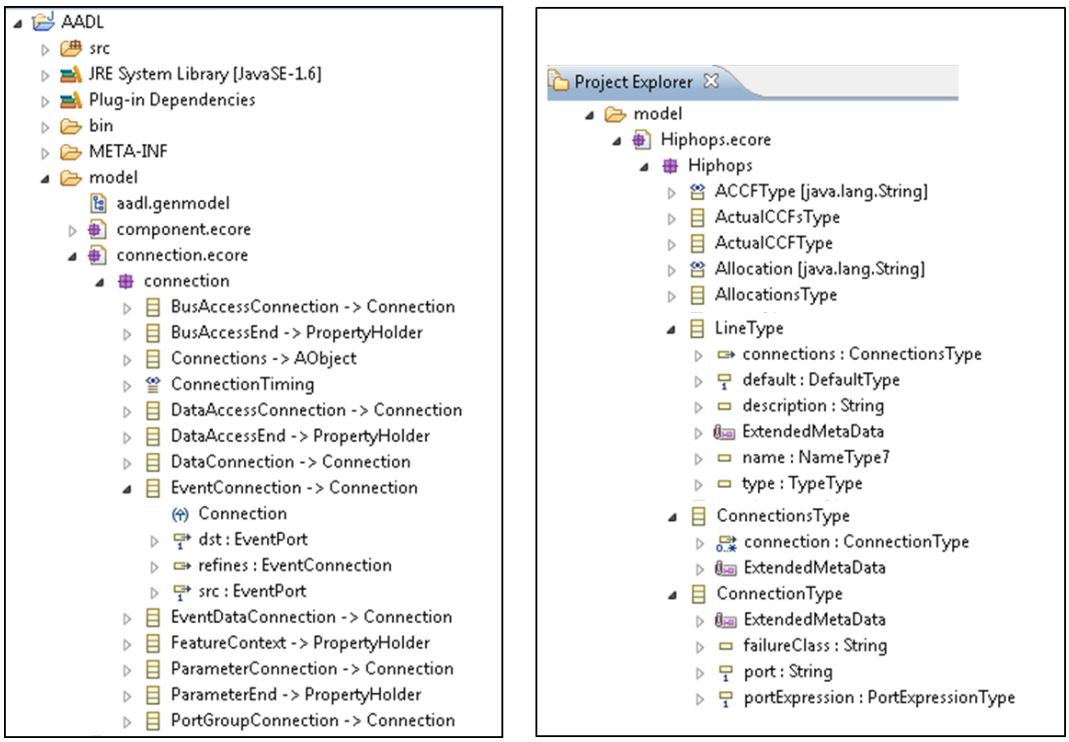

Figure 3.7 Left hand figure shows the AADL connection metamodel and right hand figure shows a corresponding HiP-HOPS Line metamodel

The information required to create HiP-HOPS Lines can be obtained from the AADL connection objects. To give a simple description of the transformation from AADL connections to HiP-HOPS Lines, consider a simple case in which only one AADL connection (called `EventConnection1`) is defined between two components `A` and `B`. Assume also that there is an "in out" error propagation called `LossData` which is defined in an error model and this error model is associated with both components `A` and `B`. The partial AADL description of this connection is

```
EventConnection1: event port A.out -> B.in;
```

For this AADL connection, the error events will propagate from the source port `A.out` to the destination port `B.in`. In particular we can associate a connection logic failure expression (called HiP-HOPS PortExpression) with port `B.in` which describes the failure at component `B` in terms of the output failure at `A`. Thus the HiP-HOPS Line for the connection from `A.out` to `B.in` would be constructed as follows

```
<Line>
  <Type>Directed</Type>
  <Connections>
    <Connection>
      <FailureClass>LossData</FailureClass> // failure in component
      <Port>B.in</Port>     // propagated into port
      <PortExpression>     //failure propagated when
        LossData-A.out
      </PortExpression>
    </Connection>
  </Connections>
</Line>
```

Each Line element contains a list of Connections. Each Connection describes how errors at one or more output ports (e.g. `LossData` at `A.out`) propagate to an error at an input port (e.g. `LossData` at `B.in`). The `<Port>` attribute identifies the port to which the error propagates. Since the Line is directed the error will propagate from the port (`A.out`) to the port (`B.in`). The PortExpression element is a Boolean expression containing AND, OR, and the names of other port or ports on the Line. The PortExpression describes the events at other components, i.e. `LossData` from `A.out`, which causes an event, in this case, `LossData` at `B.in` at the `in` port of component `B`.

The transformation of the above example AADL connection to HiP-HOPS Line is relatively straightforward as errors to the port `B.in` can come only from one port, i.e. `A.out`. The transformation transforms the AADL connection's destination port to HiP-HOPS Connection destination port and the AADL connection's source port to HiP-HOPS PortExpression. The AADL output error propagation `LossData` is transformed to a HiP-HOPS FailureClass and the portExpression is constructed in the style of <FailureClass>-<portname>.

Whereas an AADL connection joins only two ports, a HiP-HOPS Line may connect any number of ports. For each port in a Line to which error events may propagate, there is a HiP-HOPS Connection object that specifies how error events may propagate to that port from other ports in the Line. The HiP-HOPS Line also maps the names of events at source components to names of events in the destination component. Figure 3.8 illustrates an important difference between AADL connections and HiP-HOPS Lines. HiP-HOPS models multiple connections that fan-in to a single destination port using one HiP-HOPS Line. In Figure 3.8, each small circle and triangle denotes an AADL connection or a HiP-HOPS Line object respectively. In AADL, the connections between components shown in Figure 3.8 (left-hand side) are modelled using six connections (three out of A and three into E). In the same system modelled in HiP-HOPS, however, shown in Figure 3.8 (right-hand side), there are only four HiP-HOPS Lines. The three AADL connections to component E are transformed into one HiP-HOPS Line.

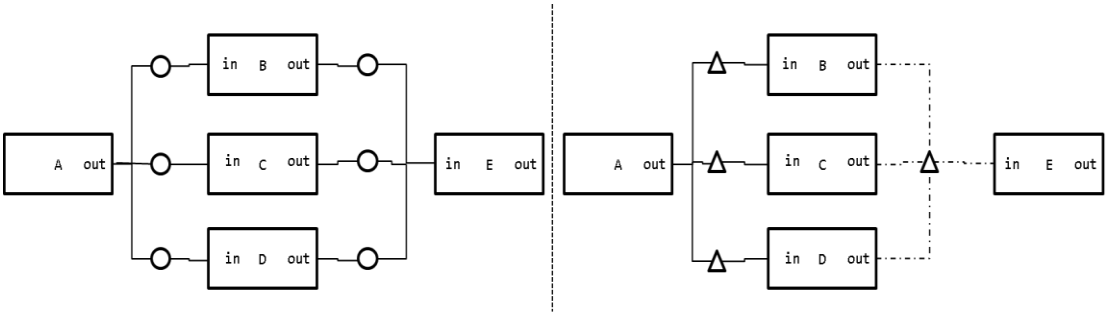

Figure 3.8 Left hand figure shows an AADL model with six connections, small circles. Right hand figure shows a corresponding HiP-HOPS model with four Lines, small triangles

Continuing with the above example, below is a partial AADL description of three connections from source ports `B.out, C.out` and `D.out` to destination port `E.in` respectively.

```
EventConnection4: event port B.out -> E.in;
EventConnection5: event port C.out -> E.in;
EventConnection6: event port D.out -> E.in;
```

Assume there is a single AADL error model associated with four components `B, C, D` and `E`. Assume that this error model has an "in out" type error propagation called `ValueIsHigh` and this error propagation will cause a state transition in `E`. This means that whenever component `E` receives a `ValueIsHigh` event at the input port `in`, irrespective of the source of the event, a `ValueIsHigh` event occurs in `E`.

Since the `ValueIsHigh` event may originate from either component `B`, `C` or `D`, the transformation introduces the OR logic operator into the port expression, i.e. `ValueIsHigh-B.out OR ValueIsHigh-C.out OR ValueIsHigh-D.out`. The port names `B.out, C.out` and `D.out` are available as the source ports of the connections. The corresponding Line object is given below.

```
<Line>
  <Type>Directed</Type>
  <Connections>
    <Connection>
      <FailureClass>ValueIsHigh</FailureClass>
      <Port>E.in</Port>
      <PortExpression>
         ValueIsHigh-B.out OR ValueIsHigh-C.out OR ValueIsHigh-D.out
      </PortExpression>
    </Connection>
  </Connections>
</Line>
```

Note that each failure class is represented in a distinct HiP-HOPS Connection within a Line. This models the situation in which different failure classes in the destination component may be caused by different failure classes from different source components although they all share the same fan-in connection topology. The transformation creates a Connection object for each of the failure classes defined in any of the connected components. The Connection object contains a PortExpression with the OR logic operator between each source port. This allows the failure class to originate from any of the connected components.

In the case that there different error models for each of the three components `B, C` and `D, t`he transformation checks whether an output propagation propagated to the destination port is defined in its connected component's error model. If it is defined then the source port will be included in the PortExpression object. For example, assume there are three output propagations (`ValueIsHigh1`, `ValueIsHigh2` and `ValueIsHigh3`) that can propagate to `E.in`. By checking the error model that is defined for each connected component (i.e., `B, C`, and `D`) it is possible to determine whether each output propagation is defined in these connected components. Assume, for example that `ValueIsHigh1` is defined both in component `B` and `C`, `ValueIsHigh2` is defined only in component `B` and `ValueIsHigh3` is defined in all the components `B`, `C` and `D`. This means that `ValueIsHigh1` could be propagated from the output of either component `B` or `C`, `ValueIsHigh2` is only propagated from component `B` and `ValueIsHigh3` could be propagated from either component `B`, `C` or `D`. Based on this data, the corresponding HiP-HOPS Line is given below:

```
<Line>
 <Type>Directed</Type>
 <Connections>
   <Connection>
      <FailureClass>ValueIsHigh1</FailureClass>
      <Port>E.in</Port>
      <PortExpression>
         ValueIsHigh1-B.out OR ValueIsHigh1-C.out
      </PortExpression>
   </Connection>
   <Connection>
      <FailureClass>ValueIsHigh2</FailureClass>
      <Port>E.in</Port>
      <PortExpression>ValueIsHigh2-B.out</PortExpression>
   </Connection>
   <Connection>
      <FailureClass>ValueIsHigh3</FailureClass>
      <Port>E.in</Port>
      <PortExpression>
         ValueIsHigh3-B.out OR ValueIsHigh3-C.out OR
         ValueIsHigh3-D.out
      </PortExpression>
   </Connection>
 </Connections>
</Line>
```

To generalise, the corresponding algorithm for transforming AADL connections to HiP-HOPS Lines is given below. In a hierarchically structured model, the algorithm is applied to the top-level system and any sub-system.

```
Let DestPorts = {c : sys.Connections • c.destination};

ConnsSameDest =
  {p: DestPorts • {c: sys.Connections | c.destination = p}}

Lines = {cd : ConnsSameDest • Line({c : cd •
   {e : c.source.component.errorsPropagated •
      ConnectionH(c.destination,
                  OR{c : cd |
                          e ∈ c.source.component.errorsPropagated
                              • e-c.source.name})}})}
```

Figure 3.9 The formal algorithm for transforming AADL Connections into HiP-HOPS Lines

The formal description of the algorithm shown in Figure 3.9 should be read as follows. The variable `DestPorts` is defined to be the set generated (• denotes generator operator) by collecting the destination port of each connection in the system. The system is represented by the variable `sys`. The variable `ConnsSameDest` is defined to be a set of connection sets. In

each connection set, all the connections share the same destination port. For each destination port, a set of connections to that port is generated by filtering (| denotes filter operator) the connections with a destination equal to a given destination port. The variable Lines is the set of HiP-HOPS Line objects. For each set of connections in ConnsSameDest, a HiP-HOPS Line object is constructed. A Line is constructed from a set of HiP-HOPS Connection (ConnectionH). A HiP-HOPS Connection is constructed for each failure class that is propagated from any component that is at the source of any connection in the set of connections to a given destination port. The HiP-HOPS PortExpression is a disjunction because the error may propagate from any of the source components, hence

OR{c : cd • e-c.source.name} where e is a failure class and c.source.name is a port name.

The operator OR denotes e-c1.source.name OR e-c2.source.name .. e-cn.source.name, for each connection ci in cd.

In this definition, c.source.component.errorsPropagated denotes the set of output error propagations (HiP-HOPS failure classes) from the component at the source of connection c. These error propagations can be obtained from the error model of the source component. Each failure class collected, denoted e, is used to qualify the connection source port name, i.e. e-c.source.name.

HiP-HOPS allows a number of abbreviated syntax forms in order to improve readability. Note that if the Line does not include any failure class element, this signifies that, for all failure classes, the error will propagate from A.out to B.in. If a model is not intended to be human-readable then such abbreviations are unnecessary. Omitting such abbreviation typically simplifies the model to model transformation and is the approach adopted in this work.

3.2 Model transformation method implementation using transformation rules

The Eclipse Modelling Framework (EMF) (Steinberg et al., 2009) is a modelling framework to provide a highly integrated tool platform. Different plugins for different models can be developed based on EMF. The Open-Source AADL Tool Environment (OSATE) developed by SEI (2004) is a set of plug-ins based on Eclipse and the EMF. The OSATE plug-ins were used in the work reported in this thesis.

The transformation from AADL to HiP-HOPS modes consists of two parts. The first part is the transformation of the component error models into a HiP-HOPS individual component fault trees. The second part is the transformation of the connections into HiP-HOPS Lines. The

algorithm that traverses the paths of the error model state-machine in order to produce HiP-HOPS fault trees (i.e. failureExpression) has been implemented directly in code, namely Java. A code implementation was chosen because of the complexity of the transformation. The transformation maps from multiple elements of different types. This is something that is not conveniently implemented using a function (or helper) in ATL. Furthermore, this algorithm does not require extensive navigation of the AADL model and so Java proved convenient. It was also straightforward to integrate the Java implementation as part of the plug-in integrated into the OSATE tool because OSATE and ATL are all Java-based tool sets.

The second part of the transformation has a greater requirement to navigate the source AADL model. For this reason, the thesis has chosen to perform the model transformation from AADL to HiP-HOPS using a declarative model transformation language. Model to model transformation languages should be well suited for the semantic mapping transformation, since both input and output are models. The thesis chose the ATLAS Transformation Language (ATL) (Jouault et al., 2008) which is a hybrid language containing a mixture of declarative and imperative constructs.

In the context of transforming AADL to HiP-HOPS, the AADL dependable model conforming to AADL metamodel is the source model. The HiP-HOPS input file is the final output of the transformation, which confirms to the HiP-HOPS schema definition. The transformation rules (sematic mappings) written in the ATL language is a model conforming to the ATL metamodel. All metamodels conforms to the Ecore (Steinberg et al., 2009) modelling language.

The advantages of using a high-level model transformation language such as ATL has been discussed in section 2.4.5. Another benefit of using a high-level rule language such as ATL is that it is possible to write rules which resemble the algorithms given in the previous section. The algorithms used to transform AADL connections into HiP-HOPS Lines depend heavily on iteration, filtering, generation and collection operations. ATL includes operators for iterating over collections, filtering the results and generating values from arbitrary expressions.

3.2.1 ATL rule implementation

In this section, a few of the most important rules are described. Without giving a detailed description of the implementation it does illustrate the nature of a rule-based transformation.

Figure 3.10 shows the rule description for transforming AADL connections to HiP-HOPS Lines. This rule is responsible for the construction of a `hiphops!Line` element and is the ATL implementation of the algorithm given in Figure 3.9. In the left-hand-side of the rule, the

variable `conn` is instantiated to the set of AADL connections in the source model. ATL is a hybrid rule language and thus allows functions to be defined and called within a rule. The expression `conn.getConnsSameDest()` is such a function call. The function (known in ATL as a helper) returns a collection of set of connection such that all the connections of any given set share the same destination port. A `hiphops!Line` has a `connections` element. The value of this element is set from the value of the `_connections` element which is a set of `hiphops!Connection`. Each `hiphops!Connection` has three main attributes, the `fc` attribute is the failure class (propagated error), the `pn` attribute is the destination port name and the `pe` attribute is the port expression. The values of these variables are found using helper functions. The helper functions in this rule are discussed next.

```
rule Connection2Line {
  from
    conn : core!Connections
  to
    li : distinct hiphops!Line foreach (
        conDestPort in conn.getConnsSameDest()) (
          connections <- _connections),
          _connections : hiphops!Connections(
            connection <- _connection),
          _connection : distinct hiphops!Connection foreach (
            elm in thisModule.getConnectionH(conDestPort)) (
              fc <- thisModule.getFailureClass(elm),
              pn <- conDestPort.getPortName(),
              pe <- thisModule.getPortExpression(elm))
}
```

Figure 3.10 Rule transforming AADL Connections to HiP-HOPS Lines

The ATL helper function `getConnsSameDest()` is shown in Figure 3.11. The result of this helper is a set connections that share the same destination port. In this helper, the variable `self` is a set of connections. This helper firstly collects from all connections, all destination ports, `c.value.dst.name`, in a set (i.e., without duplication). This set of ports is the input to the second collection. In this collection, for each of the destination port (`p`), there is a second iteration over that set of connections (variable `c`). The if-statement is executed for each connection. If the destination port of connection (`c.value.dst.name`) is the same destination port (`p`) then add this connection to the result set `res`.

```
helper context core!Connections def : getConnsSameDest():
  OrderedSet(core!Connection) =
    self.contents -> collect (c | c.value.dst.name).asSet() ->
      collect (p | self.contents ->
        iterate (c; res: OrderedSet(core!Connection) = OrderedSet{}
                  | if p = c.value.dst.name
                    then res.append (c.value)
                    endif));
```

Figure 3.11 The helper function that returns all connections that share the same destination port

The ATL helper function `getConnectionH()` is shown in Figure 3.12. In this helper, the variable `ConDestPort` is a single set of connections, all with the same destination port, as produced by `getConnsSameDest()`. From this set, a HiP-HOPS Line is constructed. This helper firstly collects all errors propagated from all the source ports in `ConDestPort`. The expression `c.source.component.errorPropagated.name`, returns the names of the errors propagated from the component at the source of the connection `c`. For each error `e`, a string portExpression is constructed using a second iteration over the set of connections. Wherever a connection source port propagates an error equal to the given error `e`, a `e-c.source.name` element is constructed. This element, together with the `OR` operator is appended to the variable `portExpression`.

```
helper def:getConnectionH(ConDestPort: OrderedSet(core!Connection)):
  OrderedSet(String) =
    ConDestPort -> collect (c |
      c.source.component.errorPropagated.name).asSet() ->
        iterate (e; ConnectionH: OrderedSet(String) = {} | ConDestPort ->
          iterate (c; portExpression: String() = ''|
                    if c.source.component.errorPropagated.name.asSet()->
                       exists(i | i = e)
                    then portExpression + e + '-' + c.source.name +'OR'
                    else portExpression endif)
          ConnectionH.append(portExpression));
```

Figure 3.12 The helper function that returns all connections in a Line. A list of connections is constructed for each of error propagated to the destination port

Note that the implementation of the helpers expressed in ATL shown in Figure 3.11 and Figure 3.12 shows poor maintainability. This is because the static variable path e.g. the `if` condition shown in Figure 3.12 (`c.source.component.errorPropagated.name`) is used to collect the required information. If the path for collecting the required information is changed the helper code has to be changed. This reduces the maintainability of the helper. In this research, the implemented ATL transformation rules and helpers are used only for demonstration purpose

and the maintainability and efficiency of the transformation language is not considered as a key concern. In the case that maintainability and efficiency is a main concern for the transformation, other model transformation languages and tools e.g. Tefkat (Lawley et al., 2004) may be used. Furthermore, the model transformation in this research is developed manually. Some code patterns are implemented repetitively, reducing code reusability and increasing the probability of programing errors. To allow an efficient (semi-automated) model transformation development, model weaving and matching transformation (Del Fabro and Valduriez, 2009) could be used.

The transformation is driven by the top-level rule which is shown in Figure 3.13. This rule has the effect of calling the `getAllComponentInstance()` rule on the root system and the `getAllConnectionsInTopsystem()` rule on the root system. These rules transform AADL components to HiP-HOPS components and AADL connections, to HiP-HOPS Lines respectively. The `component` element and the `lines` element are the two main elements of `hiphops!System.`

```
rule SystemInstance2System {
  from
    si : instance!SystemInstance -- rootSystem
  to
    model: hiphops!Model (system <- _system),
    _system: hiphops!System (
                component <- si.getAllComponentInstance(),
                lines <- si.getAllConnectionsInTopsystem())
}
```

Figure 3.13 Rule transforming AADL top system instance to HiP-HOPS top system

Below the top level, there are two kinds of component categories to be considered. The first kind is an AADL component that contains no subcomponents. The second kind is component which contains sub-components, i.e. a sub-system. Each kind has its own rule. The rule shown in Figure 3.14 transforms AADL components which contain no subcomponents. The rule shown in Figure 3.15 transforms AADL components which contain subcomponents.

