THE UNIVERSITY OF HULL

Integrated Application of Compositional and Behavioural Safety Analysis

being a Thesis submitted for the Degree of

Doctor of Philosophy

in the University of Hull

by

Septavera Sharvia BSc (Hons) MSc

February 2011

For my parents

Abstract

To address challenges arising in the safety assessment of critical engineering systems, research has recently focused on automating the synthesis of predictive models of system failure from design representations. In one approach, known as compositional safety analysis, system failure models such as fault trees and Failure Modes and Effects Analyses (FMEAs) are constructed from component failure models using a process of composition. Another approach has looked into automating system safety analysis via application of formal verification techniques such as model checking on behavioural models of the system represented as state automata. So far, compositional safety analysis and formal verification have been developed separately and seen as two competing paradigms to the problem of model-based safety analysis.

This thesis shows that it is possible to move forward the terms of this debate and use the two paradigms synergistically in the context of an advanced safety assessment process. The thesis develops a systematic approach in which compositional safety analysis provides the basis for the systematic construction and refinement of state-automata that record the transition of a system from normal to degraded and failed states. These state automata can be further enhanced and then be model-checked to verify the satisfaction of safety properties. Note that the development of such models in current practice is ad hoc and relies only on expert knowledge, but it being rationalised and systematised in the proposed approach – a key contribution of this thesis.

Overall the approach combines the advantages of compositional safety analysis such as simplicity, efficiency and scalability, with the benefits of formal verification such as the ability for automated verification of safety requirements on dynamic models of the system, and leads to an improved model-based safety analysis process. In the context of this process, a novel generic mechanism is also proposed for modelling the detectability of errors which typically arise as a result of component faults and then propagate through the architecture. This mechanism is used to derive analyses that can aid decisions on appropriate detection and recovery mechanisms in the system model.

The thesis starts with an investigation of the potential for useful integration of compositional and formal safety analysis techniques. The approach is then developed in detail and guidelines for analysis and refinement of system models are given. Finally,

the process is evaluated in three cases studies that were iteratively performed on increasingly refined and improved models of aircraft and automotive braking and cruise control systems. In the light of the results of these studies, the thesis concludes that integration of compositional and formal safety analysis techniques is feasible and potentially useful in the design of safety critical systems.

Acknowledgement

I am deeply indebted, first and foremost, to Dr Yiannis Papadopoulos. I could not have wished for a better supervisor and mentor. Without his constant support, continuous patience, and invaluable guidance along the way, this would not have been possible. I have been extremely lucky to work under his supervision, and I will always be grateful for the opportunity to embark on this journey.

I am also grateful to Dr Chandra Kambhampati and Dr Leonardo Bottaci, whom have helped improved the project with their guidance and feedback.

Special thanks to Dr Martin Walker for all his help throughout the years. His patience in revising the draft is very much appreciated, and I am truly grateful. Huge thanks to Dr David Parker for his work on HiP-HOPS and for the huge help in Consensus integration. I am also thankful to Dr Ian Wolforth for all his help, advice and friendship. Many thanks also to my colleagues and friends Nidhal Mahmud, Amer Dheedan, Shawulu Nggada, Mian Zhi Bao and Ernest Edifor. Their friendship (and polo mints and biscuits) have made life in the lab a good fun.

I would also like to thank Yvonne Tang, Dr Chan Kuan Yoow, Chin Hau Khor, Ash D'Souza, Jessica Yeo, Sabrina Cheong, Gabrielle Lo, Daniel Low, Robert Chua, and Kuan Siew Fong for their friendship, company, good time and good food.

Last but not least, my deepest thanks go to my mother, Helmina, and my father, Darwis. Their endless support and sacrifice are my source of strength and inspiration. I am also sending my love to my talented little sister, Dora Augustin. I am a very proud sister.

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. Specifically:

Much of the material in Chapters 3 & 5 was presented in (Sharvia & Papadopoulos, 2009). An extended version of the above paper was published as book chapter in (Sharvia & Papadopoulos, 2010). The material presented in chapter 6 was presented & published in (Sharvia & Papadopoulos, 2008), and contributed to (Adachi *et al.*, 2010).

Contents

CHAPTER 1. Introduction 17

1.1 Research Context and Scope 17

1.2 Research Motivation 20

1.3 Research Hypothesis 21

1.4 Research Objectives 21

1.5 Structure of Thesis 23

1.6 List of Publications 24

CHAPTER 2. Safety Analysis for Complex Safety-critical Embedded System 26

2.1 Modelling and Specifications 26

2.2 Early Functional Design 27

2.2.1 Functional Analysis 28

2.2.2 Functional Hazard Assessment (FHA) 30

2.2.3 Functional Failure Analysis (FFA) 31

2.2.4 Preliminary Systems Safety Analysis (PSSA) 31

2.3 Classical Safety Analysis 32

2.3.1 Fault Tree Analysis 32

2.3.2 Failure Mode and Effects Analysis 33

2.4 Compositional Safety Analysis 33

2.4.1 Component Fault Tree 34

2.4.2 State Event Fault Trees 35

2.4.3 Embedded Systems Safety and Reliability Analyser 36

2.4.4 Hierarchically Performed Hazard Origin and Propagation Studies 38

2.4.5 Summary of CSA Techniques 40

2.5 Behavioural Safety Analysis 40

2.5.1 Introduction to Model checking 41

2.5.2 FSAP/NuSMV-SA 44

2.5.3 ALTARICA 47

2.5.4 Summary of BSA Techniques 51

2.6 Relevant Work on Other Integrated Approaches 51

2.7 Chapter Summary 53

CHAPTER 3. A method for Integrated Application of Compositional and Behavioural Safety Analysis (IACoB) 54