The rule shown in Figure 3.14 constructs a `hiphops!Component` element from an AADL component that is not the root system nor a sub-system. Within this rule, the important elements are the `ports` and `implementation` elements. The rule specifies that the `ports` element is constructed by the rule `ci.getAllFeatureInstance()` and the `implementation` element is constructed by constructing a `hiphops!Implementation`. A `hiphops!Implementation` has one `fData` element, which is constructed by the rule `ci.getErrorModelImplemen()`.

```
-- ComponentInstance is the superclass of SystemInstance
rule ComponentInstance2Component {
  from
    ci : instance!ComponentInstance (
       not ci.IsRootSystem() and not ci.IsSubSystem())
  to
    com : hiphops!Component(ports <- ci.getAllFeatureInstance(),
                            implementation <- _implementation),
    _implementation: hiphops!Implementation(
                            fData <- ci.getErrorModelImplemen())
}
```

Figure 3.14 Rule transforming an AADL component instance to a HiP-HOPS component

The rule shown in Figure 3.15 is similar to the rule shown in Figure 3.14 except that it applies to sub-systems. In addition to constructing the elements constructed by the rule shown in Figure 3.14, an extra HiP-HOPS element `system` is bound to the HiP-HOPS `implementation` element by constructing a `hiphops!System`. This allows the rule to recursively construct the `component` element by calling the rule `ci.getAllComponentInstance()`. The `lines` element is constructed by the rule `ci.getAllConnectionsInSubsystem()`.

```
rule ComponentInstanceIsTypeOfSubSystem2Component {
  from
    ci : instance!ComponentInstance (
       not ci.IsRootSystem() and ci.IsSubSystem())
  to
    com : hiphops!Component(
                ports <- ci.getAllFeatureInstance(),
                implementation <- _implementation),
    _implementation: hiphops!Implementation(
                fData <- ci.getErrorModelImplemen(),
                system <- _system),
    _system: hiphops!System(
                component <- ci.getAllComponentInstance(),
                lines <- ci.getAllConnectionsInSubsystem())
}
```

Figure 3.15 Rule transforming AADL sub-system to HiP-HOPS component

Note that the connection data is stored differently in AADL according to whether the connection is part of the top system instance or a component instance. For this reason there are two retrieve methods: (1). Retrieve and transform AADL connections in root system instance to HiP-HOPS Lines in root system (see the ATL helper `getAllConnectionsInTopsystem()` in Figure 3.13). (2). Retrieve and transform AADL connections in component instance with category of system (i.e., sub-system) to HiP-HOPS Lines in sub-system (see the ATL helper

`getAllConnectionsInSubsystem()` in Figure 3.15). Once the connections are retrieved, however, in either situation, the same connection transformation is applied.

The transformation rules have been automated by implementing in a tool called AADL2HiP-HOPS. The tool has been integrated as a plug-in into the AADL development environment OSATE. The tool performs a model-to-model transformation (see M2M in Figure 3.16) in memory, where the AADL system architecture model and the system error model are transformed into an intermediate HiP-HOPS Ecore model. The model-to-model transformation is a conceptual mapping between the two domains and preserves the semantics of the source AADL model. The HiP-HOPS Ecore model contains all the HiP-HOPS required data produced from AADL for dependability analysis but it does not conform to HiP-HOPS' schema. The in-memory HiP-HOPS Ecore model is then output to a file in XML format (see M2T, i.e., Model to Text transformation in Figure 3.16). The model-to-text (M2T) transformation transforms the output of the first phase transformation (AADL model to intermediate HiP-HOPS model) into the concrete syntax of target model (HiP-HOPS model). The M2T output is a standard HiP-HOPS XML format for input directly into HiP-HOPS.

The separation of two different concerns of the transformation i.e. M2M and M2T allow each transformation to be a separate module, which can be developed, changed and tested independently. Furthermore, it allows the two transformations to evolve independently i.e. a change in AADL will only affect the M2M transformation and a change in HiP-HOPS will only affect the M2T transformation.

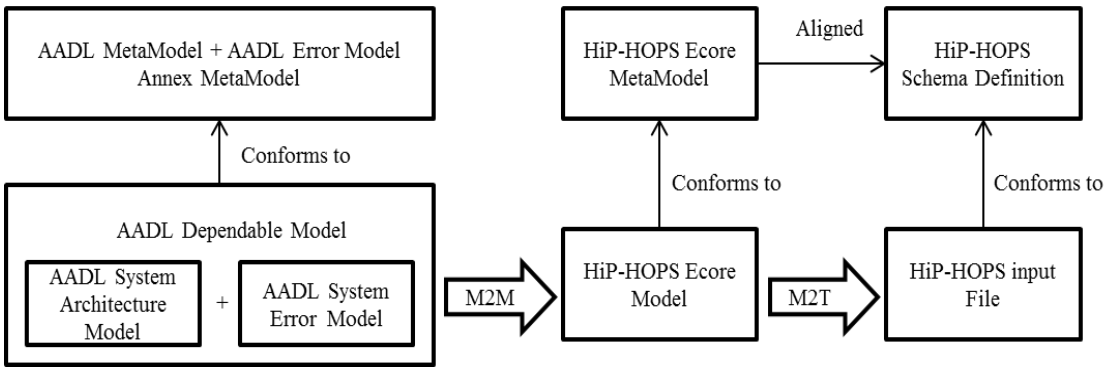

Figure 3.16 The overview of transformation design, an in-memory model to model transformation is followed by conversion of the model to XML format for input into HiP-HOPS

The model-to-model transformation is designed by considering the input requirements of the HiP-HOPS analysis. The target HiP-HOPS metamodel is firstly analysed and the required dependability data and elements such as components and their failure data and lines between the components are identified. Secondly, the AADL metamodel is analysed to identify those

elements that are required to create the target elements. A transformation rule is then defined for each of the mapping i.e. each rule defines what source model elements are required to create the target model element and how the target element is constructed from the source elements. The identified semantic mappings are derived by the analysis of both source and target metamodels.

To check that the identified mapping and rules are correct, an expert with good knowledge of AADL and HiP-HOPS may inspect the transformation design. The expert reviews all the identified mappings and also the corresponding transformation rules. The expert then checks to see if all the required target metamodel elements that are created and if all the required source metamodel elements that are covered to create the target model elements.

Also, the ATL transformation tool can help to support the correctness decision making. The ATL transformation tool can record how many rules are executed when the transformation is done. The ATL transformation tool can also track how many source and target model elements are covered when each transformation rule is executed. This can help to ensure that the developed rules are all executed and all the required elements in both source and target model are covered.

In this research, 7 rules are defined for the model transformation between the two domains. In this case, a manual expert review process is sufficient and applicable. However, it may make the review process much harder and time consuming if the size of the metamodels is quite large or dozens or more rules are applied to the transformation. One solution to this problem is to build a weaving model (Del Fabro and Valduriez, 2009) that contains different kinds of relationships between the source and target metamodels. These relationships capture different transformation patterns. Based on the weaving model one could then automatically create a model transformation design between the source and target metamodels. Also the correctness and completeness of the transformation could be automatically checked based on the weaving model.

3.3 Model transformation for AADL connections with error models

For AADL connections associated with an error model, a further AADL to AADL model transformation may be required. This is because the HiP-HOPS Line has no error model (failure data) defined for it. For this reason, any AADL connection with an associated error model is transformed into a HiP-HOPS component object, for which failure expressions may be defined. Consider the example shown in Figure 3.17, where two components `A` and `B` are connected by connection `C`. There is an associated error model defined for connection `C`. To translate this

AADL connection to an equivalent HiP-HOPS component, this AADL connection (with the associated error model) is first transformed to a special AADL component which we call a `ConnectionComponent`. The `ConnectionComponent` corresponding to the connection `C` is shown as `CC` in Figure 3.18. The `ConnectionComponent` has two ports, an input port named `in` and an output port named `out`. The `ConnectionComponent` has an error model, which is the error model of the connection. The `ConnectionComponent` has no other features. The transformation described earlier may now be applied to the `ConnectionComponent` to produce a HiP-HOPS component. The connection error model is transformed as any other component error model.

As can be seen from Figure 3.18, the transformation from AADL connection to `ConnectionComponent` requires the construction of a `ConnectionComponent` and two new connections. One connection connects the source component (`A` in Figure 3.18) to the `ConnectionComponent` and the other connects the `ConnectionComponent` to the destination component (`B` in Figure 3.18). The algorithm for this transformation is described in Figure 3.19.

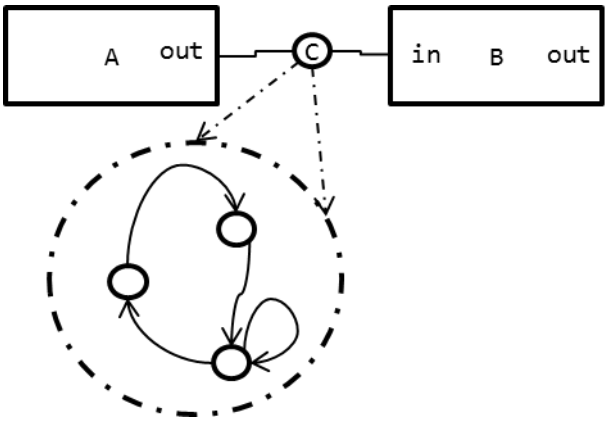

Figure 3.17 Two AADL components `A` and `B` and one AADL connection `C` with associated error model

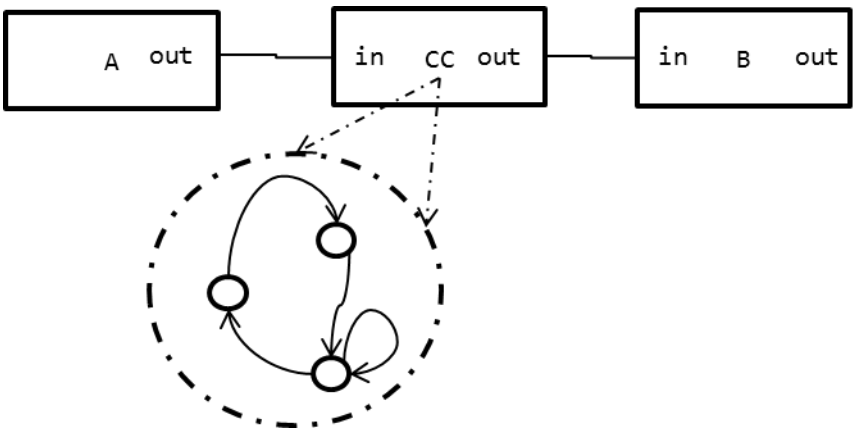

Figure 3.18 The connection C with associated error model shown in Figure 3.17 is transformed to two new connections and an intervening ConnectionComponent called CC

```
for (Connection c  in {sys.connections | c.hasErrormodel = true}) {
         CC = new ConnectionComponent();
         sys.components = sys.components ∪ {CC}
         c1 = MakeConnection(c.source, CC.in);
         c2 = MakeConnection(CC.out, c.destination);
         sys.connections = sys.connections \ {c} ∪ {c1, c2};
}
```

Figure 3.19 The algorithm for transforming AADL Connection with associated error model to two new AADL Connections and an intervening connection component called CC

The algorithm operates on a component-based model of a system which contains a collection of components and a collection of connections between those components. The first part is responsible for adding the connection component CC to each of the given target connection with associated error model. The second part adds two connections – c1 and c2. Informally, the algorithm iterates through the connections under system and replaces each connection that has associated error model with two new connections and an intervening connection component called CC. The algorithm assumes that the components and connections in a system are available from a variable sys.

3.4 Model transformation design

Due to the fact that both the AADL and HiP-HOPS modelling languages are evolving, it is important that the design of the model transformation should have the capability to adapt easily to changes in the AADL or HiP-HOPS languages. Transformations should also be adaptable to changes in the purpose and requirements of the transformation. Reusability and adaptability are the quality properties in the transformation method developed in this research. From the transformation experience gained in this research, this chapter discusses model transformation

design for reusability and adaptability. The rule transformation language ATL is used to illustrate the transformation definitions.

3.4.1 Modularising transformation definition

This section first considers how the structure of the source and target models affects the modularisation of transformation definitions and how the evolution of these models may affect the transformation. Secondly, there are different integration mechanisms such as explicit rule calls and implicit rule calls that may help ensure the required reusability and adaptability. Each mechanism has advantages and disadvantages with respect to modularisation which is discussed.

To show the advantages and disadvantages of the different dimensions in the decomposition of metamodels, a set of transformation rules are defined. The transformation rules are shown conceptually as relations between the source and target metamodels. Based on these rules, the function units required to be modularised for reusability and adaptability are identified. Then the section introduces a scenario in which the target metamodel is evolved by adding an alternative element to a component. The chapter then considers three rule integration mechanisms and evaluates their quality with respect to reusability and adaptability. The degree of coupling between rules, as discussed in (Kurtive et al., 2007), is considered.

In the context of this research, the purpose of the transformation is to transform an AADL dependability model to a HiP-HOPS dependability model. There are similarities and differences in the AADL and HiP-HOPS modelling concepts. Both languages use the concepts of component, port and connection although detailed semantics differ. For example, for dependability modelling, AADL is based on error state machine and HiP-HOPS is based on fault tree. Thus, it is necessary to identify a mapping of concepts (rules) from source to target model.

3.4.1.1 Components

To identify the transformation rules, the required concepts, and how they are implemented in the target metamodels, can be used as the starting point. In the context of this research, for example, a target HiP-HOPS component must be generated from a corresponding AADL component. The AADL metamodel describes ten component types that can be mapped to a HiP-HOPS component. This includes five software components: data, process, subprogram, thread and thread group; four hardware components: bus, device, memory and processor; and one composite component: system. In HiP-HOPS, the component is abstract. This means that HiP-HOPS component can be any one of those ten AADL component type. In order to build a HiP-

HOPS component, ten transformation rules may be used, one for each of the AADL component type. This draws forth a possible decomposition of rules based on the AADL component type. Since the decomposition start from the source (AADL) model, this method can be seen as a source-driven modularisation. The transformation rules shown in outline are the following:

```
processToComponent ( source [ProcessType],
    target [Ports, FailureData] )
…
deviceToComponent ( source [DeviceType],
    target [Ports, FailureData] )
…
systemToComponent ( source [SystemType],
    target [Ports, FailureData] )
```

Three rules are listed above: for process (`processToComponent`), device (`deviceToComponent`) and system (`systemToComponent`) types. The remaining rules are omitted to avoid repetition since they all have the same structure. Each rule consists of a source pattern, i.e. a tuple containing AADL model elements, and a target pattern, i.e. a tuple containing HiP-HOPS model elements. Each rule means that for each occurrence of the source tuple (elements) in an input model, the target tuple (elements) are created in an output model. This basic interpretation captures the essential semantics and is sufficient for the discussion of the rules.

One of the problems with the above ten rules is that all the ten rules repeat the same structure. The construction of target (HiP-HOPS) component i.e. `target [Ports, FailureData]`, is repeated ten times rather than modularised in a single reusable unit. Potentially, the replication would be as many times as the number of the source (AADL) component types. Thus, potential changes in the future for this element are error prone since the replication. For example, assume that the target model (HiP-HOPS) evolves to have the capability of optimising the architecture of dependable systems. For the optimisation purpose, each component needs one or more alternatives in order to create the search space. This alternatives attribute is common to all the ten component types. To implement this change of the component, it results in a series of changes in all the rules. For example, in the rule of `processToComponent`, an `Alternative` element needs to be added to the `target` model below.

```
processToComponent ( source [ProcessType],
    target [Ports, FailureData, Alternative] )
```

This means each rule needs to add an alternative element as in the example above (i.e. in total need to add ten times) in order to transform AADL alternative components to corresponding

HiP-HOPS alternative components. There is the danger that the person making the changes to the rules does not realise that there are ten rules that need to be modified and some of the rules are overlooked. It is easier to modify one rule rather than many rules.

Furthermore, consider that if AADL evolves in the future and a new component type is added then it is necessary to add an additional corresponding rule for the transformation of this component type. It needs to add potentially as many times as the number of the new component types. This reduces the adaptability. A well-designed transformation definition should have the resilience to adapt easily to this kind of evolution.

The transformation in the context of this research is not dependent on the component type because error models are not dependent on the component type. For these reasons, the similar parts of the rule should be abstracted. An abstract or super class component type could be used in the source so as to transform all AADL component types to HiP-HOPS component using one rule. Whenever there is a need for adding new component types there is no need to change the transformation definition since the super class abstracts all the component types. The above rules can then be replaced by one rule in order to improve the reusability and adaptability. The benefit is that if it is necessary to change the mapping of an AADL component to a HiP-HOPS component then the change is restricted to a single rule. This is better than changing many rules. The new version of the conversion is defined as follows:

```
componentTypesToComponent ( source [ComponentType],
    target [Ports, FailureData] )
```

All the ten component types shown above are the sub-class of `ComponentType` class. The above general rule will match any AADL component type and transform them to HiP-HOPS component. In fact this rule may match more than the ten component types listed.

The issue of whether to have a rule for each type of component depends on how components should be transformed. Due to the fact that HiP-HOPS does not distinguish component types as they are distinguished in AADL. HiP-HOPS abstracts component as a conceptual black-box with ports and failure data and hence the transformation in the context of this research is not dependent on the component type. For other purposes of model transformation, however, it is may be better to define the transformation depend on the component type. Assume that each component type in source model is also distinguished in the target model. In this case it would be better to define the transformation with rules that are dependent on the source and target component type. If there are any changes or evolutions for any component type the fact that there is a specific rule for that component type makes the rules more modifiable to the changes.

In HiP-HOPS, the System element is the top-most part of the actual system hierarchy, which contains a set of Components. Hierarchies are created by enabling Components to contain Systems of their own. This means that the implementation of a component can be a sub-system with sub-components. In HiP-HOPS, the elements for a component with or without subsystem are different and thus the transformation should be differentiated. The new version of the conversion is defined as follows:

```
componentTypeIsSubsystemToComponent (
    source [ComponentType (isSystemType)],
    target [Ports, FailureData, system] )

componentTypeToComponent (
   source [ComponentType (notSystemType)],
    target [Ports, FailureData] )
```

Note that in the target model, a `system` element is created if the source component is a type of sub-system. Generally, there will be guard conditions (`isSystemType`, `notSystemType`) in the rules to distinguish transformed source types. Now the transformation definition is better since the rules are modularised in two single reusable and adaptable units but there are still the problems of mixing of functionality.

3.4.1.2 Component ports and failure data

A general principle of modularity is that different functions should be implemented in different modules. A problem with the above designs is that it involves mixing of functionality within a single module. In the target model, the two rules above both refer to the elements for the `ports` and `FailureData`. Therefore, it contains concepts that belong to two different concerns, i.e. ports and failure data. Ports are used to describe the input and output points for components and as such are relevant to both normal behaviour of system and failure behaviour of the system. Failure data is used to describe only the error behaviour of the system.

Mixing of functionality within a single module can cause problems if the application models change. For example, suppose that HiP-HOPS changes its failure rate modelling concept by adding a failure rate type (e.g., constant, Poisson distribution) element to explicitly calculate the occurrence probability of an error event. This kind of change effects the transformation of the AADL error model to HiP-HOPS failure data but is irrelevant to ports. Isolating the rules that require changes is more difficult if failure data parts of rules are obscured by other concepts such as ports. There is the danger that the person making the changes to the rules does not realise that these rules also contain ports and some modification to the transformation of ports could be made by mistake. Moreover, the person making the changes to the rule may feel

confused since he believes only failure data part of the transformation needs to be changed but the mix of functionality (ports and failure data) exists within a single rule and this may suggest in his mind that failure date changes affects ports.

The reason for the problems discussed above is because only one dimension for decomposition into modules is considered. For the ten rules design, the source component type was used as the dimension for decomposition into modules. The way the source AADL model is decomposed into different component types means that each AADL component type is transformed into all the elements (such as ports and failure data) needed to construct a HiP-HOPS component. This particular dimension of decomposition leads to a good modularisation provided the future changes are restricted to just the AADL component types. If future changes involve other AADL elements or HiP-HOPS elements, as discussed above, it causes the defined rules to be less reusable, adaptable and modifiable.

Better than decomposition in one dimension is decomposition in multiple dimensions. Decomposition in multiple dimensions, as discussed in Kurtive et al. (2007), ensures reusable and evolvable transformation design along those dimensions. A multiple dimensions of decomposition could be classifying different functionalities of the component based on target model, where the ports and failure data are classified as different dimensions. Hence two extra rules are extracted for each of the dimension:

```
componentTypeToPort (source [ComponentType],
    target [Ports] )

componentTypeToFailureData (source [ComponentType],
    target [FailureData] )
```

This design of rule definition eliminates the mixing of functionality problem discussed above. The source element of the rules is defined along differentiated component type dimension. For each distinguished component type, the target elements of rules are decomposed into other two dimensions: `Ports`, and `FailureData`. The advantages of this design are as follows: First, all dimensions defined in the target model are adaptable. Second, the common parts (dimensions) in the rules are extracted into separate module. Extraction can lead to more modifiable and reusable modules (i.e. rules).

3.4.1.3 Component connections, guards and error renaming

The important dimension of model decomposition that can be used for modularity is the decomposition of a model into components and connections. Components and their associated

error models in a system are connected to each other through connections. A connection declares interactions e.g. data or flow of control between components through component ports. Connections form the main paths of error propagations between component error models. The interactions between component error models occur when error propagations occur through the connections between components.

The decomposition of a model into components and connections can be refined further into the decomposition of components, connections and port descriptions as described in guard_in(out) specifications. The guards specified in component's ports are used to filter and rename incoming and outgoing error propagations coming into or going out a component port. For example, the guard_in error property specifies incoming error propagations (through an input port) to be either unchanged, or masked or mapped to a different kind of error declared in the local component error model. The guard_out error property specifies similar guard for mapping or masking outgoing error propagations (through an output port) of a component.

The guards applied to components ports combine the local component errors and external errors together so as to compose a complete system error model. Thus, in addition to transitions of component (and associated error models) and connections (error propagation paths), the guards property must be also transformed so as to generate a complete system fault tree.

It is expected that component based models, even if not described in AADL, will have the three dimensions of components, connections and port descriptions. For example, in EAST-ADL, the FaultFailurePropagationLinks defined in EAST-ADL ErrorModel (EAST-ADL Association, 2013 (b)) are used to specify error propagation paths in the ErrorModelType. The FailureOutPort is used to specify a point for propagating outgoing failure. Similarly, the FaultInPort is used to define a point for receiving incoming faults. Guards (Chen et al., 2013; EAST-ADL Association, 2013 (b)) are specified in EAST-ADL to declare cause-effect dependencies of error state transitions. The state transition guards can be used to precisely specify under which condition (e.g. Conditon[VehicleSpeed > ABSVehicleSpeedThreshhold]) a transition will be triggered from one state (the associated "from" state) to another state (the associated "to" sate). This means that a transformation based on the three dimensions should be generally applicable.

The design experience discussed in this section is specific to AADL and ATL. The decomposition of the transformation discussed in the context of this research is along the dimensions of component, connections and boundaries of components i.e. guards on the ports of components. This is because these dimensions are likely to be contained in most of model-based modelling languages e.g. AADL and EAST-ADL.

3.4.2 Rule integration mechanisms

Until now this chapter has explored how the decomposition in the existing metamodels can influence the design of transformation rules. Next, it focuses on another aspect of transformation which is the modularity of the transformation rule-language. There are different modularity constructs available in rule-based transformation languages that may help ensure the required reusability and adaptability. Rule integration is an example of a modularity construct. Rule integration mechanisms include implicit or explicit rule calls and rule inheritance. Each mechanism has advantages and disadvantages with respect to modularisation which is discussed. The rule integration mechanism allows one rule to use the functionality of another rule. The rest of this section focuses on the implication of three rule integration mechanisms, i.e., explicit rule calls, implicit rule calls and rule inheritance for modularity.

Explicit rule calls as described in Kurtive et al. (2007), are usually used in imperative and hybrid languages. In this mechanism, rules are integrated with explicit rule scheduling of function and method calls. ATL supports explicit rule calls, where a rule may be directly invoked by another rule.

Implicit rule calls mechanism is usually found in declarative languages. It relies on indirect rule dependencies. Rules are enabled whenever certain conditions are satisfied. A scheduler schedules all the enabled rules and decides which rule is executed. Rules may create the conditions that enable other rules and so trigger the execution of other rules do not have to use explicit rule names. Note that the triggered rules may not be immediately executed. It depends on the transformation scheduling algorithm in a language. The rule might be performed later or the result could already have been generated.

Rule inheritance mechanism enables one rule to inherit functionalities from another rule. This allows one rule can be reused for many times. For example, if rule `R1` is anyone over 16 must join the army and rule `R2`, which inherits from `R1`, is anyone over 18 must vote then everyone over 18 must join the army and vote.

```
abstract rule R1 {
   from Over16 to JoinArmy
}
 rule R2 extends R1 {
   from Over18 to Vote
}
```

The semantics of `R2` is as follows: `Over18` must be a subset of `Over16` this means that the set of models which match `Over18` must also match `Over16`. Also the target of R2 is `JoinArmy`

union Vote. The result is that a match of Over16 leads to JoinArmy and a match of Over18 leads to JoinArmy and Vote. To understand rule inheritance, consider for example, if one defines rule R2 without inheriting from R1, one must define R2 as

```
rule R2 {
   from Over18 to [JoinArmy, Vote]
}
```

in order to ensure those Over18 must JoinArmy and Vote. This reduces the reusability of the rules since the JoinArmy appears twice (both in R1 and R2).

In either situation (with or without inheritance) one needs to ensure that rule R1 and R2 only matches purely Over16 and Over18 elements (types) respectively otherwise it will raise a multiple matching problem. Those people Over18 will match both rules (R1 and R2) and this is not allowed in ATL since in ATL a source model element must not be matched more than once (ATL/User Guide, 2012). When R2 inherits from R1, R1 is matched to those elements that match R1 and not R2. R2 is matched to the remaining elements.

In order to explore how each rule integration mechanism can impact the reusability and adaptability of defined rules, some scenarios are considered. Consider that the target model (HiP-HOPS) evolves in adding an alternative element to each component so as to extend the target model to have the capability for system optimisation analysis. The target metamodel is extended. The original defined model transformation rules should be modified to match this evolution so that the source model (AADL) can utilise the newly added optimisation analysis. There will be several ways (e.g. explicit rule calls, implicit rule calls and rule inheritance) to implement this change in the transformation definition. The designers must decide which rule integration mechanism that can ensure the best quality design.

First, consider using the explicit rule calls to implement the scenario (add new alternative element to each component type). Before the change, the rule would look like:

```
rule componentTypeIsSubsystemToComponent{
    from e : ComponentType (isSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
        system <- [vSystem] e
    )}
```

The assignment of alternative element could be modularised in an explicit rule call. The code below shows the implementation based on explicit rule calls. The expression [vPort] e

means the required target object generated from the source element `e` is returned and assigned to the identifier `vPort`.

```
rule componentTypeIsSubsystemToComponent{
    from e : ComponentType (isSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
        system <- [vSystem] e
    )
    do {
        SetAlternative(e, vComponent );   --Explicit rule call
    }
}
```

The rule that is called is

```
rule SetAlternative (source: ComponentType, target: Component){
    target.alternative <- [vAlternative] source;
}
```

This modification has to be done for all the rules and so there is

```
rule componentTypeToComponent{
    from e : ComponentType (notSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
    )
    do {
        SetAlternative(e, vComponent );   --Explicit rule call
    }
}
```

The generation of the `alternative` element is defined in a single rule. In this way, the rule is reusable because one rule is reused many times. The implementation, however is unnatural since a `do{}` statement may not be needed for adding the `alternative` element.

The modification described above could have been done with an implicit rule call. It would be implemented as

```
rule componentTypeIsSubsystemToComponent{
    from e : ComponentType (isSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
        system <- [vSystem] e
        alternative <- [vAlternative] e
    )
}

rule componentTypeToComponent{
    from e : ComponentType (notSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
        alternative <- [vAlternative] e
    )
}

rule alternativeClassifierToAlternative{
    from e : ComponentClassifier
    to vAlternative : Alternative(
        cost <- e.cost
        fData <- [vErrorModel] e
    )
}
```

The expression `[vAlternative] e` means the required `alternative` object generated from the source element `e` is returned and assigned to the identifier `vAlternative`. Note that the rules that create this alternative object (`alternative <- [vAlternative] e`) is implicitly referenced. The code `alternative <- [vAlternative] e` may use any rule that generates an object defined with the identifier `vAlternative` and thus lead to a low coupling between rules. In this example (implicit rule calls), the adaptability and reusability requirements of rules are achieved. Furthermore, implicit rule calls enables more modifiable rule definition. As discussed above, if an `alternative` element is added to a component then only one new rule `alternativeClassifierToAlternative` is added. If any changes related to alternative element are necessary in the future only this new rule needs to be changed.

In the alternatives component problem, the implicit rule call implementation is more natural than the explicit rule call. An `alternative` element is directly added in implicit rule call rather than invoke a `do{}` statement in the explicit rule call. In some cases, however, an explicit rule call has to be used. For example, when creating the HiP-HOPS `fData` element it is necessary to transform the AADL error state machine to HiP-HOPS fault tree (failure expression). In this situation, a complicated algorithm or function is needed to transform multiple elements of different types to a single one. A function (or helper in ATL context) in a

specific transformation language may not capable to support this kind of transformation. Hence, a further transformation with a capable algorithm might be needed to explicitly create target model element (i.e. `failureExpression`) from imperative code. Since the AADL error state machine must be transformed to fault tree for all the component types, the algorithm should be modularised in a single rule in order to be reused. The explicit rule call is used to invoke the outside function in order to create the failure expression element. For example,

```
rule componentTypeIsSubsystemToComponent{
    from e : ComponentType (isSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
        system <- [vSystem] e
    )
    do {
        GetFailureExpression(e, vComponent);   --Explicit rule call
    }
}

rule componentTypeToComponent{
    from e : ComponentType (notSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
    )
    do {
        GetFailureExpression(e, vComponent);   --Explicit rule call
    }
}

rule GetFailureExpression (source: ComponentType, target: Component){
    target.fData.failureExpression <- source.getFailureExpression();
}
```

In the case of explicit rule calling, however, if a new rule for failure expression transformation is to be used the invoking code must be changed to reflect the name of the new rule.

Consider now rule inheritance.

Before the change, the two rules are defined as:

```
rule componentTypeIsSubsystemToComponent{
    from e : ComponentType (isSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
        system <- [vSystem] e
    )
}

rule componentTypeToComponent{
    from e : ComponentType (notSystemType)
    to vComponent : Component(
        ports <- [vPort] e
        fData <- [vErrorModel] e
    )
}
```

The two rules are extended by adding the `alternative` element:

```
rule AlternativeForSubsystem extends
componentTypeIsSubsystemToComponent{
    from e : ComponentType (isSystemType and hasAlternatives)
    to vAlternative : Alternative(
        alternative <- e.alternative
    )
}

rule AlternativeForComponent extends componentTypeToComponent{
    from e : ComponentType (notSystemType and hasAlternatives)
    to vAlternative: Alternative (
        alternative <- e.alternative
    )
}
```

This implementation defines the rules `AlternativeForSubsystem` and `AlternativeForComponent` that extend two already existing rules by adding an alternative element. In this mechanism, the adaptability requirement is not satisfied. For example, suppose that a new rule shall be used for transforming the AADL component types to HiP-HOPS component. This implies that both rules `AlternativeForSubsystem` and `AlternativeForComponent` should inherit from this new rule. It is however, not allowed in the rule inheritance mechanism because one rule cannot inherit from more than one rule. This implementation also leads to tight coupling between rules because it refers to rules by name. Furthermore, the construction of alternative element is repeated twice rather than modularised in a single reusable unit.

In the context of this research, rules are integrated by using implicit rule calls so as to allow one rule to use the functionality of another rule. Explicit rule calls are also used as a combination of implicit rule calls in order to transform state machine to failure expression as discussed.

3.5 Summary

This Chapter describes a model transformation method which transforms AADL dependability models to HiP-HOPS fault tree models. This allows HiP-HOPS to perform dependability analysis on systems modelled in AADL. Transformation allows the exploitation of existing non-AADL-based methods and tools, e.g. HiP-HOPS.

The transformation algorithm developed in this research is adapted from the state machine to Boolean failure expression algorithm shown in Mahmud et al., (2010, 2011). The algorithms developed produce HiP-HOPS specific failure expressions (i.e. the notation <FailureClass>-<PortName>) from AADL state machines. These include algorithms for mapping error states and events to component ports, transforming guard_in(out) expressions to fault trees, and transforming AADL connections error model to HiP-HOPS fault trees.

The transformation algorithms were conveniently implemented using the ATLAS Transformation language. This chapter shows the feasibility of the automation by implementing the tool AADL2HiP-HOPS, which has been developed to implement the model transformation and has been integrated as a plug-in into the AADL development environment OSATE. AADL is used as the notation for capturing the system architecture model and the AADL Error Model Annex is used to capture the component faults and failure modes. The plug-in (AADL2HiP-HOPS) transforms this AADL dependability model to a HiP-HOPS model which is then used for synthesis of fault trees, FMEAs and other analyses to automatically generate the fault tree and FMEA table of system for further dependability analysis.

The benefit of this transformation method is that it opens a path that will enable the AADL language to take advantage of some of the unique capabilities of HiP-HOPS which include the synthesis of multiple failure mode FMEAs, temporal fault tree analysis and evolutionary architecture optimisation with respect to dependability and cost.

This chapter also describes some of the modularisation techniques in ATL. Transformation modules (rules) can be obtained from the correspondences between the source and target model elements. Generally, there are multiple correspondences existing between the source and target elements and thus designers should assess them with the required quality attributes e.g. reusability, adaptability and modifiability. This chapter shows that different sets of transformation rules can be derived from different decompositions in the source and target

metamodels. Thus, the designers are suggested to consider multiple decompositions in the metamodels.

The chapter introduces three rule integration mechanisms, i.e. implicit rule calls, explicit rule calls, and rule inheritance that can be used to integrate ATL rules together. Each mechanism has its advantages and disadvantages to integrate rules together. Implicit rule calls relies on indirect rule dependencies and usually lead to a low coupling between rules. It is thus could be chosen when the adaptability and reusability are set as the main quality attribute of the transformation definitions. In the context of this research, explicit rule calls is best for error state machine to fault tree conversion. This is because ATL rules take input from sources of different types i.e. they are source type sensitive. The error state machine to fault tree conversion is not source type sensitive. A transformation algorithm is needed to explicitly generate target model element (i.e. failureExpression) from imperative code.

Chapter 4 Case study: temperature monitoring system

To illustrate the useful application of the transformation method, this chapter demonstrates the transformation using a case study. In Mian et al., 2012, it illustrated the model transformation technique to a small standby-recovery system. In that small system, however, only some of the transformation rules are applicable because, for example, the fan-in connection pattern does not occur. In this Chapter, a more complex model is used, which has a multi-level hierarchical structure and contains the full range of connection patterns. With this model the case study can illustrate the transformation of:

(a) a system which includes sub-systems,

(b) transform AADL fan-in connections,

(c) transform the guard_in property into HiP-HOPS failure expression.

4.1 The temperature monitoring system

Figure 4.1 illustrates a system for monitoring a temperature controlled storage facility, an example of which is a cold store for agricultural products. The system maintains a record of the temperatures within various parts of the facility but also raises alarms when the temperatures are outside prescribed limits. At the top level (see Figure 4.1), the monitoring system consists of two subsystems. One subsystem (`SensorInput`) consists of a variety of temperature sensors and switches, the second subsystem (`TempProcess`) processes the various signals from these sensors into alarm outputs or other outputs suitable for further processing, e.g. data logging. There are two types of temperature device. One type, sensor, provides a continuous temperature reading which is used for record keeping and for average temperature calculations. Another type, switch, provides a signal when the temperature exceeds or falls below a pre-set value. The second subsystem (`TempProcess`) checks that the temperature requirements are met, which it does this by comparing current and average temperatures with pre-set values, and logging temperatures values.

In more detail, in `SensorInput (see` Figure 4.2), `SensorInput_D1`, `SensorInput_D2` and `SensorInput_D3`, are single temperature switches which provide a signal if a pre-set temperature is exceeded. These three switches are located in three different places. The outputs of these switches are input to the process `SensorInput_P1`. If any one of the switches fails then the `SensorInput_P1` process fails. `SensorInput_D4` and `SensorInput_D5`, are double temperature switches which provide a signal if the temperature moves outside a region defined by a lower and upper limit. These two switches send data to the process

SensorInput_P2. Each of switches can act as a replacement for the other. SensorInput_D6 is a temperature sensor connected to process SensorInput_P3.

The subsystem TempLimLog, receives and analyses the data from subsystem SensorInput. Temperatures that exceed limits are, for security, simultaneously logged in three different locations (i.e., process TempLimLog_P1, TempLimLog_P2, and TempLimLog_P3). Process TempLimLog_P4 is responsible for detecting alarm conditions.

System TempProcess analyses the temperature data received and transmits the temperature data for further analysis. More specifically, process TempProcess_P1 and TempProcess_P2 receive the continuous temperature data and produce a short-term (over last 5 minutes) and a long-term (over last 90 minutes) average temperature respectively. TempProcess_P3 transmits the temperature alarms and average readings to other systems.

The thesis uses a hierarchical naming strategy, i.e., system name followed by component name and then the port names. For example, SensorInput_D1_Out (shown in Figure 4.2) is the port name, where SensorInput is the system name, D1 is device one in this system, Out is the direction of this port.

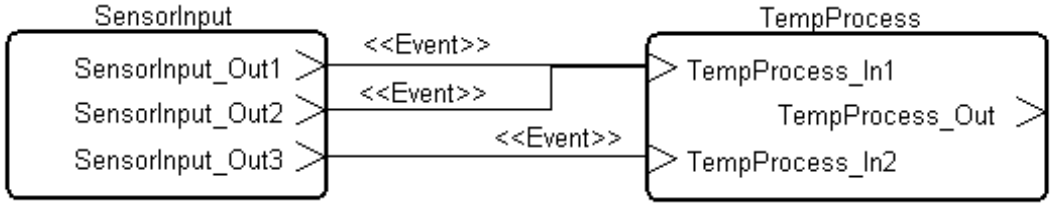

Figure 4.1 The top level structure of the temperature monitoring system

Figure 4.2 to Figure 4.4 illustrates the architecture of the sub-systems SensorInput, TempProcess and TempLimLog. Note that there are system levels where two or more Lines share the same destination port. For example, in Figure 4.1, for input port TempProcess_In1 on system TempProcess, there are two Lines which share this destination port. One comes from SensorInput_Out1 and the other comes from SensorInput_Out2.