3.1 Introduction 54

3.2 Functional Model 58

3.3 Severity Assessment of Output Function 60

3.4 Local Failure Behaviour 62

3.5 Fault Tree and FMEA Synthesis and Analysis 64

3.6 Generation of State machines and Their Translation into Model Checker Input Language 68

3.6.1 Modelling the Dynamic Nominal Behaviour of a System 71

3.6.2 Modelling the Dynamic Failure Behaviour of a System 81

3.6.3 Translation of FMEA Results to an Abstract State Machine 87

3.6.4 Translation of HiP-HOPS Failure Annotations to a Refined State Machine 90

3.6.5 Refinement of Events to Maintain Traceability 92

3.7 Application of Model Checking 104

3.7.1 Reachability 104

3.7.2 Safety 105
3.7.3 Liveness 105
3.7.4 Fairness 106
3.7.5 Common Errors Discovered Through Model Checking 106
3.8 Potential for Automation 107
3.9 Chapter Summary 110
CHAPTER 4. Case Study on Brake-by-wire 112
4.1 Introduction to Brake-By-Wire System 112
4.2 Analysis of System Functional Models 115
4.2.1 FFA 117
4.2.2 FMEA 119
4.2.3 Construction of Mode charts 127
4.2.4 Requirement Verification 132
4.2.5 Refinement of Transition Events 135
4.3 Architecture-allocated Functional Model 143
4.3.1 Analysis of Single functional failure 145
4.3.2 Analysis of multiple functional failures 147
4.4 Chapter Summary 159
CHAPTER 5. Case Study on Aircraft Wheel Brake System 160
5.1 Introduction to Aircraft Wheel Brake System 160
5.1.1 Nominal system model 161
5.2 FTA/FMEA 164
5.3 Revised Model 166
5.4 Construction of Mode charts 169

5.5 Model Design Evolution from Requirement Verifications 171
5.6 Chapter Summary 176
CHAPTER 6. Detectability 177
6.1 Detectability in FMEA 177
6.2 Detection and Response to Failures 179
6.3 General Modelling of Detectability 180
6.4 General Analysis of Detectability 183
6.5 Example 185
6.5.1 Cruise Control System 185
6.5.2 Detectability in Cruise Control 189
6.6 Chapter Summary 196
CHAPTER 7. Conclusions 198
7.1 Contributions 198
7.2 Limitation of concepts 203
7.3 Future Work 205
APPENDIX A: Backup structure for brake-by-wire system 208
A.1. Brake Demand Input Function 208
A.2. Local Parameters Input Function 208
A.3. Vehicle Level Processing Function 209
A.4. Local Level Processing Function 209
A.5. Braking Energy Function 210
APPENDIX B: NuSMV model for brake-by-wire 211
APPENDIX C: Summary of Quantitative Analysis 220
Appendix D: Prime Implicants for Cruise Control System 222
APPENDIX E: List of Abbreviation 225
REFERENCES 227

Figures

Figure 1: Automated Model-Based Safety Analysis (adapted from Johsi *et al.*, 2006) . 19

Figure 2: Functional analysis process (summarized from FAA, 2006) 29

Figure 3: CEG in a Component Fault Tree 35

Figure 4 : SEFT fragment (source: Kaiser, 2007) 37

Figure 5: Main phases in HiP-HOPS 39

Figure 6: Commonly used temporal connectives table 43

Figure 7 : Execution tree showing next possible states 44

Figure 8: Fragment sample of NuSMV model for one-bit adder (source: Bozzano *et al.*, 2003) 45

Figure 9 : NuSMV model extended with failure mode (source: Bozzano *et al.*, 2003) . 46

Figure 10: Sample of block 49

Figure 11 : Analysis phases of Altarica 51

Figure 12: Process outline of IACoB method 56

Figure 13: System development and safety assessment process (source: ARP 4754) ... 57

Figure 14: Functional model in basic block diagrams 58

Figure 15: Example of a functional block 60

Figure 16: Connection flow between functions 60

Figure 17: Local failure behaviour for Function F1 63

Figure 18: Example of the functional architecture 65

Figure 19: Example of generated fault trees 66

Figure 20: Identified critical functions based on failure propagation..... 67

Figure 21: Generation of state machines..... 70

Figure 22: Sample of state chart..... 72

Figure 23: Status and steps in state charts semantics (source : Harel & Naamad, 1996) 74

Figure 24: Simple state machine without hierarchy 78

Figure 25: Simple state machine with hierarchy 78

Figure 26: Relationships between static and dynamic models hierarchy of the system . 79

Figure 27: Sample state chart for S1..... 80

Figure 28: Modules to model hierarchy in NuSMV 81

Figure 29: Mode charts showing high level and low level of system state transitions.. 86

Figure 30: Example of mode chart constructed from FMEA-ModeChart Assitance Table 89

Figure 31: Refinement for system A..... 94

Figure 32: Mode chart for system A..... 96

Figure 33: Refined transitions for System A..... 96

Figure 34: Mode charts for system (and subsystems of) A..... 99

Figure 35: NuSMV model for system A..... 100

Figure 36: Failure propagation for subsystem A3 101

Figure 37: Failure propagation for subsystem A4 102

Figure 38: Failure propagation for subsystem A1 102

Figure 39: Failure propagation for subsystem A2 103

Figure 40: General topology of Brake-By-Wire system..... 114

Figure 41: Abstract functional model for Brake-By-Wire..... 116

Figure 42: Redundant module for Vehicle-Level Processing function 122

Figure 43: Revised model with duplex redundant mechanism 124

Figure 44: Mode chart for Brake-By-Wire 129

Figure 45: Brake-By-Wire revised model showing Electrical and Hydraulic sources. 130