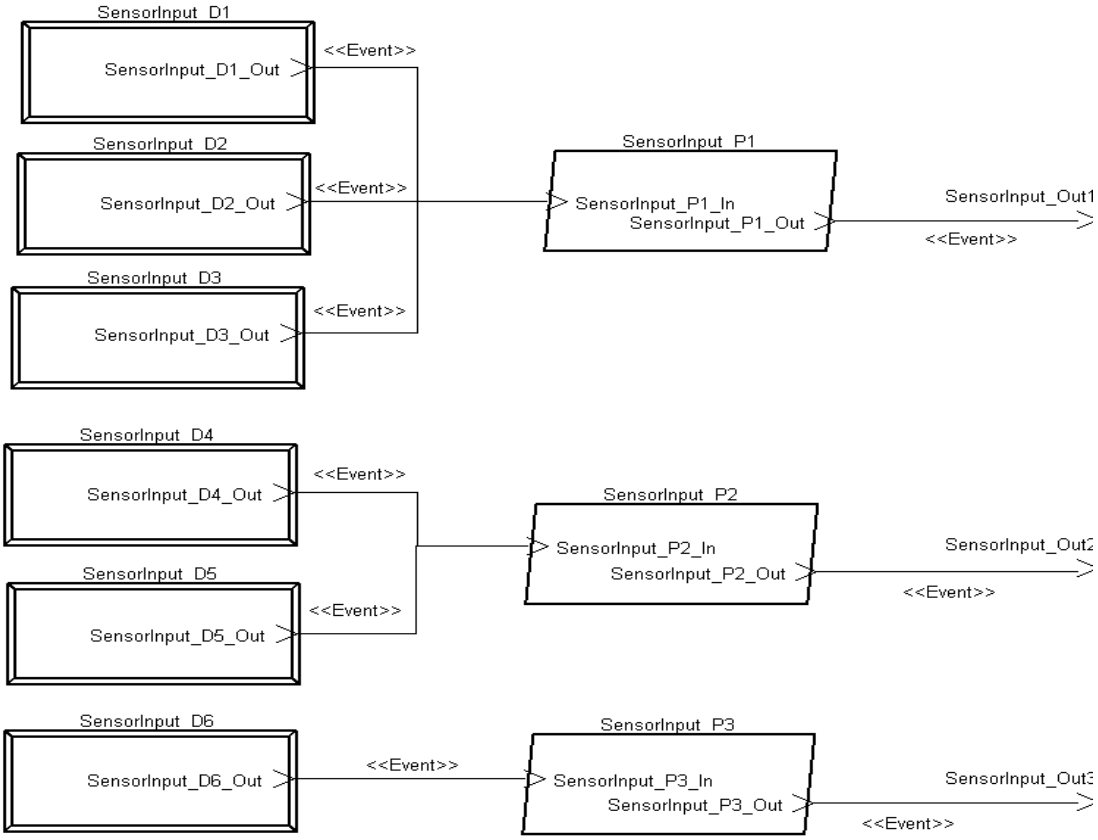

Figure 4.2 The architecture of sub-system `SensorInput`

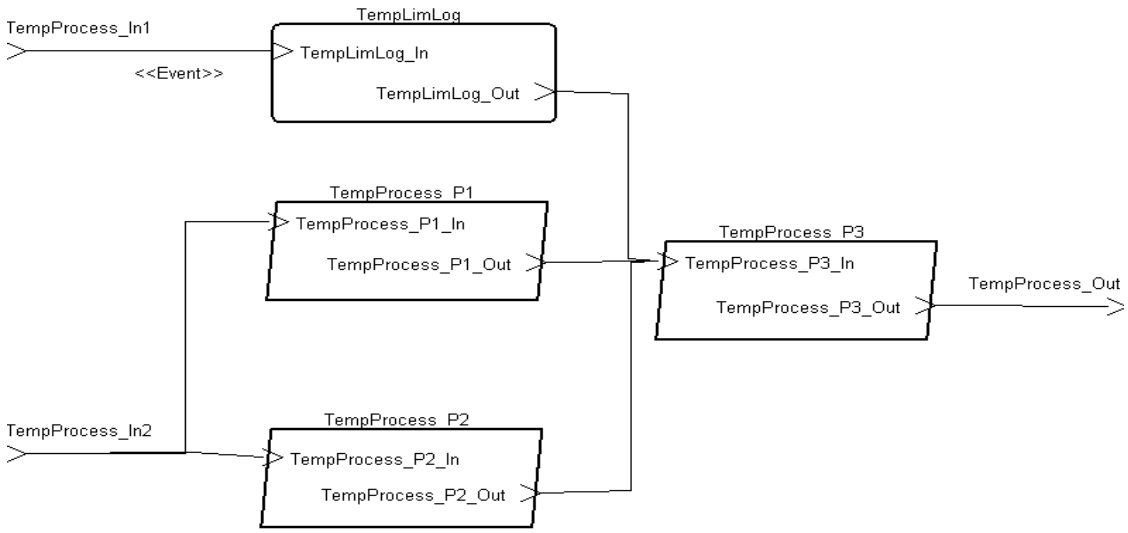

Figure 4.3 The architecture of sub-system `TempProcess`

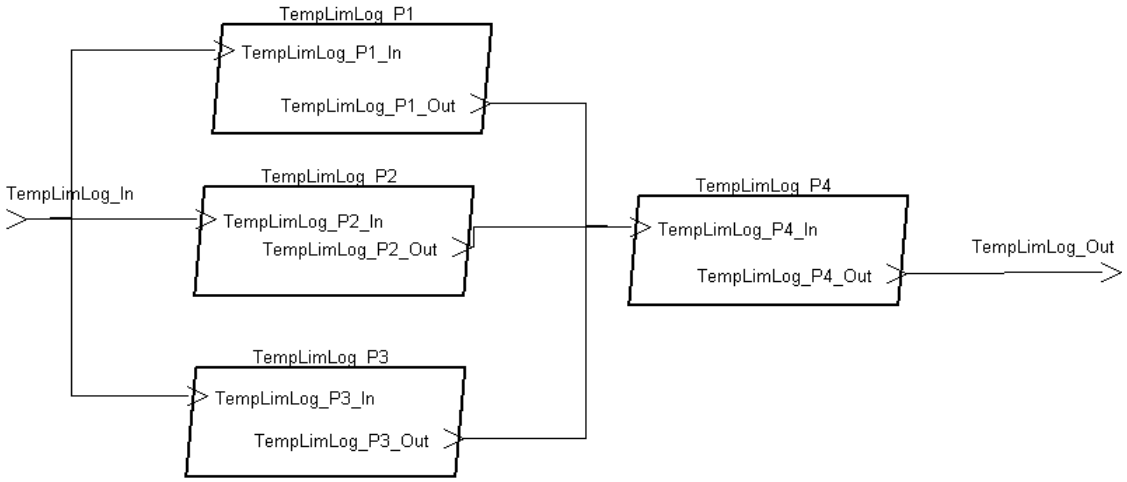

Figure 4.4 The architecture of sub-system `TempLimLog`

In this system, each component has a failure mode "Fail" which represents an internal failure of the component. A component failure causes the omission of output from that component. In addition, for each component, an omission of input will cause an omission of output.

4.2 AADL modelling of temperature monitoring system

A partial AADL description for the temperature monitoring system is shown in Figure 4.5 to Figure 4.7. Note that in Figure 4.5, for the error model associated with system implementation, `TempProcess.Impl`, there is a guard_in error property defined on port `TempProcess_In1`. The guard_in property specifies how an input error is propagated to a port of a component based on the error state of that component and the error states or output error propagations of connected components. Here, the guard_in expression specifies that the omission of input `Omission_TempProcess_In1` is caused by the omission of output `Omission_SensorInput_Out1` and the `Omission_SensorInput_Out2` from the output ports of `SensorInput`. The other guard_in error properties in this system are shown in Figure 4.6.

Figure 4.7 shows AADL error model type and the implementation declaration for the system `TempProcess`, device `SensorInput_D1` and process `SensorInput_P1`. In Figure 4.7, for system `TempProcess`, it is specified that there is an `EMI` fail for all the components within the `TempProcess` system.

There are also two types of error model implementations. Type one (see `type1` in Figure 4.7) indicates that the component may fail as the result of an internal failure. Type two (see `type2`

in Figure 4.7) indicates that the component may fail as the result of an internal failure or if there is an omission of input to the component.

```
system implementation TempControl.Impl
  subcomponents
    SensorInput: system SensorInput.Impl;
    TempProcess: system TempProcess.Impl;
  connections
    EventConnection1: event port SensorInput.SensorInput_Out1 -> TempProcess.TempProcess_In1;
    EventConnection2: event port SensorInput.SensorInput_Out2 -> TempProcess.TempProcess_In1;
    EventConnection3: event port SensorInput.SensorInput_Out3 -> TempProcess.TempProcess_In2;
end TempControl.Impl;

system implementation SensorInput.Impl
  subcomponents
    SensorInput_D1: device SensorInput_D1.Impl;
    SensorInput_D2: device SensorInput_D2.Impl;
    SensorInput_D3: device SensorInput_D3.Impl;
    SensorInput_P1: process SensorInput_P1.Impl;
    SensorInput_D4: device SensorInput_D4.Impl;
    SensorInput_D5: device SensorInput_D5.Impl;
    SensorInput_D6: device SensorInput_D6.Impl;
    SensorInput_P2: process SensorInput_P2.Impl;
    SensorInput_P3: process SensorInput_P3.Impl;
  connections
    EventConnection1: event port SensorInput_D1.SensorInput_D1_Out -> SensorInput_P1.SensorInput_P1_In;
    EventConnection2: event port SensorInput_D2.SensorInput_D2_Out -> SensorInput_P1.SensorInput_P1_In;
    EventConnection3: event port SensorInput_D3.SensorInput_D3_Out -> SensorInput_P1.SensorInput_P1_In;
    EventConnection4: event port SensorInput_D4.SensorInput_D4_Out -> SensorInput_P2.SensorInput_P2_In;
    EventConnection5: event port SensorInput_D5.SensorInput_D5_Out -> SensorInput_P2.SensorInput_P2_In;
    EventConnection6: event port SensorInput_D6.SensorInput_D6_Out -> SensorInput_P3.SensorInput_P3_In;
    EventConnection7: event port SensorInput_P1.SensorInput_P1_Out -> SensorInput_Out1;
    EventConnection8: event port SensorInput_P2.SensorInput_P2_Out -> SensorInput_Out2;
     EventConnection9: event port SensorInput_P3.SensorInput_P3_Out -> SensorInput_Out3;
  annex Error_Model {**
    model => ErrorModel_TempControl_System::Basic.SensorInput;
**};

end SensorInput.Impl;

system implementation TempProcess.Impl
  subcomponents
    TempLimLog: system TempLimLog.Impl;
    TempProcess_P1: process TempProcess_P1.Impl;
    TempProcess_P2: process TempProcess_P2.Impl;
    TempProcess_P3: process TempProcess_P3.Impl;
  connections
    EventConnection1: event port TempProcess_In1 -> TempLimLog.TempLimLog_In;
    EventConnection2: event port TempProcess_In2 -> TempProcess_P2.TempProcess_P2_In;
    EventConnection3: event port TempProcess_P2.TempProcess_P2_Out -> TempProcess_P3.TempProcess_P3_In;
    EventConnection4: event port TempProcess_P1.TempProcess_P1_Out -> TempProcess_P3.TempProcess_P3_In;
    EventConnection5: event port TempProcess_P3.TempProcess_P3_Out -> TempProcess_Out;
    EventConnection6: event port TempLimLog.TempLimLog_Out -> TempProcess_P3.TempProcess_P3_In;
    EventConnection7: event port TempProcess_In2 -> TempProcess_P1.TempProcess_P1_In;
  annex Error_Model {**
    model => ErrorModel_TempControl_System::Basic.TempProcess;
    report => Omission_TempProcess_Out;
    guard_in => Omission_TempProcess_In1
          when TempProcess_In1[Omission_SensorInput_Out1, Omission_SensorInput_Out2],
                mask when others
    applies to TempProcess_In1;
**};
end TempProcess.Impl;
```

Figure 4.5 Partial AADL architecture for temperature monitoring system

```
device implementation SensorInput_D1.Impl
  annex Error_Model {**
    model => ErrorModel_TempControl_System::Basic.SensorInput_D1;
**};
…
process implementation SensorInput_P1.Impl
  annex Error_Model {**
   model => ErrorModel_TempControl_System::Basic.SensorInput_P1;
    report => Omission_SensorInput_P1_Out;
    guard_in => Omission_SensorInput_P1_In
      when SensorInput_P1_In[Omission_SensorInput_D1_Out,
                             Omission_SensorInput_D2_Out,
                             Omission_SensorInput_D3_Out],
                mask when others
    applies to SensorInput_P1_In;
**};
end SensorInput_P1.Impl;
….
process implementation TempProcess_P3.Impl
  annex Error_Model {**
    model => ErrorModel_TempControl_System::Basic.TempProcess_P3;
    guard_in => Omission_TempProcess_P3_In
      when TempProcess_P3_In[Omission_TempProcess_P1_Out]
        OR TempProcess_P3_In[Omission_TempProcess_P2_Out]
        OR TempProcess_P3_In[Omission_TempLimLog_Out],
                mask when others
    applies to TempProcess_P3_In;
**};
end TempProcess_P3.Impl;
…
process implementation TempLimLog_P4.Impl
  annex Error_Model {**
    model => ErrorModel_TempControl_System::Basic.TempLimLog_P4;
    guard_in =>  Omission_TempLimLog_P4_In
      when TempLimLog_P4_In[Omission_TempLimLog_P1_Out,
                            Omission_TempLimLog_P2_Out,
                            Omission_TempLimLog_P3_Out],
                 mask when others
    applies to TempLimLog_P4_In;
**};
end TempLimLog_P4.Impl;
```

Figure 4.6 Partial AADL implementations associated with error models

```
package ErrorModel_TempControl_System
public
  annex Error_Model {**
    error model Basic --basic error model
    features
      ErrorFree: initial error state;
      Fail, Fail1, Fail2, Fail3, EMI: error event;
      Omission_SensorInput_D1_Out, Omission_SensorInput_D2_Out, Omission_SensorInput_D3_Out,
      Omission_SensorInput_D4_Out, Omission_SensorInput_D5_Out, Omission_SensorInput_D6_Out,
      Omission_SensorInput_P1_Out, Omission_SensorInput_P2_Out, Omission_SensorInput_P3_Out,
      Omission_SensorInput_Out1, Omission_SensorInput_Out2, Omission_TempProcess_P1_Out,
      Omission_TempProcess_P2_Out, Omission_TempProcess_P3_Out, Omission_TempProcess_Out,
      Omission_TempLimLog_P1_Out, Omission_TempLimLog_P2_Out, Omission_TempLimLog_P3_Out,
      Omission_TempLimLog_P4_Out, Omission_TempLimLog_Out:
          error state;
      Loss_Data, Loss_Data1, Loss_Data2, Loss_Data3: out error propagation {occurrence => fixed 0.8};
      Omission_SensorInput_P1_In, Omission_SensorInput_P2_In, Omission_SensorInput_P3_In,
      Omission_TempProcess_P1_In, Omission_TempProcess_P2_In, Omission_TempProcess_P3_In,
      Omission_TempProcess_In1, Omission_TempLimLog_P1_In, Omission_TempLimLog_P2_In,
      Omission_TempLimLog_P3_In, Omission_TempLimLog_P4_In:
          in error propagation;
    end Basic;

   error model implementation Basic.SensorInput
   transitions
      ErrorFree -[ Fail1 ]-> Omission_SensorInput_Out1;
      Omission_SensorInput_Out1 -[ out Loss_Data1 ]-> Omission_SensorInput_Out1;
      ErrorFree -[ Fail2 ]-> Omission_SensorInput_Out2;
      Omission_SensorInput_Out2 -[ out Loss_Data2 ]-> Omission_SensorInput_Out2;
      ErrorFree -[ Fail3 ]-> Omission_SensorInput_Out3;
      Omission_SensorInput_Out3 -[ out Loss_Data3 ]-> Omission_SensorInput_Out3;
    properties
      occurrence => poisson 1.0e-6 applies to Fail1;
      occurrence => poisson 1.0e-6 applies to Fail2;
      occurrence => poisson 1.0e-6 applies to Fail3;
    end Basic.SensorInput;

   error model implementation Basic.TempProcess
   transitions
      ErrorFree -[ EMI ]-> Omission_TempProcess_Out;
      ErrorFree -[ Omission_TempProcess_In1 ]-> Omission_TempProcess_Out;
      Omission_TempProcess_Out -[ out Loss_Data ]-> Omission_TempProcess_Out;
    properties
      occurrence => poisson 1.0e-6 applies to EMI;
    end Basic.TempProcess;

    error model implementation Basic.SensorInput_D1--type1
    transitions
      ErrorFree -[ Fail ]-> Omission_SensorInput_D1_Out;
      Omission_SensorInput_D1_Out -[ out Loss_Data ]-> Omission_SensorInput_D1_Out;
    properties
      occurrence => poisson 1.0e-6 applies to Fail;
  end Basic.SensorInput_D1;
  ...
    error model implementation Basic.SensorInput_P1--type2
    transitions
      ErrorFree -[ Fail ]-> Omission_SensorInput_P1_Out;
      Errorfree -[Omission_SensorInput_P1_In]-> Omission_SensorInput_P1_Out;
      Omission_SensorInput_P1_Out -[ out Loss_Data ]-> Omission_SensorInput_P1_Out;
    properties
      occurrence => poisson 1.0e-6 applies to Fail;
  end Basic.SensorInput_P1;
  **};
end ErrorModel_TempControl_System;
```

Figure 4.7 Partial AADL error model type and implementation declaration

4.3 Model transformation from AADL to HiP-HOPS

The example below shows how the rule for transforming AADL connections to HiP-HOPS Line works. This rule is responsible for the construction of a `hiphops!Line` element. In the top level of the temperature monitoring system (see Figure 4.1 and Figure 4.5), there are three event connections, two of which share the same destination port (`TempProcess_In1`).

```
EventConnection1: SensorInput.SensorInput_Out1 ->
                  TempProcess.TempProcess_In1;
EventConnection2: SensorInput.SensorInput_Out2 ->
                  TempProcess.TempProcess_In1;
EventConnection3: SensorInput.SensorInput_Out3 ->
                  TempProcess.TempProcess_In2;
```

The third data connection, which does not share a port with any other data connection, is transformed into a single Line. The first and second event connections share the same destination port `TempProcess_In1` and thus correspond to a single HiP-HOPS Line. The PortExpression is obtained by using the connection to Line transformation algorithm. The algorithm first obtains a set of output propagations (or error states) that propagate to the port (`TempProcess_In1`) from the error models of connected (source) components. Here, the set of output propagations includes the output error states defined in the when clause of the guard_in expression for port (`TempProcess_In1`), i.e. {`Omission_SensorInput_Out1, Omission_SensorInput_Out2`}. The algorithm then creates a HiP-HOPS Connection for each of the output propagation (or error state). Each of the output propagation (or error state) will be treated as a failure class and for each of the failure class the `PortExpression` is obtained by using the AADL connection to HiP-HOPS Line conversion algorithm as discussed before. The algorithm includes the error and source port if the output propagation (or error state) is defined in the error model of that source component. The source ports appear in the `PortExpression` with the syntax `<output propagation>-<portname>`. The HiP-HOPS result is shown below.

```
<Line>
  <Type>Directed</Type>
  <Connections>
     <Connection>
        <FailureClass>Omission_SensorInput_Out1</FailureClass>
        <Port>TempProcess.TempProcess_In1</Port>
        <PortExpression>
           Omission_SensorInput_Out1-SensorInput.SensorInput_Out1 OR
           Omission_SensorInput_Out1-SensorInput.SensorInput_Out2
        </PortExpression>
     </Connection>
     <Connection>
        <FailureClass>Omission_SensorInput_Out2</FailureClass>
        <Port>TempProcess.TempProcess_In1</Port>
        <PortExpression>
           Omission_SensorInput_Out2-SensorInput.SensorInput_Out1 OR
           Omission_SensorInput_Out2-SensorInput.SensorInput_Out2
        </PortExpression>
     </Connection>
  </Connections>
</Line>
```

Figure 4.8 shows the resulting HiP-HOPS dependability model for the process `SensorInput_P1` (see Figure 4.2, Figure 4.6 and Figure 4.7). Note that, both the AADL architecture model including component name and ports name and type and the associated error model including error events and error states etc. are transformed into the HiP-HOPS model.

In more detail, the Ports section (lines1 to 11) is obtained by transforming the architecture of component `SensorInput_P1` in Figure 4.2. The section from line 14 to line 40 is obtained by transforming the model property defined in `SensorInput_P1.Impl` shown in Figure 4.6 and the error model implementation `Basic.SensorInput_P1` shown in Figure 4.7. For example, the basic event `Fail` in line 19 and its failure rate `1.0e-6` in line 22 are transformed from the error event `Fail` and its occurrence property defined in Figure 4.7. The output deviation `Omission_SensorInput_P1_Out-SensorInput_P1_Out` in line 29 and its failure expression

```
(Omission_SensorInput_D1_Out-SensorInput_P1_In AND
 Omission_SensorInput_D2_Out-SensorInput_P1_In AND
 Omission_SensorInput_D3_Out-SensorInput_P1_In) OR Fail
```

in line 32 to line 34 are transformed from the state machine transitions defined in `Basic.SensorInput_P1` shown in Figure 4.7 and the guard_in error property defined in Figure 4.6.

```
0 <Component>
1    <Name>SensorInput_P1</Name>
2    <Ports>
3      <Port>
4        <Name>SensorInput_P1_In</Name>
5        <Type>in</Type>
6      </Port>
7      <Port>
8        <Name>SensorInput_P1_Out</Name>
9        <Type>out</Type>
10     </Port>
11   </Ports>
12   <Implementations>
13     <Current>
14       <Name>BasicSensorInput_P1</Name>
15       <Cost>0</Cost>
16       <FailureData>
17         <BasicEvents>
18           <BasicEvent>
19             <Name>Fail</Name>
20             <UnavailabilityFormula>
21               <Constant>
22                 <FailureRate>1.0e-6</FailureRate>
23               </Constant>
24             </UnavailabilityFormula>
25           </BasicEvent>
26         </BasicEvents>
27         <OutputDeviations>
28           <OutputDeviation>
29             <Name>Omission_SensorInput_P1_Out-SensorInput_P1_Out</Name>
30             <SystemOutport>true</SystemOutport>
31             <FailureExpression>
32                (Omission_SensorInput_D1_Out-SensorInput_P1_In AND
33                Omission_SensorInput_D2_Out-SensorInput_P1_In  AND
34                Omission_SensorInput_D3_Out-SensorInput_P1_In) OR Fail
35             </FailureExpression>
36           </OutputDeviation>
37         </OutputDeviations>
38       </FailureData>
39     </Current>
40   </Implementations>
41 </Component>
```

Figure 4.8 The result HiP-HOPS model of process `SensorInput_P1` transformed from AADL models

Once the AADL model is transformed into a HiP-HOPS model, HiP-HOPS is able to analyse the dependability of this model and generate the FMEA table, fault trees, and etc. Figure 4.9 and Figure 4.10 show the screenshot of the resultant FMEA tables. In Figure 4.9, the FMEA shows that for seven components, a `Fail` will directly affect the system output. For example, a failure of `SensorInput_D6` will directly cause omission of the system and therefore a critical failure. The FMEA indicates that these seven components (i.e., from component `SensorInput_D6` to

component `TempProcess_P3`) are the critical elements in this design and therefore should be designed unlikely to fail to reduce chance of failure.

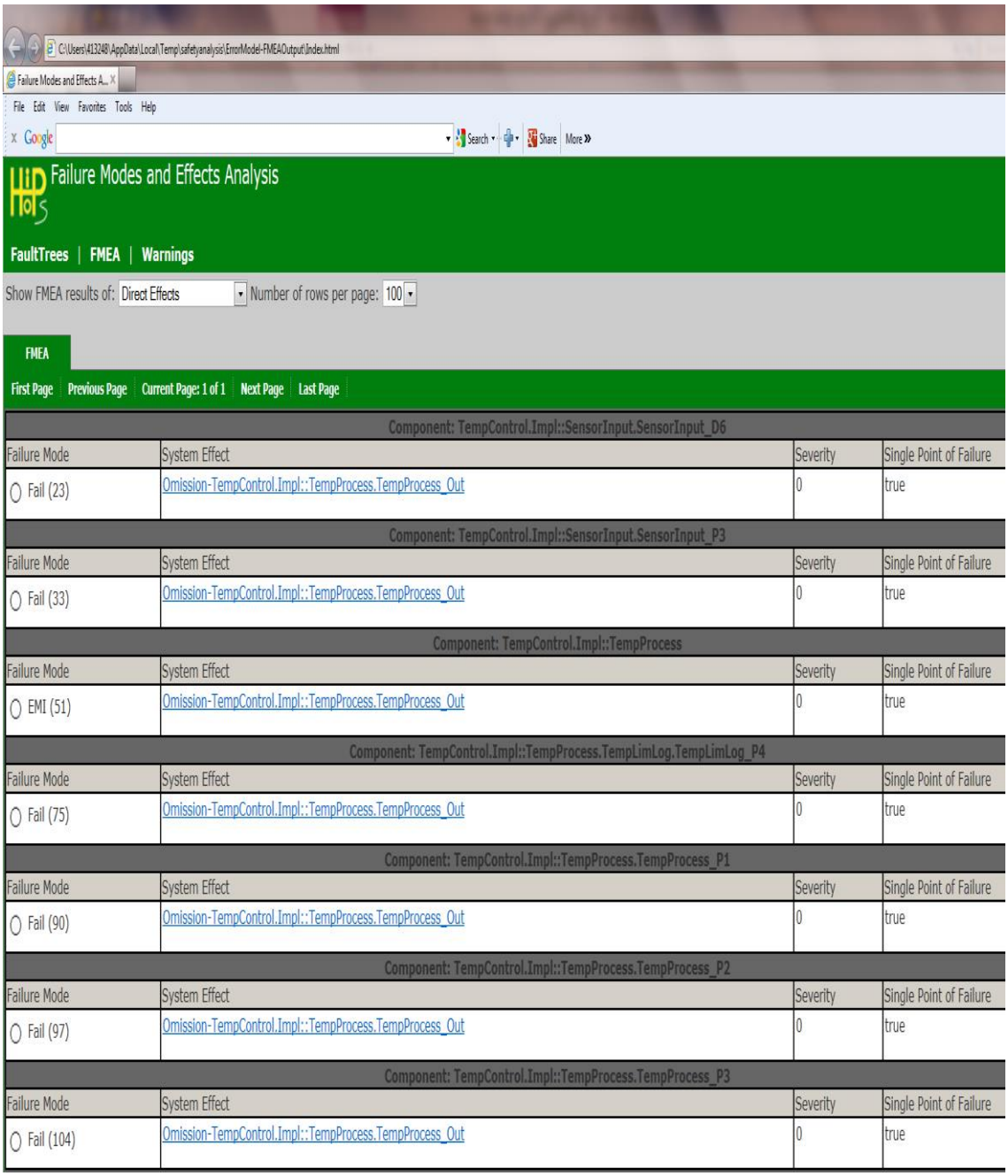

Figure 4.9 The screenshot of resultant FMEA of temperature monitoring system generated by HiP-HOPS from the initial AADL model after transformation: Direct effects

Figure 4.10 shows the FMEA further effects for component `SensorInput_P1`. The FMEA shows that the conjunction fail of `SensorInput_P1` and `SensorInput_P2` will cause the system fail. Similarly, the system will fail when `SensorInput_P1`, `SensorInput_D4` and `SensorInput_D5` fail simultaneously. This indicates the design should also pay attention to the common cause failures, e.g., EMI, to these components.

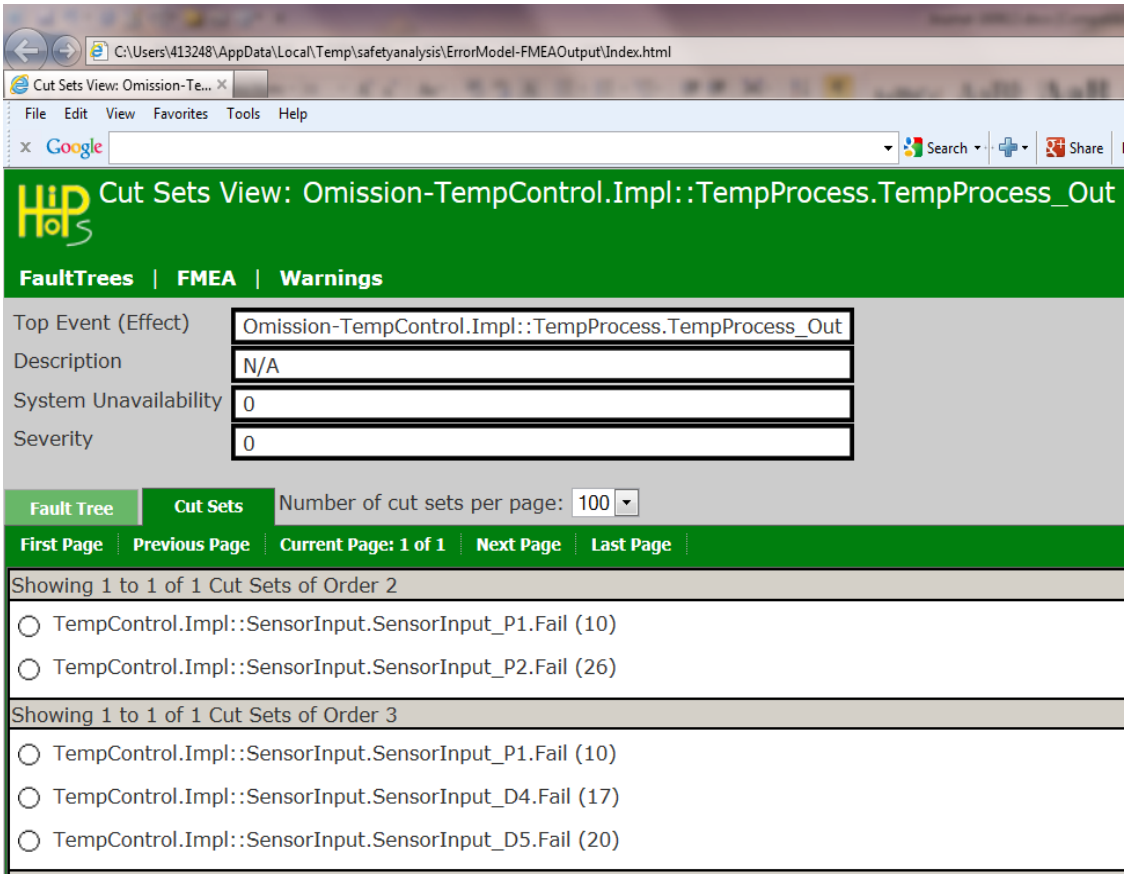

Figure 4.10 The screenshot of resultant FMEA of temperature monitoring system generated by HiP-HOPS from the initial AADL model after transformation: Further effects of component `SensorInput_P1`

4.4 Model transformation checking

In order to evaluate the developed transformation method between AADL and HiP-HOPS, a test system (the temperature monitoring system discussed in this chapter) is selected to test the transformation method. The selection criterion to the test system is that the test system should contain all the elements and characteristics that cover all the identified metamodel elements defined in the transformation rules. A three-step expert review method is adopted on the selected test system:

In the first step, the expert first reviews the functional description of the selected test system. Especially, the architectural description and the dependability-related data e.g. failure behaviour is reviewed. The expert then reviews the implementation of the test system modelled in AADL and HiP-HOPS respectively. The AADL and HiP-HOPS implementations of the test system are reviewed separately and independently by the expert. This ensures that the abstracts of the test system are completely reserved when it is implemented in two different modelling languages according to the system description. This is done by reviewing whether the test system is

modelled completely according to each language's modelling concepts. For example, the use of AADL error state machine and HiP-HOPS failure expression for modelling the failure behaviour of the test system respectively.

In the second step of the review method, the expert reviews the transformation rules defined between the two domains i.e. AADL and HiP-HOPS. Each transformation rule (mappings between AADL and HiP-HOPS) is then reviewed to ensure that the selected test system covers all the metamodel elements that are defined in the transformation rules. A check list of all the metamodel elements defined in the transformation rules are checked against all the metamodel elements showing in the test system. This ensures whether the selected test system covers all the metamodel elements defined in the transformation rules. The ATL transformation tool can indicate how many rules are executed when executing the transformation on the test system. This helps to check that all the rules are executed for the test system.

In the third step of the review method, the expert runs the two implemented models of the test system and obtains two HiP-HOPS outputs i.e. fault trees from each run. The first HiP-HOPS outputs (fault trees) are generated by modelling the test system in HiP-HOPS and HiP-HOPS directly produces the fault trees. The second HiP-HOPS outputs (fault trees) are produced from the output of the model transformation that is the input to HiP-HOPS. In this step, the test system is firstly implemented in AADL and then transformed to HiP-HOPS by using the developed model transformation tool – AADL2HiP-HOPS. Once the two HiP-HOPS output (fault trees) in hand they are then compared by the expert according to the correction criteria. The correction criterion is: the two fault trees must be exactly the same i.e. the failure expression for the top event of the system or component should be logically equal. The expert reviews the failure expressions for the top event and checks if any differences exist. The minimal cut sets of the top event are checked and compared each other. The correctness check is done to the component level. For example, both the HiP-HOPS outputs (failure expressions) for the top event of Omission_SensorInput_P1_Out-SensorInput_P1_Out component SensorInput_P1 are exactly the same as it shows in the <FailureExpression> element in Figure 4.8.

4.5 The transformation cost estimation

The research hypothesis is that model transformation is a cost effective way to maximise the utility of model-based dependability analysis and optimisation to AADL models. The implementation of the transformation method and testing using case studies is evidence towards the validation of the model transformation method and that it is an effective way to analyse the AADL dependability models. In all engineering tasks, cost is also important. To consider the

cost effectiveness of the transformation method it is necessary to consider the costs of using the transformation method. The only evidence available for the cost of the method is the cost of the work done in this thesis. To consider the transformation cost taken in this project, the author estimates the following times were used for the tasks listed below:

1. Four months to learn the AADL model,

2. Three months to learn the HiP-HOPS model,

3. Three months to learn the ATL and Eclipse plug-in development,

4. Twenty months to design, implement and test the model transformation.

The estimated time is a full cost of a thirty months' work. The author supposes that the transformation designer is not an AADL or HiP-HOPS expert and the transformation designer has to learn the two models first and then to design the transformation. This was the case for the author who did not have any background knowledge of AADL or HiP-HOPS before the start of the project. Note that this is not a rigorous measure of the cost but the author believes it is a reasonable estimation based on his development experience. Furthermore, because there is no objective data available for the cost of model transformation tasks then the author believes that this estimated cost data is useful evidence.

4.6 Summary

In this chapter, a temperature monitoring system was used as a case study to illustrate and test the model entity transformation rules.

The transformation method is tested on a test system (the temperature monitoring system discussed in this chapter). The test system is developed in AADL development tool OSATE and OSATE can automatically validate the validity of the developed system in the context of AADL. This ensures the test system is valid to AADL. The output of the transformation is a HiP-HOPS input XML file which conforms to HiP-HOPS's metamodel. If an input of a HiP-HOPS XML file is not complete or correct HiP-HOPS will identify problems of the input file and will not produce any output e.g. fault trees. This gives confidence that the transformed output conforms to HiP-HOPS metamodel. However, this does not implicate the transformation itself is correct. To ensure a correct transformation a check method is required.

An expert review method is used to validate the model transformation. In this validation method, two HiP-HOPS fault trees are generated from two different paths. One HiP-HOPS fault trees is produced by directly implementing the test system in HiP-HOPS and the other is

produced by transforming the same test system that is implemented in AADL to HiP-HOPS. The two produced fault trees are compared and checked by the dependability expert. Correctness of the transformation is ensured by checking that the fault trees (logical failure expressions) for the top event of the system are exactly the same.

The adopted transformation validation method is mainly a manual process. One drawback of this method (expert review) is that it is time consuming and limits to the size of the case study. For this test system, it contains only few components and connections and the hierarchy of the system is simple. In this case, a manual expert review process is sufficient and applicable. However, it may make the review process much harder and time consuming if the size of the system is quite large e.g. when southlands of components are contained. Also, the check process is mainly based on the knowledge of the expert and thus becomes unrealistic when no such expert is available.

Future work could include automatic checking and comparison to ensure that the two fault trees are the same. Moreover, to validate the model transformation method shown in this work, more experiment using AADL models with distinct size and complexity is required to show the feasibility and scalability of the method. In particular, testing approaches to model transformation validation should also be explored. Baudry et al., (2006) discussed some issues related to define techniques for testing transformations e.g. the issue for test data that must conform to the structure, constraints and precondition of metamodel. This requires an automated test case generation approach to produce test case systematically. A systematic validation of model transformation, as discussed in Kuster (2004, 2006) may also be considered in the future.

The model transformation approach has been shown to be an cost effective way to provide fault tree and FMEA analysis of AADL models. It is possible, using a model transformation approach, to provide fault tree and FMEA analysis of AADL models by exploiting an existing non-AADL method and supporting tool, i.e. HiP-HOPS.

In this chapter the method and tool is shown to provide fault tree and FMEA analysis of AADL models. The next chapter (Chapter 5), however, will focus on providing AADL models with access to multi-objective system optimisation.

Chapter 5 Multi-objective architecture optimisation for AADL dependable systems

This Chapter first tackles the problem of describing, within an AADL model, the design space of alternatives. A new AADL property set is developed for modelling component and system variability for cost and dependability optimisation. Secondly, the developed method is illustrated with an example of an AADL model of a safety critical embedded system with fault tolerant schemes (Adachi et al., 2011). The schemes comprise self-protection, self-checking and checkpoint-restart and process-pair mechanisms. Not all of these mechanisms need be employed at a given component and the choice of mechanism at each component leads to a space of design choices. In general there is a dependability-cost trade-off that should be optimised. For large systems, this design space is very large and cannot be explored efficiently without automation. Third, the Chapter extends the model transformation on AADL to HiP-HOPS transformation by opening up the optimisation capabilities of HiP-HOPS to AADL models. This allows the architecture optimisation with respect to dependability and cost of AADL models. The relative rules for optimisation are defined and implemented in ATL.

Note, some of the material in this chapter has been published in Mian and Bottaci (2013), Mian et al. (2013a) and Mian et al. (2013b).

5.1 Optimisation modelling for AADL dependability systems

5.1.1 Introduction

Common critical systems design requirements are high dependability and low cost. Design is a complex collaborative exercise and typically there are many ways in which dependability requirements can be achieved, including, among others, employing fault tolerant architectures and incorporating high integrity components. In a typical system, the space of possible designs is enormous and therefore finding a solution that meets the dependability requirements with minimal cost is difficult. If no solution can meet all the dependability requirements the goal is often to design a system that achieves the key requirements with the best possible trade-offs between dependability and cost.

For a given system, optimisation may be performed with respect to various objectives, for example, cost, reliability etc. There is still a lack, however, of analysis techniques and tools that can perform a dependability analysis and optimisation of AADL models.

A cost effective way of adding system dependability analysis and optimisation to models expressed in AADL is to exploit these capabilities of HiP-HOPS. Model transformation can be

used to transform the AADL model into an equivalent HiP-HOPS model. Chapter 3 has previously described a new model transformation method (AADL2HiP-HOPS). This has been integrated as a plug-in in the AADL model development tool: OSATE, for the automatic generation of HiP-HOPS dependability models from high level AADL architecture models. This Chapter extends Chapter 3 on AADL to HiP-HOPS transformation by opening up the optimisation capabilities of HiP-HOPS to AADL models. This allows the architecture optimisation of AADL models with respect to dependability and cost.

5.1.2 Method

Model optimisation depends on the possibility of choosing between model variations. Variability is a prerequisite for optimisation, because it creates the design space of alternatives which is explored in the search of the best architectural solutions. In this work, variability is introduced at the component level, by associating alternative components to particular components. A designer, for example, may wish to consider a number of alternative components to implement a particular function in a system. As another example, the designer may wish to add, or at least consider the possibility of adding, additional dependability to selected components of the model. In this case, the two alternatives are inclusion or absence of components that provide protection, redundancy or recovery. In both these cases there is a requirement to model component alternatives.

HiP-HOPS has an optimisation capability for system dependability and cost. The HiP-HOPS system model contains features that directly support optimisation. One of these is the specification of alternatives for a component. For each component and its alternatives, attributes that are relevant to the optimisation must be specified. These attributes include cost, weight, reliability etc.

In order to specify the application of the optimisation process, it is possible to specify for each component whether it should be replaced with an alternative. This allows the designer flexibility in including or excluding components in the optimisation. In addition, from the given set of alternatives for that component, it is possible to specify that a specific alternative should not be used as a replacement. This allows the designer fine-grained control over the particular alternatives used for a component, a useful feature when there is a library supplied set of replacement components.

At the system level, there are a number of optimisation parameters that include optimisation objectives, e.g., cost, reliability etc. and the search termination criterion. Optimisation is inherently a search based process that may continue indefinitely where there is a large search

space and the difficulty of recognising an optimal solution. The termination criterion can be expressed in terms of the computing resources consumed during the search.

The AADL modelling language has no direct support for optimisation of models. This has motivated the development of a new set of properties into the AADL modelling language. These properties allow an AADL model to capture the information required to perform model optimisation. In particular they can represent the relevant optimisation properties and allow component alternatives to be specified. The model transformation from AADL to HiP-HOPS is able to use this information to produce a HiP-HOPS optimisation model.

5.1.2.1 Optimisation example

To illustrate the use of alternative components, consider the AADL description of a system shown in Figure 5.1. This figure shows part of the AADL text description of a motor vehicle Pre-collision system (PCS) described by Adachi et al. (2011), which is to be optimised for dependability and cost. A Pre-collision system (PCS) first sends out warning messages to the driver once a potential collision threat has been identified. If the brakes are failed to be applied by the driver, then a brake assist mechanism will be triggered immediately to increase the braking force before a collision to reduce the collision injury. Eventually emergency braking is initiated, according to the environmental conditions.

Figure 5.1 shows the AADL description of 42 subcomponents in the PCS including sensors, processes, software modules and actuators. There are 7 key components listed as `Detection_Module` through to `Actuator_Driver`. For each of these 7 components there is a fault-tolerant scheme. This scheme involves the presence or absence of self-checking and recovery components described in Adachi et al. (2011). A specific design for the PCS system can be modelled as a specific architecture involving particular selections from alternative components. Where, for example, a self-checking component is present, the "present" alternative is selected, if the self-checking component is absent, the empty alternative is selected. Depending on the degree of dependability required, different parts of the fault-tolerant scheme can be selected. In total, there are 12 potential component architectures for each of these 7 components. The size of the design space to be explored is therefore of the order of 12^7 models and as such it is very difficult to do this kind of optimisation manually.