Figure 46: Updated mode chart 132

Figure 47: Modified mode chart for Brake-By-Wire..... 135

Figure 48: Fault tree for Omission of Hydraulic Failure 136

Figure 49: Expanded mode chart with minimal cut sets mapped to transition events . 137

Figure 50: Structural model of Braking Energy 138

Figure.51: Mode chart for failure behaviour in *ACTUATOR*..... 139

Figure 52: Excerpt of the NuSMV model for the Braking Energy 142

Figure 53: Architecture-allocated functional model for brake by wire system..... 144

Figure 54: Failure propagation to BCU 146

Figure 55: Updated BCU for wheels with Intentional Diagonal Locking (*DL*)..... 149

Figure 56: Fault tree for L-FL_BrakingPressure..... 150

Figure 57: Mode chart for DL Controller 152

Figure 58: Mode chart for Wheel BCU 153

Figure 59: Expanded transition based on Minimal Cut Sets..... 154

Figure 60: Mode chart for modules relating to Locking of Front Left (FL) wheel..... 155

Figure 61: Simulink model of wheel brake system 163

Figure 62: Revised model for wheel-brake system 168

Figure 63: Abstract state machine for wheel brake system..... 170

Figure 64: Revised model developed with assistance of model-checker..... 175

Figure 65: Internal malfunction in Detection Module 182

Figure 66: Event *Miss* in Detection_Module 182

Figure 67: Cruise Control System 187

Figure 68: Cruise Control with Detection Module 191

Figure 69: Cruise Control System with Fading Brake..... 194

Tables

Table 1: Example of FHA on car brake function (source: Johannessen *et al.*, 2001) 30

Table 2: Example of FFA on car brake function 31

Table 3: Allocation of severity category based on consequences to people and service (IEC-61508)..... 61

Table 4: Example of FMEA table..... 66

Table 5: FMEA-ModeChart Assistance Table 88

Table 6: Failure behaviour for System A and Subsystems A1, A2, A3, A4..... 95

Table 7: Functional failure analysis of Brake-By-Wire..... 117

Table 8: Functional blocks internal malfunctions 118

Table 9: FMEA for Basic Brake-By-Wire functions..... 119

Table 10: Direct ffects FMEA for revised model..... 125

Table 11: Further Effects FMEA for revised brake-by-wire for failure O-BasicBraking 125

Table 12: FMEA- ModeChart Assistance Table 128

Table 13: Updated FMEA-Mode chart Assistance Table..... 131

Table 14: Functional failures for single wheel braking function 147

Table 15: Internal failure for Wheel Brake System components 164

Table 16: FMEA Direct Effects for Wheel Brake System..... 165

Table 17: FMEA Further Effects for Wheel Brake System..... 165

Table 18: Detection Evaluation Criteria (Quality Associates, 1997) 178

Table 19: Example of FMEA Table Extended with Detectability information 178

Table 20: Failure information for Cruise Control functions 188

Table 21: Failure information for Detection Modules..... 192

Table 22: Failure Information for Cruise Control with Fading Brake..... 195

CHAPTER 1. Introduction

1.1 Research Context and Scope

This thesis is concerned with the integrated application of emerging safety analysis techniques for safety-critical systems.

Safety critical systems are systems whose operational deviations can potentially lead to catastrophic consequences or loss of human lives. These systems are widely employed in many industries, including the automotive, aerospace, weapons and nuclear industries. Modern safety-critical systems often incorporate numerous embedded control components, involve various engineering disciplines, and employ distributed architectures and complex communication structures. (Knight, 2002) discusses several other major challenges in safety-critical systems which include the elimination of "physical separation" due to resource sharing, and ineffective interaction between software engineering and system engineering. These characteristics present substantial challenges, and considering the consequences of failure in these systems, as well as the fact that safety critical systems have become more prevalent in everyday life, it is crucial that these systems are subjected to a rigorous safety assessment process.

Classical safety assessment techniques such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) are still employed to predict the safety of such systems. However, classical techniques are traditionally applied in a manual process, which in the context of a complex system become difficult, laborious, expensive and error-prone. For this reason, FTA and FMEA are rarely performed more than once and often at the late stage of lifecycle when the design has been finalized. This late contribution means that results from the process miss the opportunity to influence system design, potentially incurring extra cost and effort in late design modifications. Problems also arise in the lack of systematic methods to capture and manage design models and safety artefacts as in traditional practices system design models and safety analyses are often created and handled separately. With these drawbacks, classical

safety analysis techniques face tremendous challenges and are no longer deemed to be sufficiently effective in managing the rising intricacy of modern complex design.

To address some of those difficulties, recent research has been focused on investigating and developing more-effective and robust safety assessment techniques through automation of the analysis process. Model-Based Safety Analysis (MBSA) is a collective body of work which introduced semi-formal and formal models in the centre of the design and assessment process. MBSA extended the popular model-based development approach, in which effort is focused on the construction of the formal specification of the system model. This specification model is subsequently used as the foundation for various development activities like visualization, code generation, testing or prototyping (Heimdahl, 2007). Although the primary focus is placed on the development of software (digital) systems, model-based tools and techniques can also be used to model physical hardware components (for example, electrical or mechanical components).