The optimisation process is a guided search of the possible models (combinations of possible choices) that result when components are replaced by alternatives. Each model instantiation produced by a particular choice of components is evaluated by HiP-HOPS for dependability and cost. It has been shown that optimisation heuristics such as genetic algorithms described by

Parker (2010) can exploit a model dependability evaluation function to obtain optimum or near optimum designs.

```
system implementation PrecollisionSystem.impl
  subcomponents
    Radar_Sensor: device SensorMemory.RadarSensor;
    Vehicle_Sensor: device SensorMemory.VehicleSensor;
    Pedal_Sensor: device SensorMemory.PedalSensor;
    Data_Memory: device SensorMemory.DataMemory;
    Switch_Sensor: device SensorMemory.SwitchSensor;
    Detection_Module: process DetectionModule.impl;
    Monitoring_Module: process MonitoringModule.impl;
    Switch_Controller: process SwitchController.impl;
    PCS_Logic1: process PCS_Logic1.impl;
    PCS_Logic2: process PCS_Logic2.impl;
    Decision_Module: process DecisionModule.impl;
    Actuator_Driver: process ActuatorDriver.impl;
    SP01: process SelfProtection.SP01;
    SP02: process SelfProtection.SP01;
    . . .
    SP16: process SelfProtection.SP02;
    SC01: process SelfChecking.SC01;
    SC02: process SelfChecking.SC01;
    . . .
    SC07: process SelfChecking.SC02;
    CR_PP01: process CRAndPP.CR_PP01;
    CR_PP02: process CRAndPP.CR_PP01;
    . . .
    CR_PP07: process CRAndPP.CR_PP02;
  connections
    . . .
  properties
    Optimisation_Attributes::Maximum_Generations => 1000;
    Optimisation_Attributes::List_of_Objectives =>
         ([Objective=>Cost; Goal=>Minimise; Bound=>0..85;],
          [Objective=>Risk; Goal=>Minimise; Bound=>0..0.18;]);
end PrecollisionSystem.impl;
```

Figure 5.1 AADL text description of subcomponents and associated optimisation properties for the pre-collision system

5.1.3 Optimisation modelling in AADL

It is necessary for components and subsystems in AADL architecture models to be associated with one or more alternatives with respect to dependability and cost. AADL, however, has no specific features whereby the optimisation alternatives for a component can be represented.

In AADL, a component type can have several implementations, each of which must implement all the features declared in the component type description. This suggests that the optimisation

alternatives for a component might be represented as different implementations of that component. In practice, however, this could be unsatisfactory. The reasons for this dissatisfaction are explained in the following.

Although a component implementation and any of its replacements must all be of the same type, the concept of a different implementation of a component is more general than that of an optimisation replacement component. A model, for example, may contain two different implementations of a component, one of which complies with the European licensing authority, and another implementation which complies with the American licensing authority. Clearly, the choice of which of these two implementations to use in the system is not an optimisation concern but dependent on the system location. The optimisation method would need to be able to identify which implementations are optimisation alternatives and which are not.

Another disadvantage of using multiple implementations is that the alternatives are restricted to components that are of the same type, i.e. all alternative implementations must comply with the same type definition. Once the features have been specified in a component type definition, all alternative implementations are restricted to use the same features. Those alternative implementations that have different hierarchical architectures (with sub-components) are excluded. This seems to be a severe and an unnecessary restriction to the space of design alternatives.

Another possible scheme for representing variability within an AADL model is to use the modes concept as defined by Feiler and Gluch (2012). A mode can be used to distinguish different operational states, for example, “active” versus “idle”. Modes can be defined locally for each component and so a mode can be defined for each alternative to that component. By assigning each alternative component to a different mode, alternatives could be selected for each component by specifying a specific mode for that component. Again this scheme is unsatisfactory. Each mode combination simply identifies an optimisation alternative. The modes introduced to represent alternative components have no meaning as modes within the system and should therefore remain hidden from the model.

Both of the schemes for representing variability in AADL considered above suffer from the problem that the representation of alternatives is not explicit but is “riding on the back” of an existing AADL concept, either alternative implementations or modes. In the case of the alternative implementations, optimisation alternatives must be distinguished as “optimisation” implementations and in the cases of the modes concept, specific “optimisation modes” would be necessary. In both cases, the legitimate uses of these concepts would be obstructed.

An explicit and secure way to define the alternatives for a component is required which does not compromise any existing AADL concept. A possible scheme is to define a new property for a component to hold the alternative components for a given component in a specific alternatives list which is a property of that component. The content of the alternatives list could be populated by reference to a library of alternatives which contains pre-specified alternative component for various component types. With this scheme, a designer, according to his assessment of a particular component situation in the model, has the flexibility of adding specific optimisations to the library or to that component by specifying alternative implementations. The following section will discuss this developed AADL optimisation modelling concept.

5.1.4 The definition of alternatives and optimisation properties

5.1.4.1 AADL extensibility

As described by Feiler and Gluch (2012), AADL is an extensible language. AADL allows the designer to introduce additional properties and property types through the concept of a property set. This is used when it is necessary to add information to an AADL model and no appropriate predefined property has been declared. The property set is used to define a conceptually related group of properties, property types and property constants. Property set names are unique identifiers. Each property set provides a separate name space. By preceding the property name, property type name, or property constant name with the property set name followed by "::" one can uniquely reference them.

Figure 5.2 shows an example of an AADL property set declaration which describes some properties of a person.

```
property set personProperties is
  Height: aadlreal applies to (person);
  Sex: type enumeration (Male, Female);
  Name: constant aadlstring => "Amy";
end personProperties;
```

Figure 5.2 The AADL property set declaration for some example properties of a person

The AADL description of the general syntax for declaring a property set is shown in Figure 5.3. The `property set` type is identified by its name. The optional `with` clause identifies any other property sets that are referenced from within the property set. The `property`, `property type` and `property constant` declarations can be defined in any order.

```
property set <property set name> is
[with <property set name(s)>;]
  <property declaration(s)>
  <property type declaration(s)>
  <property constant declaration(s)>
end <property set name>;
```

Figure 5.3 The AADL property set declaration syntax

In more detail, the `property` declarations define a property by declaring a name and a type for the property and identify which AADL model element (e.g., software component, hardware component, and connection) accepts values for this property through property association. The `property type` declarations define a type for the values that are acceptable to a property. One can define a property to be of a specific property type, hence limiting the values that this property accepts in property association. The syntax for a `property type` declaration must include a `type` keyword followed by the type definition. The type definition can be one of the built-in property types (e.g., `aadlinteger`, `aadlstring`, `aadlboolean` and `aadlreal`) or it can be defined with one of the type constructor (e.g., `range of real values`, `classifier` and `record`) that can refer to another property type or property name. The `property constant` declarations define a symbolic name for a property value. One can reference this name in property declarations wherever the value itself is permissible.

5.1.4.2 New AADL optimisation properties

Figure 5.4 shows the AADL description of using an AADL `property set` to define new optimisation properties named `Optimisation_Attributes`. The names of the optimisation properties such as `Cost` (of type `aadlreal`), `Weight` (of type `aadlreal`) are used at the component level. In the HiP-HOPS optimisation model, it is possible to selectively apply optimisation to particular sets of components. This is the motivation for the two properties `Optimise` (of type `aadlboolean`) and `Exclude_From_Optimisation` (of type `aadlboolean`) which are used at the component level. This is indicated by the `applies to (all)` clause. Other attributes, i.e. `Maximum_Generations` (defined as a constant with type of `aadlinteger`) is a property of the optimisation. This property is a termination condition and limits the search of model space, which, if not limited for large models, might proceed for a very long time.

```
property set Optimisation_Attributes is
with Optimisation_Objectives;
  Cost: aadlreal applies to (all);
  Weight: aadlreal applies to (all);
  Optimise: aadlboolean applies to (all);
  Exclude_From_Optimisation: aadlboolean applies to (all);
  Maximum_Generations: constant aadlinteger => 1000;
  List_of_Objectives: list of Optimisation_Objectives:
                      Objectives applies to (system);
end Optimisation_Attributes;
```

Figure 5.4 The use of AADL property set to define optimisation properties

Another property that applies to the optimisation is the `List_of_Objectives` property that specifies a list of specific optimisation objectives. An optimisation objective has a name, a direction and a range. For example, (cost, minimise, [700, 23000]). The AADL description of the type `Objectives` (defined in Figure 5.5) is a `record` type with three fields, `Objective`, `Goal` and `Bound`. Each of these fields has its own type respectively, i.e., `aadlstring`, `aadlstring`, and `range of aadlreal`. This property is applicable to the system component type. Note that, the `List_of_Objectives` property is defined as list of `record` defined in the property set `Optimisation_Objectives`. Therefore, one must include that property set in the "`with`" clause of `Optimisation_Attributes`.

```
property set Optimisation_Objectives is
  Objectives: type record (
  Objective: aadlstring; Goal: aadlstring; Bound: range of aadlreal;
  );
end Optimisation_Objectives;
```

Figure 5.5 The use of AADL property set to define a new `Objectives` property type

Finally, in order to specify the alternative components, there is an AADL property (the property is of type `classifier`) named `List_of_Alternatives` (shown in Figure 5.6) which is defined in the property set named `Alternatives`. This property is applicable to all of the component types. The AADL description of the `Alternatives` property (shown in Figure 5.6) defines a list of AADL classifiers that are alternative implementations of a given component. A classifier is an AADL type for representing component type and implementation declarations.

```
property set Alternatives is
  List_of_Alternatives: list of classifier applies to (all);
end Alternatives;
```

Figure 5.6 The use of AADL property set to define the `List_of_Alternatives` property. This property specifies the alternative components for a given component

Figure 5.1 also shows the association of optimisation properties for the Pre-collision system. In Figure 5.1, the maximum generation (`Maximum_Generations`) is set to value 1000. The objectives property is defined as a list of record type. Two objectives are defined. The first objective is `Cost` with the goal of `Minimise` and the bound of `Cost` is between 0 and 85. The second objective is `Risk` with the goal of `Minimise` and the bound of `Risk` is between 0 and 0.18. For each objective, there is a defined lower bound and an upper bound. The bounds restrict the optimisation searches to find candidates within those lower and upper bounds for each objective. The use of lower and upper bounds is an important requirement in multi-objective optimisation as they can be used to limit the optimal solutions found to the region of interest.

Figure 5.7 shows the association of defined optimisation properties for process type `CR_PP01` and the alternative implementations of this type. From Figure 5.7, the `Optimise` property with value `true` means that alternatives for process `CR_PP01` should be considered in the optimisation process. This allows the optimisation process to be applied selectively to parts of the system. For the implementation component, `CR_PP01.impl`, the property `Exclude_From_Optimisation` property with value `true` means that this alternative component should not be used as a replacement, i.e., is not an available alternative. This allows the designer fine-grained control over the alternatives used in the optimisation. The `Cost` property specifies the cost of this implementation. The process `CR_PP01` has two alternative implementations (`CR_PP01.NA_01` and `CR_PP01.CR_01`), which is specified in the properties via setting the property value of `List_of_Alternatives`. For each of the implementation, the cost, weight and other optimisation properties are associated via set the property value respectively.

```
process CR_PP01
  features
    CR_PP01_Input: in event port;
    CR_PP01_Output: out event port;
  properties
    Optimisation_Attributes::Optimise => true;
end CR_PP01;

process implementation CR_PP01.impl
  properties
    Optimisation_Attributes::Exclude_From_Optimisation => false;
    Optimisation_Attributes::Cost => "25";
    Alternatives::List_of_Alternatives => (
      process CR_PP01.NA_01, process CR_PP01.CR_01);
  annex Error_Model {**
    model => ErrorModel_PCS::Basic.Process_Pair_01; **};
end CR_PP01.impl;

process implementation CR_PP01.NA_01
  properties
    Optimisation_Attributes::Exclude_From_Optimisation => true;
    Optimisation_Attributes::Cost => "0";
  annex Error_Model {**
    model => ErrorModel_PCS::Basic.CR_PP_NA_01; **};
end CR_PP01.NA_01;

process implementation CR_PP01.CR_01
  properties
    Optimisation_Attributes::Exclude_From_Optimisation => false;
    Optimisation_Attributes::Cost => "20";
  annex Error_Model {**
    model => ErrorModel_PCS::
       Basic.CheckPoint_Restart_01;**};
end CR_PP01.CR_01;
```

Figure 5.7 The association of defined optimisation properties for process type `CR_PP01` and different implementations of this type

The above AADL optimisation properties and component alternatives form the basis for the transformation of the AADL model to an optimisable HiP-HOPS model. To perform this transformation into an optimisable HiP-HOPS model, the early transformation described in Chapter 3 is extended with additional transformation rules (see section 5.2 below). These rules transform the AADL elements that comprise the AADL optimisation model into corresponding HiP-HOPS optimisation elements. These transformations are largely structural. The values of the basic attributes involved e.g., cost, reliability etc. do not require translation. The transformation transforms the AADL model shown in Figure 5.1 and Figure 5.7 using the new properties into the HiP-HOPS model. Once the AADL system model is transformed into HiP-HOPS, HiP-HOPS will automatically analysis and optimise the system architecture.

Ultimately, the nature of the optimisation determines the properties that must be represented in the model. This section targets the HiP-HOPS optimisation objectives of cost and dependability but the developed method in this section is not specific to the HiP-HOPS optimisation techniques. The developed optimisation modelling scheme can be easily transplanted to other similar search-based optimisation methods and tools. This follows from the use of the high level modelling techniques shown in this section. For example, the designer can easily specify their own optimisation objectives in the system by using the defined `List_of_Objectives` property shown in Figure 5.4. The `List_of_Objectives` property is defined as list of `record` type (defined in Figure 5.5) so that the designers can easily add, delete or modify any element in the record type.

5.2 The model transformation from extended AADL system optimisation model to HiP-HOPS optimisation model

The resulting AADL model extended with optimisation properties is transformed into a HiP-HOPS model for dependability analysis and optimisation. The model transformation from AADL to HiP-HOPS places component alternatives into the HiP-HOPS optimisation model. Each model instantiation produced by a particular choice of components is evaluated by HiP-HOPS for dependability and cost. This information guides the genetic algorithm search (Parker, 2010) to obtain optimum or near optimum designs.

The resulting model of the above optimisation modelling process is named as the AADL dependability and optimisation model. Figure 5.8 shows the overview of extended model transformation from AADL to HiP-HOPS. The extended AADL system optimisation model is shown as a box with dotted lines. The AADL system optimisation model is developed by using the optimisation modelling method described in section 5.1.

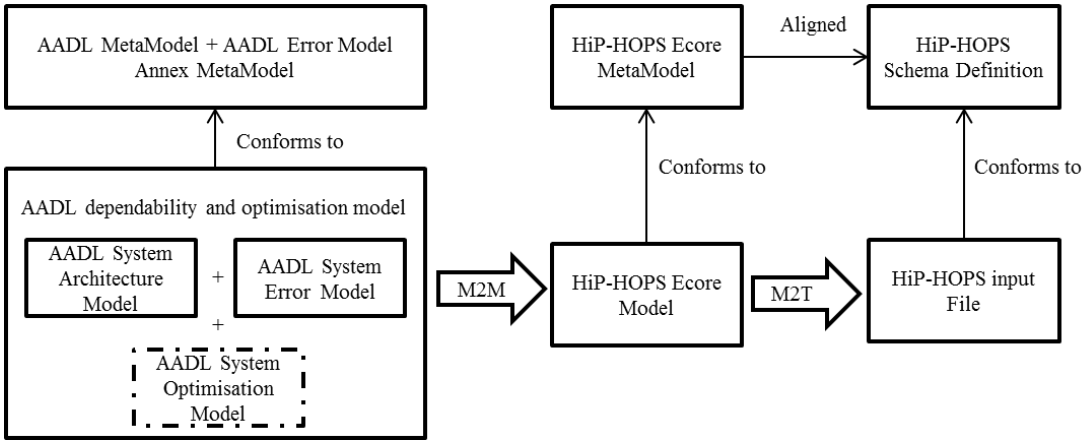

Figure 5.8 The overview of the extended transformation design

Figure 5.9 shows the transformation rule, which transforms AADL component alternatives in the extended AADL system optimisation model to HiP-HOPS `Alternatives` object. Each alternative defined in the AADL alternative property is an AADL `Classifier`. The rule obtains the information needed by matching the AADL object with that `Classifier` and transforms it to a HiP-HOPS `Alternative` object. A HiP-HOPS `Alternative` is a type of HiP-HOPS implementation which consists of some optimisation attributes such as `excludeFromOptimisation` and `cost` and a Failure data (`fData`) object. The value of these attributes and object is obtained by invoking the ATL helper, i.e., `IsAlternativeExcludeFromOptimisation()`, `getAlternativeCost()`, and `getAlternativeErrorModelImplementation()` respectively.

```
rule AlternativeClassifier2Alternatives{
from
  al : instance!ClassifierValue
to
  impalt : hiphops!Alternative(
    name <- al.componentClassifier.name,
    excludeFromOptimisation <-
       al.IsAlternativeExcludeFromOptimisation(),
    cost <- al.getAlternativeCost(),
    fData <- al.getAlternativeErrorModelImplementation()
  )
}
```

Figure 5.9 The ATL rule for transforming AADL alternative classifier to HiP-HOPS Alternative

5.3 Summary

This chapter discusses the potential routes to the representation of variability within an AADL model in order to optimise AADL dependable models based on dependability and cost. Some new AADL properties have been defined. Some of the optimisation properties need to be set in type declarations and some of them need to be set in implementation declarations. These properties allow a designer to express component variability in an AADL model. The optimisation modelling method is illustrated both at system and component levels of a dependable system. The benefit of this optimisation modelling method is that it allows designers to automatically optimise models on the basis of component level variability.

The chapter also illustrates the model transformation (in ATL rule) from extended AADL system optimisation model to HiP-HOPS optimisation model. The rule shows how AADL component alternatives contained in the extended AADL system optimisation model are transformed to corresponding HiP-HOPS Alternatives object. With this optimisation modelling

method and alternative transformation rule shown in this chapter together with the model transformation rules shown in Chapter 3, the AADL dependability and optimisation model can be created and transformed to HiP-HOPS for dependability and cost optimisation analysis. The overall benefit of this optimisation modelling and model transformation is that it automates and simplifies the dependability design process.

Chapter 6 Case study: aircraft wheel brake system

To illustrate the useful application of the optimisation modelling and transformation method, this chapter demonstrates the transformation using a case study. This case study has been published in Mian et al. (2013c).

6.1 System description

The aircraft wheel brake system model is adapted from the Aerospace Recommended Practice (in page 14 and 16 of ARP 4761, 1996) aircraft wheel brake system, which is also presented in (Joshi et al., 2006; and Sharvia, 2011). Some components such as accumulators and mechanical pedals are left out. These are part of the fault tolerant mechanisms. These fault tolerant components are left is because the research supposes the system is designed with a basic design at the beginning. Then the research aims to show how the assessment approach (i.e. HiP-HOPS architecture optimisation based on dependability and cost) may help the system designers to find design solutions based on both dependability and cost constrains from a more basic design.

These adaptations should not pose any threats to the validity of the obtained results because the system is chosen to demonstrate the application of the optimisation. The dependability and optimisation analysis results are dependent on the structure of the system, but the any changes in this structure (or failure data), can be reflected in the analysis accordingly. For example, the system architecture is expanded with additional components, there will be bigger search space (more configuration alternatives) during the optimization but it will not affect the validity of the results.

Figure 6.1 shows the basic system structure and Figure 6.2 shows the corresponding AADL description of the wheel brake system. The primary function of the wheel brake system is to provide safe braking function for aircraft which requires supplying correct pressure and preventing skidding. Braking can be either manual or automatic. Manual braking is controlled via brake pedals, while automated braking does not require pedal application. The automated braking is realised via Autobrake function which allows the pilot to provide the deceleration rate prior to takeoff or landing.

The braking system operates in one of two modes, `Normal` or `Alternate`. In `Normal` braking mode, `GreenPump` provides the required hydraulic pressure, and `Alternate` mode is held on standby. If failure occurs on `Normal` mode, the system moves to `Alternate` mode and hydraulic power is generated by the `BluePump`. In the original ARP 4761 example, another backup mechanism is in place lest both of the pumps fail. In this chapter, however, it has been

deliberately excluded to demonstrate how HiP-HOPS can be used to help guide the analysis process and the identification of potential safety measures.

The Brake System Control Unit (BSCU), is the digital controller which accepts inputs to compute braking and anti-skid commands. In its Normal operational mode, BSCU receives information from various input sources. It obtains brake pedal positions as input and processes this information to produce command signals to the brakes. When Autobrake is true, deceleration rate and aircraft speed are used to calculate the brake command. BSCU also monitors signals which indicate certain critical aircraft and system states to provide correct brake function, generate warnings, indications and maintenance information to other system.

Two hydraulic pressure lines are used: the Green line, powered by the GreenPump (Normal) and the Blue line, powered by the BluePump (Alternate). The GreenValve and the BlueValve are used to control the pressure from the GreenPump and BluePump respectively. The SelectorValve is located across the Green and Blue hydraulic lines, and selects only one of the two hydraulic systems to provide pressure to the brakes. This pressure is relayed to the corresponding meter valves, CMD/ASMeterValveG and CMD/ASMeterValveB respectively. The meter valves take two inputs: the incoming pressure and the valve position command. The valve position is adjusted to output the required amount of pressure based on the command from the BSCU.

The system switches to Alternate mode when the pressure along the green line falls below a threshold. Once BSCU identifies that Alternate line should be activated, it sends an OnAlternate signal which commands SelectorValve to switch to the Blue line. Once the system switches to Alternate, it will not revert back to Normal. The component labelled WBS is the pressure output block, the components NormalP and AlternateP serve only to propagate failures.