To perform a thorough safety assessment, it is crucial to understand not only how a system behaves in its normal working condition (represented in the *nominal* model), but also in the presence of failure(s). This is done by extending the nominal model with failure information to construct the failure-augmented model, termed *fault model* (Johsi *et al.*, 2006) or *error model* (Walker *et al.*, 2008).

The automated analysis of these extended models enables various safety assessments to be performed. Such analyses typically include fault simulation and prediction of effects of failure, proof that certain safety properties hold in the model and causal safety analysis resulting in synthesis of fault trees which link causes to effects of failure. Figure 1 illustrates this point and shows the type of analyses that can be performed on a system model extended with faults in MBSA. Automated analysis of models brings substantial benefits as it lightens the burden on designers and analysts, simplifies the process, saves time and contributes to more reliable results. More importantly, it enables safety analysis to be incorporated as part of an iterative design process - as new results can be more easily generated to reflect changes – and therefore driving the design with safety in mind.

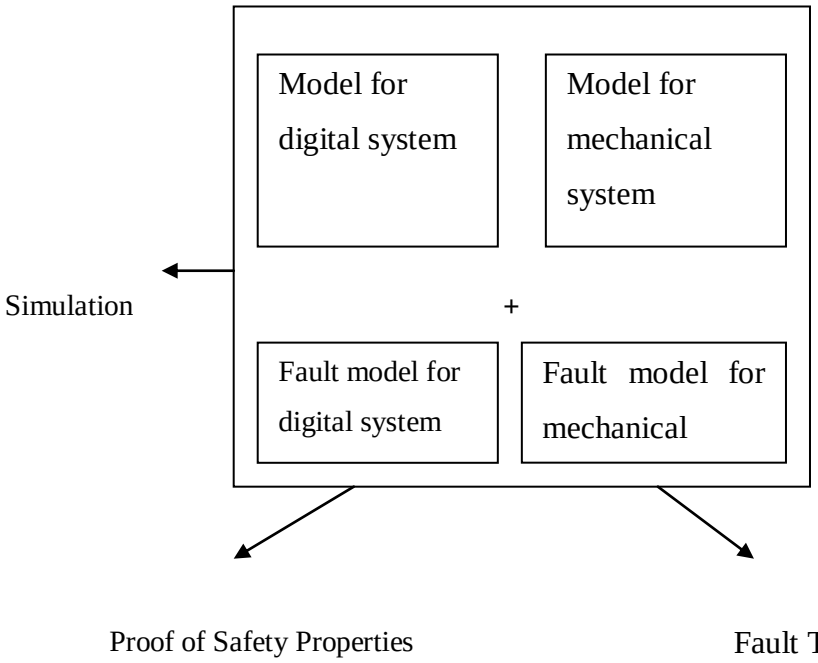


Figure 1: Automated Model-Based Safety Analysis (adapted from Johsi *et al.*, 2006)

The two most prominent paradigms of MBSA today are *Compositional Safety Analysis (CSA)* and *Behavioural Safety Analysis (BSA)*. Techniques which are based upon the CSA approach include Hierarchically Performed Hazards Origin and Propagation Studies (HiP-HOPS) (Papadopoulos & McDermid, 1999), Component Fault Trees (Kaiser *et al*., 2003), and State-Event Fault trees (SEFT) (Grunske *et al*., 2005). CSA uses a process of composition to construct system failure models from the topology of a system and local failure models of its components.

BSA, on the other hand, uses exhaustive exploration of behavioural models of the system to assess satisfaction of safety requirements. Because this approach mainly employs model checking as its primary method of assessment, the term '*model-checking based*' is often used interchangeably to characterise this type of safety analysis. A model checker typically verifies conformance of the model to its safety requirements and, if requirements are violated, it relates those violations to combinations of causes, e.g. component failures. Prominent examples based on this approach include Altarica (Arnold *et al*., 2000) and FSAP/NuSMV (Bozzano & Villafiorita, 2006).

1.2 Research Motivation

CSA and BSA techniques have emerged with little integration. Both techniques are fundamentally different in their objectives of assessment, working mechanisms, and application process. CSA is often used to facilitate reliability engineering. For example with techniques like HiP-HOPS, it is possible to effectively enable not only the identification of root failures through qualitative analysis, but also advanced probabilistic quantitative analysis. BSA on the other hand, places primary focus on the application of model checking for validation and verification of various safety properties. Also, while CSA relies largely on Boolean-based analysis, BSA explores all possible system states in brute force manner. The computationally efficient and iterative nature of CSA means that the technique can be applied from the early stages of design and on models that have a high level of abstraction. BSA on the other hand requires more mature and detailed behavioural models and is, therefore, applicable at later stages of the development process. In this thesis, it is argued that understanding these differences and exploiting each technique's strengths can bring substantial values to the development process, in particular at early design stages.

Early functional design is arguably the most appropriate phases to address design problems and take remedial measures. The volume of design information and system complexity naturally increase as system development progresses with time. The more complex the design artefact, the more difficult it is to identify problems within and the more extensive the remedy required to address problems. It is therefore best to address problems as early as possible when models are still abstract and then continue to do so as more detail is added to the design of the system.

One of the difficulties in the current industrial practice is that among classical safety analysis techniques, there is a lack of rigorous and effective techniques that can help analysis of models and identification of potential problems. (Johannessen *et al.*, 2001) highlights that "*there is still uncovered demand for early hazard analysis at functional level*", and SAE Aerospace Recommended Practices documents ARP4761 have

recommended Preliminary System Safety Assessment (PSSA) to be performed at the earliest stages together with the design activities.