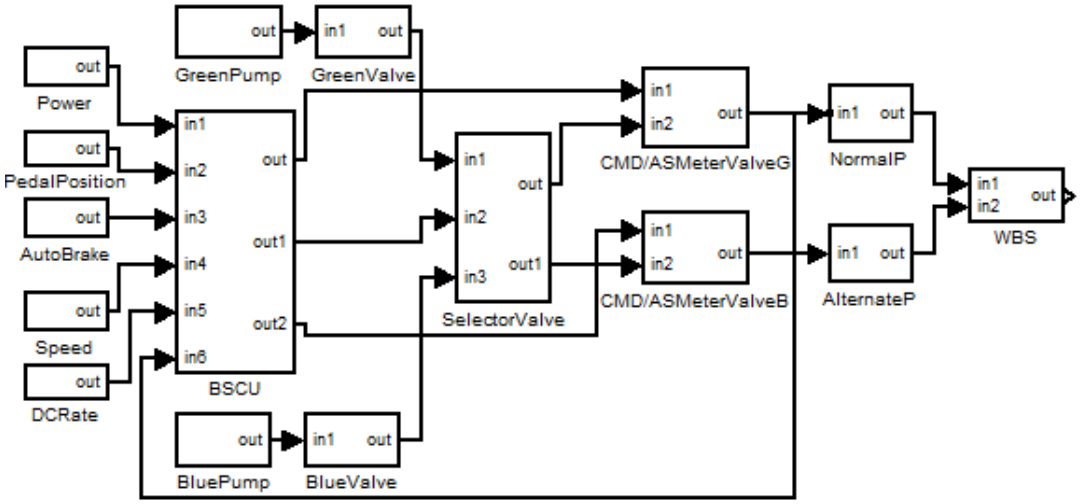

Figure 6.1 The basic system structure of aircraft wheel brake system

```
system implementation AWBS.SystemImpl1
  subcomponents
    Power: device DeviceType1.Power;
    PedalPosition: device DeviceType1.PedalPosition;
    AutoBrake: device DeviceType1.AutoBrake;
    NormalP: device DeviceType2.NormalP;
    AlternateP: device DeviceType2.AlternateP;
    CMDASMeterValveG: device DeviceType3.CMDASMeterValveG;
    . . .
    --Other components omitted
    . . .
    WBS: device DeviceType3.WBS;
    SelectorValve: device DeviceType4.SelectorValve;
    BSCU: device DeviceType5.BSCU;
  connections
    DataConnection1: data port Power.Output -> BSCU.Input1;
    DataConnection2: data port PedalPosition.Output -> BSCU.Input2;
    DataConnection3: data port AutoBrake.Output -> BSCU.Input3;
    . . .
    --Other connections omitted
    . . .
    DataConnection17: data port CMDASMeterValveG.Output ->
                                BSCU.Input6;
    DataConnection18: data port NormalP.Output -> WBS.Input1;
    DataConnection19: data port AlternateP.Output -> WBS.Input2;
end AWBS.SystemImpl1;
```

Figure 6.2 The AADL description for the aircraft wheel brake system

6.2 Failure data

The AADL Error Model Annex is used to model the system failure behavior. For simplicity, each component is assumed to be vulnerable to one internal failure which leads to the omission of component output. Other types of component failure (for example, commission or value failure) are not discussed but may be treated analogously. The internal failures for components GreenPump, GreenValve, BluePump, BlueValve, CMD/ASMeterValveG, CMD/ASMeterValveB and SelectorValve are denoted as GreenPumpBE, GreenValveBE, BluePumpBE, BlueValveBE, GCMDASBE, BCMDASBE, SelValveBE respectively. Internal failure in the BSCU (the command unit) is denoted as CMDBE. The input to the BSCU comes from Power, PedalPosition, AutoBrake, Speed and DCRate component. Internal failures in these input components are denoted as PowerBE, PedalPositionBE, AutoBrakeBE, SpeedBE, and DCRateBE respectively.

Figure 6.3 shows the AADL error model type definition and error model implementation for component Power, GreenValve and BSCU. Figure 6.4 shows the association of error model implementation Basic.BSCU (shown in Figure 6.3) to component BSCU, the guard_in and guard_out error properties for component implementation DeviceType5.BSCU and the alternative implementations of this component. Note the guard_in and guard_out error properties shown in Error_Model in DeviceType5.BSCU. These error properties specify conditions under which the input or output error propagations occur. From Figure 6.4, the Optimise property with value true means that alternatives for device DeviceType5 should be considered in the optimisation process. This property allows the optimisation process to be applied selectively to parts of the system. For the implementation component, DeviceType5.BSCU2, the property Exclude_From_Optimisation property with value true means that this alternative component should not be used as a replacement, i.e., is not an available alternative. This allows the designer fine-grained control over the alternatives used in the optimisation. The Cost property specifies the cost of this implementation. The device DeviceType5 has three alternative implementations (DeviceType5.BSCU2, DeviceType5.BSCU3, and DeviceType5.BSCU4), which is specified in the properties via setting the property value of List_of_Alternatives. For each of the implementations, the cost and other optimisation properties are specified.

```
package AWBS_ErrorModel
public
  annex Error_Model {**
    error model Basic --basic error model
    features
      ErrorFree: initial error state;
      GreenPumpBE, GreenValveBE,BluePumpBE, BlueValveBE, GCMDASBE,
      BCMDASBE, SelValveBE, CMDBE, PowerBE, PedalPositionBE,
      AutoBrakeBE,SpeedBE, DCRateBE: error event;
      Failed, Failed1, Failed2, Failed3: error state;
      Loss_Data: in out error propagation {occurrence => fixed 0.8};
      Loss_Data1, Loss_Data2: in error propagation;
    end Basic;

    error model implementation Basic.Power
    --error transitions omitted
    end Basic.Power;

    error model implementation Basic.GreenValve
    --error transitions omitted
    end Basic.GreenValve;

    error model implementation Basic.BSCU
    transitions
      ErrorFree -[ Loss_Data1 ]-> Failed1;
      ErrorFree -[ Loss_Data2 ]-> Failed1;
      ErrorFree -[ CMDBE ]-> Failed2;
      Failed1 -[ out Loss_Data ]-> Failed1;
      Failed2 -[ out Loss_Data ]-> Failed2;
    properties
      occurrence => poisson 1.0e-10 applies to CMDBE;
    end Basic.BSCU;
   …
  **};
end AWBS_ErrorModel;
```

Figure 6.3 AADL error model type definition and error model implementation for component `Power`, `GreenValve` and `BSCU`.

```
device DeviceType5
  features
    Input1: in data port; Input2: in data port; Input3: in data port;
    Input4: in data port; Input5: in data port; Input6: in data port;
    Output: out data port; Output1: out data port; Output2: out data
            port;
  properties
    Optimisation_Attributes::Optimise => true;
end DeviceType5;

device implementation DeviceType5.BSCU
  properties
    Optimisation_Attributes::Exclude_From_Optimisation => false;
    Optimisation_Attributes::Cost => "50";
    Alternatives::List_of_Alternatives => (device DeviceType5.BSCU2,
                 device DeviceType5.BSCU3, device DeviceType5.BSCU4);
  annex Error_Model {**
    model => AWBS_ErrorModel::Basic.BSCU;
    guard_in =>
      Loss_Data1 when Input1[Loss_Data],
      mask when others
      applies to Input1;
    guard_in =>
      Loss_Data2 when Input2[Loss_Data] AND
      (Input3[Loss_Data] OR Input4[Loss_Data] OR Input5[Loss_Data]),
      mask when others
      applies to Input2,Input3,Input4,Input5;
    guard_out =>
      Loss_Data when self[Failed2],
      mask when others
      applies to Output1;
  **};
end DeviceType5.BSCU;

device implementation DeviceType5.BSCU2
  properties
    Optimisation_Attributes::Exclude_From_Optimisation => true;
    Optimisation_Attributes::Cost => "20";
  annex Error_Model {**
    model => AWBS_ErrorModel::Basic.BSCU2;
  **};
end DeviceType5.BSCU2;

--Implementation of DeviceType5.BSCU3 and DeviceType5.BSCU4 similar to
above but cost and error model differ
```

Figure 6.4 Associated AADL error model, guard_in and guard_out error model properties for component BSCU and alternative implementations of this component

6.3 Dependability analysis of the wheel brake system model

The AADL wheel brake system model is transformed into a HiP-HOPS model for dependability analysis and optimisation. HiP-HOPS produces, FTA and FMEA. The analysis of the results of FTA and FMEA shows, for example, that the omission of `Power`, `BSCU` command unit and `SelectorValve` directly leads to omission of pressure on the wheel brake. The other single point failures are also identified. The obtained fault tree and FMEA in this work is checked with the result published in Sharvia, (2011) since the same case study is used. The correctness of the transformation is checked by comparing the obtained FMEA tables (i.e. generated from the transformation of the AADL dependability model to HiP-HOPS dependability model) with the validated FMEA published in Sharvia, (2011). For a small design model, manual analysis may be manageable. But for larger system, where this architecture may be nested within a more complex design, manual analysis becomes laborious and error-prone.

6.4 Design optimisation

In this case study, it is assumed that each component has four different alternatives (each with different failure rate and different cost). Components with lower failure rates have a higher cost. Table 6.1 summarises the failure rates and costs data for the component alternatives. The failure rates and costs of `CMD/ASMeterValves` follow those of green valves and blue valves. It should be noted that the values of failure rates are not based on any empirical data, but chosen hypothetically to illustrate the method.

Table 6.1 The failure rates and costs data for the component alternatives

Component	Failure Rate λ	Cost
BSCU1/ SelectorValve1	1e-10	50
BSCU2/ SelectorValve2	2e-10	20
BSCU3/ SelectorValve3	3e-10	10
BSCU4/ SelectorValve4	5e-10	5
GreenPump1/Valve1/BluePump1/Valve1	1e-8	16
GreenPump2/Valve2/BluePump2/Valve2	2e-8	8
GreenPump3/Valve3/BluePump3/Valve3	3e-8	4
GreenPump4/Valve4/BluePump4/Valve4	4e-8	2

In general, there are 4 potential component architectures for each of these 8 components. The size of the design space to be explored is therefore 4^8 possible configurations. As such it is very difficult to do this kind of optimisation manually.

6.5 Optimisation results

Based on these parameters the multi-objective optimization problem is to minimize both system risk and cost.

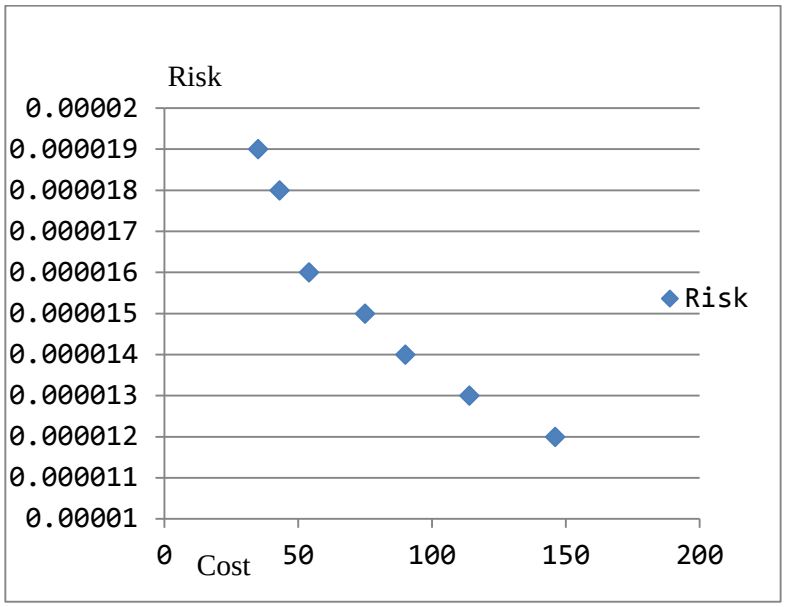

Figure 6.5 The Pareto front optimal solutions

Figure 6.5 shows the optimal architectures on the Pareto front produced by HiP-HOPS. These solutions are less risky than all other more costly solutions. To obtain specific solutions from the Pareto front, the goal of the optimization was defined as:

$$\text{Risk} \leqslant 0.000015,\ \text{Cost} \leqslant 120$$

Three solutions which satisfy this constraint are presented in Table 6.2. HiP-HOPS uses its optimisation algorithm to search and calculate each candidate's characteristics such as the cost and risk vales and finally generates those solutions that meet the defined goals. The configuration shows the combination of component alternatives selected for the solutions.

Table 6.2 The three solutions which satisfy the constraint: Risk ≤0.000015, Cost ≤120

Component	Solution 1	Solution 2	Solution 3
BSCU	BSCU2	BSCU2	BSCU1
BluePump	BluePump2	BluePump2	BluePump2
BlueValve	BlueValve3	BlueValve1	BlueValve3
CMD/ASMeterValveB	CMDASMeterValveB1	CMDASMeterValveB4	CMDASMeterValveB2
CMD/ASMeterValveG	CMDASMeterValveG3	CMDASMeterValveG1	CMDASMeterValveG3
GreenPump	GreenPump3	GreenPump3	GreenPump1
GreenValve	GreenValve2	GreenValve3	GreenValve3
SelectorValve	SelectorValve3	SelectorValve2	SelectorValve2
Cost	74	90	114
Risk	0.000015	0.000014	0.000013

The various design solutions shows different potential configurations of components to achieve the pre-defined risk and cost restrictions. BSCU and SelectorValve are highly critical components and therefore should be robust. This is illustrated by how Solution 3, which has the lowest risk among the three selected sample solutions within the restricted cost, employs the BSCU1 and SelectorValve2. The results presented here represent a preliminary step in the overall safety assessment process. The multi-objective assessment routine can be performed iteratively by adjusting design parameters (risk and cost) until requirements are met in the process of an evolving design. The optimization is automated and therefore can be repeated efficiently in the course of design iterations.

6.6 Summary

In this chapter, an aircraft wheel brake system was used as a case study to illustrate and validate the transformations between AADL dependability and optimisation model to HiP-HOPS optimisation model. The AADL dependability and optimisation model of the aircraft wheel brake system is obtained by extending the system dependability model through adding newly defined AADL optimisation properties to both system and component levels. Also, the alternatives for each component in the system are added. The AADL dependability and optimisation model of the aircraft wheel brake system is then transformed into the HiP-HOPS optimisation model.

The correctness of the transformation is validated by comparing the obtained fault tree (i.e. generated from the transformation of the AADL dependability model to HiP-HOPS dependability model) with the validated fault tree published in Sharvia, (2011). The alternatives

of each component is checked to ensure that all the characteristics e.g. failure data, cost that are related to the alternatives are transformed correctly.

HiP-HOPS uses this optimisation model to search and obtain the optimal architectures on Pareto front. The Pareto front optimisation solutions shows the combination of component alternatives selected for the solutions. This allows the AADL designers to select, among a large number of potential design spaces, the solutions that satisfy both dependability and cost constraints. The model transformation approach is an effective way to provide such dependability and cost optimisation of AADL models.

Chapter 7 Conclusions and future work

The general research hypothesis of this thesis is that model transformation is a cost-effective way to advance model-based dependability analysis and optimisation to models expressed in AADL. The thesis has developed a model transformation method (in Chapter 3) to show that model transformation is a fundamental technique to maximise the utility of AADL because it provides a route for the exploitation of mature and tested tools in the AADL context.

Following a survey of the current system dependability analysis and optimisation methods and tools e.g. ArcheOpterix (Aleti et al., 2009) and AQOSA (Li et al., 2011, Etemaadi and Chaudron, 2012) for AADL models, limitations in the exploitation of current system optimisation methods are identified. In order to enable AADL use the existing system dependability analysis and optimisation techniques, this thesis develops a model-based dependability modelling and analysis and architecture optimisation method using model transformation.

If AADL models could be transformed into the models used by other methods and tools then it would extend the optimisation analysis that could be done on AADL models. HiP-HOPS is a state-of-the-art model-based dependability and optimisation analysis technique but HiP-HOPS requires the system to be optimised is expressed as a HiP-HOPS model using the HiP-HOPS modelling language. This problem can be overcome by transforming the AADL model into an equivalent HiP-HOPS model. In this thesis, a model transformation method has been devised and implemented for automatic dependability and cost optimisation of AADL dependable models.

The thesis presents transformation rules for transforming an AADL model which contains an error state machine into a HiP-HOPS fault tree model. The transformation rules have been implemented in the ATL language. The ATL language is selected as a transformation language in this research not only because it hides the complexity of model transformation behind the simple syntax but also because it is a hybrid model-to-model transformation language that uses declarative rules and imperative rules which makes the language suitable to implement the transformations method. The thesis has shown the feasibility of the automation of the transformation method by implementing the tool AADL2HiP-HOPS, which has been integrated as a plug-in into the AADL development environment OSATE.

The AADL2HiP-HOPS tool has been developed for the automatic generation of target HiP-HOPS-oriented dependability analytical models from high level source AADL dependable models. AADL is used as the notation for capturing the system architecture model and the

AADL Error Model Annex is used to capture the component faults and failure modes. The method transforms this AADL dependability model to a HiP-HOPS model which is then used for synthesis of fault trees, FMEAs and other analyses to automatically generate the fault tree and FMEA table of system for further dependability analysis.

AADL and HiP-HOPS use different modelling concepts and hence have different models and different modelling languages. There are similarities and differences in the AADL and HiP-HOPS modelling concepts. Both languages use the concepts of component, port and connection although detailed semantics differ. To ensure the semantics between the source and target model remain unchanged during the transformation, the mappings between the source and target model are comprehensively analysed. The transformation mappings are driven by considering the information needed from the target HiP-HOPS model and the location of that information in the source AADL model.

The transformation from AADL to HiP-HOPS consists of two parts. The first part is the transformation of the component error models into HiP-HOPS individual component fault trees. A state machine to fault-tree conversion algorithm is used for this part. The second part is the transformation of the AADL connections into HiP-HOPS Lines. Although, a transformation between source and target models is semantics preserving, the target model may represent in an explicit property, properties that are implicit in the source model. An example of this is the HiP-HOPS Line object which models the propagation of failure events. Failure propagation information is available in the AADL source model but is not so explicitly represented. In particular, the information in the guard_in(out) expressions as well and the AADL connections is required in order to create the corresponding HiP-HOPS Line objects.

In order to make the developed transformation method more reusable and adaptable, the thesis also discusses the guidance for the AADL to HiP-HOPS model transformation design. The guidance discusses some of the modularisation techniques in rule-based transformation languages. The ATL transformation language is used to illustrate the transformation design. Usually, different sets of transformation rules can be derived from different decompositions in the source and target metamodels and hence, the thesis proposes that the designers consider multiple decompositions in the metamodels. Moreover, the thesis considers the appropriate use of the three rule integration mechanisms, i.e. implicit rule calls, explicit rule calls, and rule inheritance that can be used to integrate rules together. Each mechanism has its advantages and disadvantages to integrate rules together. Implicit rule calls relies on indirect rule dependencies and usually lead to a low coupling between rules. It thus can be chosen when adaptability and reusability are set as the main quality attribute of the transformation definitions. In the context of this research, explicit rule calls is best for error state machine to fault tree conversion. This is

because ATL rules take input from sources of different types i.e. they are source type sensitive. The error state machine to fault tree conversion is not a source type sensitive transformation. For this reason, imperative code is used to implement the transformation needed to explicitly generate the target model element (i.e. fault tree failureExpression).

One challenge for the optimisation of AADL models is how to enable the representation of "variability" in the AADL system. Variability is a prerequisite for optimisation, because it creates the design space of alternatives which needs to be explored in order to seek the best architectural solutions. Variability includes the possibility of designating one or more alternatives to a given component or subsystem. Clearly, for automated optimisation, the variability must be constrained. In the optimisation method considered in this thesis, component and subsystems may be associated with one or more alternatives with respect to dependability and cost. The optimisation process searches the large space of possible designs defined by the combinations of possible choices, and uses optimisation heuristics such as genetic algorithms to obtain optimal or near optimum designs.

In AADL, however, there is no scheme for directly modelling alternatives and optimisation parameters. The thesis has discussed the problem of defining variability in AADL and has developed an optimisation modelling method which allows the AADL designer to specify variable elements of the system model. The developed optimisation modelling method extends the AADL modelling features by adding an optimisation modelling scheme. This includes a high level way to define the alternatives for a component and optimisation needed properties such as component cost, system optimisation objectives. Explicitly specifying alternative implementations gives the designer the flexibility to add or remove component specific optimisations as needed. The benefit of this developed optimisation modelling method is that it supplies designers an open path to design their own specific optimisation properties with lower load. The application of this optimisation modelling method lead to a AADL dependability and optimisation model, which is then transformed to a HiP-HOPS optimisation model for dependability and cost optimisation analysis.