1.3 Research Hypothesis

CSA and BSA have been developed as two competing paradigms in MBSA. In this thesis it is argued that the traditional gap between the two approaches can be overcome, as CSA and BSA are effectively combined in a novel model-based design and safety analysis process which therefore benefits from the advantages of both approaches, namely the flexibility, early applicability and scalability of CSA and the precision, behavioural analysis capabilities and detailed insights offered by BSA.

In the proposed process, integration of CSA and BSA is meaningful. Traditionally, early behavioural system models used in BSA are constructed in an *ad hoc* manner via human translation of textual requirements into state-machines. In the proposed process, these state-machines are largely constructed in a systematic manner driven by the results of a CSA analysis of the system. The proposed MBSA process can facilitate a more rigorous and well-rounded safety assessment at early design stages. It can therefore increase the confidence in design models before the decision is taken to progress towards refinement of the model or implementation.

1.4 Research Objectives

To test the hypothesis outlined above, the following overarching research aim has been set:

> "To develop a novel approach in which the combined application of CSA and BSA can be achieved and to evaluate the benefits and limitations of this approach using realistic examples and case studies."

To achieve this aim the following research objectives were defined:

1. To examine CSA and BSA techniques and investigate their strengths, limitations, and applications in different stages of design development. This thesis determines

complementary aspects of these techniques that can be exploited via synergistic combined application.

2. To propose a systematic technique to utilize analysis results from CSA and BSA in the course of design. This involves investigating how input to each technique can be systematically constructed, in particular, how results of CSA can assist the construction of behavioural model for BSA's formal verification. It is also important to understand how these results can provide constructive feedback to designers towards an iterative system modelling process.

3. To illustrate how a chosen CSA and a chosen BSA technique can in practice be harmonised in the context of a method for combined application. Different MBSA techniques assume different representations of failure information and system modelling. In the context of combined application, it is important to explore ways for translation of information (in particular, failure information) between relevant models. In the context of this thesis, HiP-HOPS has been selected as a representative example of CSA. NuSMV has been selected to perform symbolic model checking and enable formal verification to support BSA. The thesis shows the integration of HiP-HOPS with NuSMV and defines a process for useful semi-automatic translation of information between the two models.

4. Overall, the thesis proposes an improved approach to MBSA. The final research objective is to study the potential use of this approach in the design of mechanisms for detection and recovery from failures. More specifically, we propose a generic mechanism for modelling the *Detectability* (or NOT) of errors propagated among components of an architecture within a typical CSA. We show that the inclusion of this mechanism makes it possible to use the results of CSA as a basis for rational decisions about the inclusion of fault tolerant mechanisms in a design.

1.5 Structure of Thesis

The remainder of the thesis is structured as follows:

Chapter 2: Background

The background chapter presents an overview of modelling and safety analysis techniques. It includes a brief discussion on system modelling, and discussions on early functional model and safety assessment techniques performed at this stage (FHA, PSSA).

This chapter also discusses relevant safety analysis techniques, including those briefly mentioned in the Introduction chapter. It discusses classical techniques like FTA and FMEA, and more recent CSA developments such as HiP-HOPS, CFTs and SEFTs. In this thesis, HiP-HOPS is representative of CSA and therefore it is discussed in more detail. BSA and relevant techniques (Altarica and FSAP/NuSMV) are also presented here. This chapter also explains further the distinction between the two techniques.

Chapter 3: Integrating CSA and BSA in a unified MBSA process

This chapter describes in detail a method for combined, harmonized application of CSA and BSA techniques in the context of an improved MBSA process. HiP-HOPS and NuSMV provide two representative CSA and BSA techniques employed here. Stages involved in the method include: construction of system model from requirements, failure severity analysis, local failure behaviour annotation, translation of CSA results into the BSA model, generation of abstract state machines, and application of formal verification through model checking. This chapter also discusses how different models (and relevant failure information) can be obtained and translated between different models of CSA and BSA.

Chapter 4: Case Study on Brake-by-wire System

This chapter describes a case study on a brake-by-wire system to demonstrate the practicability and usefulness of the proposed method. Both functional and more-refined models of the system are presented. This chapter shows how CSA is effectively applied

on the early functional model and facilitates early improvement of system design. Safety artifacts from CSA are used as the basis of BSA models construction, which are then formally verified. This chapter also highlights how model checking can be used to verify a simple control & recovery procedure of diagonal-locking mechanism in car wheels.

Chapter 5: Case Study on Aircraft Wheel Brake System

This chapter describes a case study on an aircraft wheel brake system, and presents a model which was adopted from (ARP 4761, 1996). This second case study provides a second example of the feasibility of the process and demonstrates how CSA and BSA shape the development of design.

Chapter 6: Detectability Analysis

This chapter introduces and describes the concept of detectability analysis, and its role in the overall modelling and analysis of the proposed method, particularly as a part of CSA. It shows how its application can be generalized and how it can be implemented in HiP-HOPS. A small example of a cruise control system is also presented to illustrate these points.

Chapter 7: Conclusions

This chapter describes conclusions drawn from this work and gives recommendations for future work

1.6 List of Publications

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

- Sharvia, S., Papadopoulos, Y., 2009. Model-based safety analysis using compositional analysis and formal verification, *ICCSIS'09,5th Int'l Conference on Computer Science & Information Systems*, July 2009, Athens.

- Sharvia, S., Papadopoulos, Y., 2008. Non-coherent modelling in compositional safety analysis, *IFAC, 17th World Congress*, International Federation of Automatic Control, Seoul, July, 2008, published in ifac-papersonline.net.
- Adachi, M., Papadopoulos, Y., Sharvia, S., Parker, D., Tohdo, T., 2011. An approach to optimization of fault tolerant architectures using HiP-HOPS, *Software Practice and Experience*, Wiley Interscience . DOI: 10.1002/spe.1044.
- Sharvia, S., Papadopoulos, Y., 2010. Integrating compositional safety analysis and formal verification, *Strategic Advantage of Computing Information Systems in Enterprise Management*, (ed) Majid Sarrafzadeh. Volume containing revised selected papers from Int'l Conference in Computer Systems and Information Systems 2009-2010, pp. 181-201, ISBN: 978-960-6672-93-4.

CHAPTER 2. Safety Analysis for Complex Safety-critical Embedded System

This chapter provides an overview of contemporary safety analysis techniques. It focuses on CSA and BSA techniques, the two classes of model-based safety analyses that have been identified in Chapter 1.

2.1 Modelling and Specifications

During the design of a system, a set of abstract informal specifications are typically transformed into sets of progressively refined, more detailed models that can be used for the implementation, production and manufacturing of the system.

Models can be classified according to several perspectives, and different modelling notations are used to reflect the selected aspects (for example information flows, control flows, or behaviour) of the system to be represented. Sommerville (2004) points out that generally a system can be viewed through: an external perspective showing the system's context and its relationships with its environment, a behavioural perspective showing the behaviour of the system, or a structural perspective showing the system's data architecture. Through these different perspectives, various models can be developed during the design phase - for example: a data-flow model which shows how data is transferred and processed at different stages; an architectural model which shows the composing sub-systems and their interrelationships; or a stimulus/response model (also known as a state/transition model) which shows how the system reacts to internal and external events.

Models can also be distinguished according to their structure into conceptual, computational and mathematical structures (Jones & Mitchell, 1987). Conceptual models are used to capture and understand high level design concepts. Computational models provide more detailed aspects in terms of operation between participating agents. Mathematical models can define a system in terms of equations between terms, or by functions that map programs to corresponding abstract values, or by logical definitions of effects of an action to a state. While conceptual models are relatively

abstract, computational models and mathematical models usually fall into the category of more-detailed specification models, and depending on the formality of the model can be analyzed formally. Specification models usually cover device models which capture the physical part of the system.

For complex embedded systems, model-based development is becoming an increasingly popular approach for development. In model-based development (Davey, 2007), focus is placed on semi-formal or formal specifications. The term 'formal specification' is usually used interchangeably with 'formal model' in the literature and refers to models with strong mathematical foundation. Although model-based tools and techniques are primarily used to model system software components, they can also be used to model physical components. Joshi (2006) combines models containing digital components (hardware and software) with models of mechanical components (like pumps and valves) which can be extended with failure information to produce extended system models upon which various safety analysis techniques (like formal verification and fault tree analysis) can be performed.

This thesis focuses primarily on conceptual models. In particular, early design where abstract *functional models* of the system are being produced to describe functions, their dependencies and abstract behaviour. It is being increasingly recognised that safety assessment should start as early as possible to prevent expensive design iterations later on. Techniques that enable safety assessment of model that describe functional designs are therefore highly desirable (Faller, 2009).

2.2 Early Functional Design

To better understand and explore the context of *functional model* and *early functional design*, we identify and examine several fundamental key questions relating to the functional design environment: *how early is early? What information is available at this stage? What are the current existing analysis techniques?* These questions are briefly discussed below before being explored further in their proceeding sections.

By "early" in the design process, we mean early enough to make design changes or incorporate new requirements without incurring excessive cost, time or effort. The artefacts produced at this stage are often functional model, capturing system

requirements. Such models do not make references to specific hardware architecture and are abstract and minimal in detail. Refinement of such models is typically necessary to derive more detailed architectural models and system implementations.

2.2.1 Functional Analysis

Functional analysis is defined as "*the process of identifying, describing, and relating the functions a system must perform in order to fulfil its goal and objectives*" (NASA, 1995). The result of this process is a functional model. A function is performed by one or more system elements composed of hardware, software, firmware, people, and procedures to achieve system operations. In the early stage, functional analysis plays an important role in assisting system engineers to understand the objectives and constraints in the process of developing and formulating system design solutions. All functional aspects of the system are identified, organized and defined. It can be also be used to derive requirements, which are then allocated to solutions in the form of a physical architecture. (NASA, 1995) highlights several of its key roles especially in identifying system requirements, identifying measures of system effectiveness and performance, excluding design alternatives that do not meet requirements, and providing insights to system-level model builders.

Functional analysis deals more with *what* the system has to deliver than *how* to do it. It examines system functions and sub-functions that will accomplish system's goals. As the level of details is refined and functions are decomposed into sub-functions, the requirements associated with the functions are decomposed as well. This decomposition increase manageability as it organizes functionalities and connections into a more easily understood hierarchy. The process is repeated until each process is decomposed into basic sub-functions, and until connections between functions, sub-functions, and environment are fully defined.

This functional analysis flow process is described and summarized in (FAA, 2006) as shown in Figure 2. The process starts off with list of requirements and constraints as input, from which top level functions are defined. These functions can then be organized into logical relationships, decomposed, and evaluated accordingly to produce the functional architecture and more refined requirements and constraints.