The model transformation approach has been illustrated and tested by applying it onto several case studies. The model transformation approach has been shown to be an effective way to provide fault tree and FMEA analysis and optimisation analysis of AADL models. The direct benefit of the transformation presented in this thesis is that it opens a path that will enable the AADL language to take advantage of an existing dependability analysis and optimisation technique. The technique may be used early in the design and makes the analysis of complex dependable systems practical and cost-effective.

Beyond this research, future work may focus on:

1. Dependability analysis and optimisation of AADL systems that have temporal ordering characteristics.

HiP-HOPS Pandora technology (Walker and Papadopoulos, 2006) can be used to implement the dependability analysis for AADL models with temporal ordering characteristics. Future work could adapt and test the transformation method based on a case study to make the transformation method capable to transform AADL model (with temporal ordering characteristics) into temporal fault trees.

2. This research focuses on the static architecture of AADL models. The static architecture is represented by a collection of components that are structured in a component hierarchy. For each component in the hierarchy, its component implementation defines the internal structure in terms of subcomponents and interactions in terms of connections. The dynamic architecture is represented by a collection of component and connection configurations that are controlled by modes (Feiler and Gluch, 2012). Due to the constraints imposed by AADL Error Model Annex, certain types of failure modes cannot be specified in the error model and one challenge in this research (as encountered in (Joshi et al., 2007)) is the failure modes are not considered since they cannot be obtained in the error model. Future work should consider failure modes separately.

3. More detailed cost modelling

The cost attribute in the optimisation process is assumed directly as a cost value. The total cost for all of the components included in a design solution is simply the sum of each component's cost. In reality, however, there are situations where more complicated cost calculations are required. For example, a discount may be possible if several repeated components are selected in the same architecture design. Thus, in this situation, the actual total cost should be calculated as the total cost of these repeated components subtract the discount. Maintenance cost is also a detail that should be included. This means that the system cost should be calculated over a given lifetime.

4. Future work could focus on improving the modularity of the transformation so that it can be more easily re-targeted to different optimisation tools. Furthermore, a library of alternatives can be built for each type of component in AADL so that once the designer choose one type of component then the alternatives of this particular component type can be automatically selected for optimisation. Future work may also consider new techniques for describing model variability. In addition to replacing a single component with an alternative component, a

designer may wish to introduce other replacement patterns. For example, a replacement pattern may require that two matching components are always replaced as a pair.

References

Adachi M, Papadopoulos Y, Sharvia S, Parker D, and Tohdo T, 2011, An approach to optimization of fault tolerant architectures using HiP-HOPS, *Software Practice and Experience*, 41(11), pp 1303-1327.

Aleti A, Bjornander S, Grunske L, Meedeniya I, 2009, ArcheOpterix: An extendable tool for architecture optimization of AADL models, *Proceedings of the 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pp 61-71.

Aleti A, Buhnova B, Grunske L, Koziolek A, Meedeniya I, 2012, Software architecture optimization methods: a systematic literature review, *IEEE Transactions on Software Engineering (September (99))*, ISSN: 0098-5589.

AltaRica project, 2014, Available: http://altarica.labri.fr/wp/http://compass.informatik.rwth-aachen.de/ [Accessed 18 July 2014].

ARP 4761, 1996, Aerospace recommended practice: guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment, Society of Automotive Engineering, Warrendale, PA, Tech. Rep.

ASSERT project, 2014, Available: http://www.esa.int/TEC/Software_engineering_and_standardisation/TECJQ9UXBQE_0.html [Accessed 21 July 2014].

ATL/User Guide, 2012, ATL/User Guide - The ATL Language, [online], Available: http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language [Accessed 18 February 2014].

ATLAS group, 2005, ATL: Atlas Transformation Language, ATL Starter's Guide.

AUTOSAR, 2014, Available: http://www.autosar.org [Accessed 18 July 2014].

Avizienis A, Laprie J C and Randell B, 2001, Fundamental Concepts of Dependability, LAAS Report no. 01-145 [online], Available: http://www.cs.cmu.edu/~garlan/17811/Readings/avizienis01_fund_concp_depend.pdf [Accessed 01 October 2013].

Baudry B, Dinh-Trong T, Mottu J-M., Simmonds D, France R, Ghosh S, Fleurey F, Traon Y L, 2006, Model Transformation Testing Challenges, *In: Proceedings of IMDT workshop in conjunction with ECMDA'06*, Bilbao, Spain.

Bieber P, Castel C, and Seguin C, 2002, Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System, *In Proceedings of the 4th European Dependable Computing Conference*, pp 19 – 31, Springer-Verlag.

Biehl M, 2010, Literature Study on Model Transformations [online], Available: http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/papers/BiehlModelTransformations.pdf [Accessed 05 January 2014].

Biehl M, Chen D and Torngren M, 2010, Integrating Safety Analysis into the Model-based Development Toolchain of Automotive Embedded System, *LCTES'10*, Stockholm, Sweden.

Bozzano M and Villafiorita A., 2003, Improving System Reliability via Model Checking: the FSAP / NuSMV-SA Safety Analysis Platform, *In SAFECOMP*, pp 49–62, Edinburgh.

Chen D, Feng L, Lonn H and Tolvanen J P, 2013, EAST-ADL and automotive system modelling (Part 3) [online], Available: http://www.eetindia.co.in/STATIC/PDF/201306/EEIOL_2013JUN06_EMS_TA_01.pdf?SOURCES=DOWNLOAD [Accessed 17 March 2014].

COMPASS project, 2013, Available: http://compass.informatik.rwth-aachen.de/ [Accessed 18 July 2014].

Czarnecki K and Helsen S, 2003, Feature-based survey of model transformation approaches, *IBM Systems Journal*, 45(3), pp 621-645.

Czarnecki K and Helsen S, 2006, Classification of Model Transformation Approaches, *OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, Anaheim, USA*.

Dehlinger J and Dugan J B, 2008, Analyzing Dynamic Fault Trees Derived from Model-Based System Architectures, *Nuclear Engineering And Technology* 40(5).

Del Fabro M D and Valduriez P, 2009, Towards the efficient development of model transformations using model weaving and matching transformations, *Software and Systems Modeling* 8(3), pp 305-324.

Dugan J B, Pai G and Xu H, 2007, Combining Software Quality Analysis with Dynamic Event/Fault Trees for High Assurance Systems Engineering, *In Proceedings 10th IEEE High Assurance System Engineering Symposium*, pp 245-255, Dallas, TX.

EAST-ADL Association, 2013 (b), EAST-ADL Domain Model Specification, Version V2.1.11 [online], Available: http://www.east-adl.info/Specification/V2.1.11/EAST-ADL-Specification_V2.1.11.pdf [Accessed 17 March 2014].

EAST-ADL Association, 2013(a), About EAST-ADL Association, Available: http://www.east-adl.info [Accessed 18 July 2014].

Etemaadi R, Chaudron, M R V, 2012, A model-based tool for automated quality driven design of system architectures, *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA'12)*, Lyngby, Denmark.

Feiler P and Gluch D, 2012, *Model-Based Engineering with AADL-An Introduction to the SAE Architecture Analysis & Design Language*, Pearson Education, USA.

Feiler P and Rugina A, 2007, Dependability Modelling with the Architecture Analysis & Design Language (AADL), *Report prepared for the SEI Administrative Agent*, CMU/SEI-2007-TN-043.

Feiler P, 2010, Software-reliant System Validation with AADL [online], Available: https://wiki.sei.cmu.edu/aadl/images/f/fa/ToolIntegrationinAADL-June2010.pdf [Accessed 15 December 2013].

Feiler P, Gluch D and Hudak J, 2006, The Architecture Analysis & Design Language (AADL): An Introduction [online], Available: http://www.sei.cmu.edu/reports/06tn011.pdf [Accessed 25 November 2010].

Fenelon P and McDermid J A, 1993, An integrated toolset for software safety analysis, *Journal of Systems and Software*, 21(3) pp 279–290.

Fenelon P, McDermid J A, Nicholson M, Pumfrey D J , 1994, Towards integrated safety analysis and design, *ACM SIGAPP Applied Computing Review - Special issue on saftey-critical software*, 2(1), pp 21-32.

Giese H, Tichy M, and Schilling D, 2004, Compositional Hazard Analysis of UML Component and Deployment Models, *In Computer Safety, Reliability, and Security*, volume 3219 of LNCS, pp 166–179, Springer.

Granske L and Han J, 2008, A comparative study into architecture-based safety evaluation methodologies using AADL's error annex and failure propagation models, *11th IEEE High Assurance Systems Engineering Symposium*.

Grunske L and Kaiser B, 2005, Automatic Generation of Analyzable Failure Propagation Models from Component-level Failure Annotations, *QSIC*, pp 117–123.

Grunske L, Lindsay P, Bondarev E, Papadopoulos Y, Parker D, 2007, An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems, In: de Lemos, R., et al. (Eds.), *Architecting Dependable Systems IV*, LNCS, vol. 4615, pp 188–209.

Hamann R, Uhlig A, Papadopoulos Y, Rüde E, Grätz U, Walker M and et al., 2008, Semi Automatic Failure Analysis Based on Simulation Models, *The ASME 27th International Conference on Offshore Mechanics and Arctic Engineering OMAE2008*, 15-20 June, Estoril.

Hecht M, Lam A, Howes R and Vogl C, 2010, Automated Generation of Failure Modes and Effects Analyses from AADL Architecturral and Error Models [online], Available: https://wiki.sei.cmu.edu/aadl/images/0/06/HechtFaultModelingMay2010.pdf [Accessed 15 December 2013].

Hecht M, Vogl C and Lam A, 2009, Application of the Architectural Analysis and Design Language (AADL) for Quantitative System Reliability and Availability Modeling [online], Available: https://wiki.sei.cmu.edu/aadl/images/7/78/Vogl_Hecht_Lam_Aerotech_09.pdf [Accessed 15 December 2013].

IMCA, 2002, Guidance on Failure Modes & Effects Analyses (FMEAs), Report IMCA M 166 [online], Available: http://www.imca-int.com/media/73361/imcam166.pdf [Accessed 10 October 2013].

Johnsen A and Lundqvist K, 2011, Developing Dependable Soft-ware-Intensive Systems: AADL vs. EAST-ADL, (Eds.): Ada-Europe 2011, LNCS 6652, pp 103-117, Springer-Verlag.

Joshi A and Heimdahl M P, 2005, Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier, *In SAFECOMP*, volume 3688 of LNCS, pp 122–135, Springer-Verlag.

Joshi A and Heimdahl M P, 2007, Behavioral Fault Modeling for Model-based Safety Analysis, *Proceedings of 10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, Dallas.

Joshi A, Heimdahl M P, Miller S, and Whalen M, 2006, Model-Based Safety Analysis, *NASA contractor report*, NASA/CR-2006-213953.

Joshi A, Miller S, Whalen M, and Heimdahl M P, 2005, A Proposal for Model-Based Safety Analysis, *In Proceedings of 24th DASC*.

Joshi A, Vestal S and Binns P, 2007, Automatic Generation of Static Fault Trees from AADL Models, *In DSN Workshop on Architecting Dependable Systems*, DSN07-WADS, Edinburgh, Scotland-UK.

Jouault F and Kurtev I, 2005, Transforming models with ATL, Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica [online], Available: http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev__transforming_models_with_atl.pdf [Accessed 10 October 2013].

Jouault F, Allilaire F, Bezivin J, Kurtev I, 2008, ATL: A model transformation tool, *Science of computer programming*, volume (72), pp 31-39.

Kaiser B, Gramlich C, and orster M F, 2007 State/event fault trees–a safety analysis model for software-controlled systems, *Reliability Engineering & System Safety*, 92(11), pp 1521–1537.

Kaiser B, Liggesmeyer P, and ackel O M, 2003, A new component concept for fault trees, *In Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software (SCS'03)*, pp 37-46, Adelaide.

Konak A, Coit D W, Smith A E, 2006, Multi-objective optimization using genetic algorithms, *Reliability Engineering & System Safety* 91 (9), pp 992-1007.

Kurtev I, Berg K and Jouault F, 2007, Ruled-based modularization in model transformation languages illustrated with ATL, *Science of Computer Programming*, 68(2007), pp 138-154.

Küster J M, 2004, Systematic Validation of Model Transformations *In: WiSME 2004 (associated to UML 2004)*, Lisbon, Portugal.

Küster J M, 2006, Definition and validation of model transformations, *Software and Systems Modeling* 5(3), pp 233 - 259.

LABRI, 2014, Available: http://www.labri.fr/ [Accessed 18 July 2014].

Lawley M, Duddy K, Gerber A and Raymond K, 2004, Language Features for Re-Use and Maintainability of MDA Transformations, *In: OOPSLA workshop on Best Practices for Model-*

Driven Software Development, Vancouver, Canada [online], Available: http://s23m.com/oopsla2004/duddy.pdf [Accessed 28 July 2014].

Li R, Etemaadi, R, Emmerich, M T M, Chaudron, M R V, 2011, Automated Design of Software Architectures for Embedded Systems using Evolutionary Multiobjective Optimization, *Proc. of the VII ALIO/EURO*.

MAENAD project, 2013, EAST-ADL Domain Model Specification, version V2.1.11 [online], Available: http://east-adl.info/Specification/V2.1.11/EAST-ADL-Specification_V2.1.11.pdf [Accessed 30 September 2013].

Mahmud N and Mian Z, 2013, Automatic Generation of Temporal Fault Trees from AADL Models, *European Safety and Reliability conference (ESREL 2013)*, Amsterdam.

Mahmud N, 2012, Dynamic model-based safety analysis: From state machines to temporal fault trees, PhD dissertation, Department of Computer Science, University of Hull, UK.

Mahmud N, Papadopoulos Y and Walker M, 2010, A translation of State Machines to temporal fault trees, *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Chicago, USA, pp 45-51.

Mahmud N, Walker M, and Papadopoulos Y, 2011, Compositional synthesis of temporal fault trees from state machines, *The six international conference on Availability, Reliability and Security*, pp 429-435.

Medvidovic N and Taylor R N, 2000, A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering 26*, pp 70-93.

Meedeniya I, Aleti A, Bühnova B, 2009, Redundancy allocation in automotive systems using multi-objective optimisation, *Symposium of Avionics/Automotive Systems Engineering (SAASE'09)*, San Diego.

Mens T and Van Gorp P, 2006, A taxonomy of model transformation, *Electronic Notes in Theoretical Computer Science*, vol. 152, pp 125 - 142.

Mian Z and Bottaci L, 2013, Multi-objective Architecture Optimisation Modelling for Dependable Systems, *the 4th IFAC Workshop on Dependable Control of Discrete Systems (DCDS2013)*, York University, UK.

Mian Z, Bottaci L and Papadopoulos Y, 2013b, Multi-objective Architecture Optimisation for Dependable Systems, extended abstract for the *3rd International Workshop on Model Based Safety Assessment, IWMBSA'2013*, Versailles, France.

Mian Z, Bottaci L, Papadopoulos Y and Adachi M, 2013a, Multi-objective Architecture Optimisation for Dependable Systems, *Reliability Engineering & System Safety Journal*, Elsevier, Accepted.

Mian Z, Bottaci L, Papadopoulos Y and Biehl M, 2012, System Dependability Modelling and Analysis Using AADL and HiP-HOPS, *14th IFAC Symposium on Information Control Problems in Manufacturing*, Bucharest, Romania.

Mian Z, Bottaci L, Papadopoulos Y, Sharvia S and Mahmud N, 2013c, Model Transformation for Multi-objective Architecture Optimisation of Dependable Systems, In Zamojski W (Ed.) Dependability problems of complex information systems, *Springer Verlag*, Accepted.

Mobius, 2014, Möbius Overview, Available: http://www.mobius.illinois.edu/ [Accessed 18 July 2014].

Nicol D M, Sanders W H and Trivedi K S, 2004, Model-Based Evaluation: From Dependability to Security, *IEEE Transactions on Dependable and Secure Computing*, 1(1), pp 48-65.

OMG, 2005, Introduction To OMG's Unified Modelling Language (UML) [online], Available: http://www.omg.org/gettingstarted/what_is_uml.htm [Accessed 30 September 2013].

OMG, 2012, OMG Systems Modeling Language (OMG SysML), version 1.3 [online], Available: http://www.omg.org/spec/SysML/1.3/ [Accessed 30 September 2013].

Papadopoulos Y and Grante C, 2005, Evolving car designs using model-based automated safety analysis and optimisation techniques, *The Journal of Systems and Software*, 76(1), pp 77-89.

Papadopoulos Y and Maruhn M, 2001, Model-based synthesis of fault trees from matlab-simulink models, *In 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, pp 77–82, IEEE Computer Society.

Papadopoulos Y, Parker D, and Grante C, 2004, Automating the failure modes and effects analysis of safety critical systems, *In Int. Symp. on High-Assurance Systems Engineering (HASE 2004)*, pp 310–311, IEEE Computer Society.

Parker D J, 2010, Multi-objective Optimisation of Safety-Critical Hierarchical Systems, PhD dissertation, The University of Hull.

Popic P, 2005, The Impact of Error Propagation on Software Reliability Analysis of Component-based Systems, PhD thesis, The West Virginia University.

Rauzy A, 2002, Mode automata and their compilation into fault trees, *Rel. Eng. & Sys. Safety (RESS)* 78(1), pp 1-12.

Rouvroye J L and Bliek E G V D, 2002, Comparing safety analysis techniques, *Reliability Engineering and System Safety*, 75(2002), pp 289-294.

Rugina A E, 2007, Dependability modelling and evaluation - From AADL to stochastic Petri nets, PhD dissertation, LAAS/CNRS.

Rugina A E, Kanoun K and Kaâniche M, 2007, An Architecture-based Dependability Modelling Framework Using AADL, *10th IASTED International Conference on Software Engineering and Applications (SEA'2006), Dallas (USA)*, pp 222-227.

Rugina, A E, Kanoun K and Kaaniche M, 2008, The adapt tool: From aadl architectural models to stochastic petri nets through model transformation, *In Proc. 7th European Dependable Computing Conf. (EDCC)*, Kaunas, Lituanie.

SAE-AADL Meta Model/XMI V0.999, 2006, Annex C AADL Meta Model and Interchange Formats Normative [online], Available: http://aadl.sei.cmu.edu/aadlinfosite/downloads/AADLmetamodel/AADLmetamodel04102006.zip [Accessed 06 December 2010].

SAE-AS5506, 2006, Architecture Analysis and Design Language (AADL), Society of Automotive Engineers (SAE).

SAE-AS5506/1, 2006, Architecture Analysis and Design Language Annex Volume 1, Annex E: Error Model Annex, Society of Automotive Engineers (SAE).

SAE-AS5506/2, 2011, Architecture Analysis and Design Language (AADL) Annex Volume 2, Annex D: Behaviour Annex, Society of Automotive Engineers (SAE).

SEI, 2004, OSATE: An extensible Source AADL Tool Environment, *SEI AADL Team technical Report*.

Sharvia S, 2011, Integrated Application of Compositional and Behavioural Safety Analysis, PhD dissertation, The University of Hull.

Steinberg D, Budinsky F, Paternostro M, and Merks E, 2009, EMF: *Eclipse Modeling framework*, Pearson Education, Boston, USA.

Trivedi K S, 2001, Probability and Statistics with Reliability, Queuing, and Computer Science Applications, John Wiley and Sons, New York, Second Edition, ISBN 0-471-33341-7.

Vesely W, Stamatelatos M, Dugan J, Fragola J, Minarick J and Railsback J, 2002, Fualt Tree Handbook with Aerospace Applications, *NASA Office of Safety and Mission Assurance*, USA.

Walker M and Papadopoulos Y, 2006, Pandora: The Time of Priority-AND gates, *12th IFAC Symposium on Information Control Problems in Manufacturing*, St Etienne, France, pp 237-242.

Walker M, and Papadopoulos Y, 2009, Qualitative temporal analysis: towards a full implementation of the Fault Tree Handbook, *Control Engineering Practice 17(10)*, pp 1115–1125, ISSN 0967 0661.

Walker M, Reiser M-O, Tucci-Piergiovanni S, Papadopoulos Y, Lönn H, Mraidha Ch, Parker D, Chen D-J, Servat D, 2013, Automatic optimisation of system architectures using EAST-ADL, *Journal of Systems and Software*, 86(10), pp 2467-2487.

Wallace M, 2005, Modular architectural representation and analysis of fault propagation and transformation, *Electr. Notes Theor. Comput. Sci.*, 141(3), pp 53–71.

XMI, 2003, XML Meta Interchange (XMI), Version 2.0, Object Management Group (OMG) [online], Available: http://www.omg.org/technology/documents/formal/xmi.htm [Accessed 06 March 2011].

XML, 2001, XML Schema, Version 1.0, World Wide Web Consortium (W3C), [online], Available: http://www.w3.org/XML/Schema [Accessed 06 March 2011].

Yang W, Horwitz S, and Reps T, 1992, A program integration algorithm that accommodates semantics-preserving transformations, *ACM Transactions on Software Engineering and Methodology*, 1 (3), pp 310 - 354.