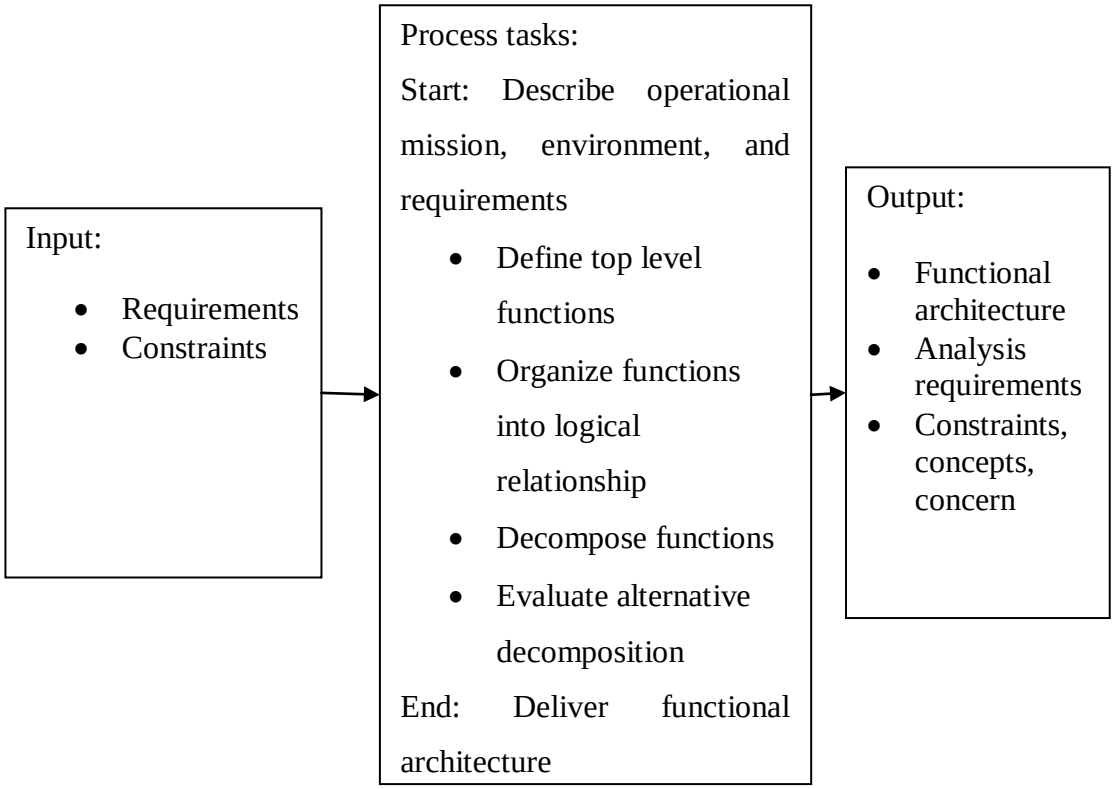


Figure 2: Functional analysis process (summarized from FAA, 2006)

Complex safety critical system requires hazard analysis to be performed as *early* and as *often* as possible to avoid costly design iteration. Therefore it is beneficial to start safety analysis early at functional level before design solutions progress too far. Once the functional model is established, the design is evaluated to detect design limitations and weak points to help establish a more robust and improved design. Functional Hazard Assessment (FHA) and Preliminary System Safety Assessment (PSSA) are classical preliminary assessment techniques widely accepted and practiced in this role (ARP 4754, 1994). FHA looks at failure conditions associated with the system functions. PSSA is applied after FHA to demonstrate how the system meets the qualitative and quantitative requirements for various hazards identified. It also derives safety requirements for subsystems, mainly using FTA. PSSA is iterative and performed continuously throughout the system design phase.

2.2.2 Functional Hazard Assessment (FHA)

FHA identifies and classifies failures associated with system functions, covering both functional losses and malfunctions. It can be organized according to

- System levels, for example, in the avionics system presented in (ARP 4761), the two levels of FHA are the Aircraft level FHA and System level FHA.
- Or system operation phases and modes, for example, ground idle, landing, take off.

Similar to the decomposition of functional models, FHA is conducted starting from higher-level functions to lower-level functions. Failure conditions related to these functions are considered and the effects of the failures are identified and classified.

The primary aim of FHA is to identify hazardous functional failure conditions. Its methods are relatively direct and results are usually represented in a tabulated format as show in Table 1. First, the function and its purpose and behaviours are defined, and phases of the systems where functions can be performed are also recorded. Hypothetical failure conditions (for example: loss of functions) that can occur for this function and its effects are identified. This identification of the effects of function failures on the system allows a representative severity class to be assigned. Lastly, a comments column records necessary modification ideas and describes potential methods of addressing the failures.

Table 1: Example of FHA on car brake function (source: Johannessen *et al.*, 2001)

Function	Failure Condition	Phase	Effects on System	Severity	Comments
Electric brake force distribution	Loss of function	Straight dry road	More brake force on rear wheels	Marginal	Only affects a loaded car, which gain longer braking distance
...	...	...	...	...	...

2.2.3 Functional Failure Analysis (FFA)

FHA is extended in (Johannessen *et al.*, 2001) and (Papadopoulos, 2001) to include failure classes, similar to the classes used in HAZOP (Kletz, 1997). Failures are classified, although not restricted, into 'Omission', 'Commission', 'Timing', and 'Value'. Further discussion and analysis of the meaning of these failure classes can be found in (Bondavalli, 1990). Hazards identified in FFA can be used to represent top events of a fault tree through HiP-HOPS. The extended FFA in (Papadopoulos, 1998) organizes the tabulated FHA to include: function, failure type(s), effects of failure on system, severity of failure, detection method, recovery plan, and design recommendation. An example of FFA is presented in the Table 2.

Table 2: Example of FFA on car brake function

Function	**Failure Type**	**Effects on System**	**Severity**	**Detection**	**Recovery Plan**	**Design Recommendation**
Brake Pressure	Omission	No brake force available; vehicle cannot be stopped; driver loses control.	Catastrophic	Using feedback from pressure sensor	Not possible	Redundant components and back up mechanism should be introduced
...	...	...	...	...	...	...

2.2.4 Preliminary Systems Safety Analysis (PSSA)

PSSA builds upon FHA to generate a complete list of updated system requirements, and is used to demonstrate how a system will fulfil requirements for hazards identified in FHA. In PSSA, design and architectural decisions are made and these help to generate lower-level system requirements. Safety analysis techniques like FTA are often employed to perform top-down analysis to determine how failures can lead to functional hazards identified in FHA. This process also identifies remedial strategies, for example

by introducing fail-safe architectures, to meet the safety requirements. PSSA is iterative and applied continuously throughout design process to derive thorough system requirements.

FTA will be discussed along with other safety analysis techniques in the next section.

2.3 Classical Safety Analysis

Classical safety analysis techniques such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) are employed to predict the safety of safety critical systems. However, as modern systems are becoming increasingly complex, employing distributed architectures and programmable electronic components, new approaches are being developed to meet the rising intricacy of designs. Model-Based Safety Analysis (MBSA) is one such recent development.

Before further discussing the two prominent paradigms of MBSA – Compositional Safety Analysis (CSA) and Behavioural Safety Analysis (BSA) – we first study the background of several commonly used classical safety analysis techniques which essentially underpin the newer MBSA approaches.

2.3.1 Fault Tree Analysis

Fault Tree Analysis (FTA) is an approach that aims to identify the root causes of an undesired event by performing top-down traversal of a fault tree. A fault tree itself is a diagrammatic description that shows how combinations of component failures (*basic event*) can cause the undesired event (*top event*) to occur. These component failures are connected within the fault tree through logical operators (for example, AND/OR).

Two types of analysis can be performed in FTA: quantitative and qualitative analysis. Quantitative analysis is performed to calculate the probability of the top event. Qualitative analysis is performed to identify the *necessary* and *sufficient* combination(s) of basic events that cause the top event. These necessary and sufficient combinations are called *minimal cut sets* (Vesely *et al.*, 1981). The identification of minimal cut sets in a fault tree helps the designer to focus on the design weak points. For example, if the failure of component C1 is identified during FTA as being a direct cause of the failure

of the system, the system designer is now informed about this critical component, and can reassess the design (e.g. by introducing a backup component to prevent this single point failure).

2.3.2 Failure Mode and Effects Analysis

Failure Mode and Effects Analysis (FMEA) provides an analysis that details possible failure modes for each component and their effects on the system. FMEA is presented in tabular manner and can contain additional information about the component failure (e.g. criticality and probability of occurrence). Classical FMEA is unable to determine complex failure modes resulting from multiple component failures. This limitation is addressed and overcome in HiP-HOPS where FTA and FMEA are automatically generated and analyzed from system model, in hierarchical approach, enabling it to determine further effects of a component failure.

Most classical techniques operate in either an inductive or a deductive way. Inductive techniques attempt to determine the effects of a failure, while deductive techniques attempt to discover the causes of a failure. FTA is a deductive approach, whereas FMEA is an inductive approach. FTA and FMEA are traditionally a laborious and manual process.

2.4 Compositional Safety Analysis

In CSA, predictive models of system failure are typically produced in the form of well-known safety artefacts like fault trees. This technique models the failure behaviour of the system - as opposed to the nominal (working) behaviour - by extending components with local failure information.

The process starts from requirements which are translated into preliminary models. These models can be decomposed into structural hierarchies, and the local failure logic of components in these hierarchies is provided by analysts. Faults trees or FMEAs are then automatically produced by establishing how the local effects of component failures combine as they propagate through the topology of the system. The process is flexible

and adaptable to different stages of model development (from early functional models to more detailed architectural models).

This is especially valuable as assessment can be started early in the design process when concrete system details are still scarce. CSA produces safety artefacts which are familiar to safety engineers including fault trees and FMEAs. These artefacts reveal potential failures and design weaknesses (e.g. single points of failure) which can guide possible design modifications, and help to derive and refine requirements. CSA techniques allow quantitative analysis and in some cases also architectural optimization.

One key limitation of CSA is the inability to perform formal verification. Another limitation is the fact that FTA and FMEA are static analyses, which do not take into consideration the changes in system states and are therefore unable to capture dynamic behaviour. This limitation has been to some extent addressed in HiP-HOPS with a recent extension that enables assessment of sequences of failures via synthesis of temporal fault trees and FMEAs (Walker *et al.*, 2006).

Examples of techniques based on CSA are: Component Fault Tree, State Event Fault Tree, Embedded Systems Safety and Reliability Analyser, and Hierarchically Performed Hazard Origin and Propagation Studies.

2.4.1 Component Fault Tree

The Component Fault Tree concept (CFT) (Kaiser, 2003) is an extension to traditional fault trees that allows definition of partial fault trees corresponding to actual technical components. Although traditional fault tree allows modularization, it paths the failure propagations to the root causes. Component failures are often affected by other components, and therefore it is hard to model component independently.

Apart from the similarities with traditional fault tree - including the analysis techniques - CFT also introduces the concept of a ‘port’ to enable the modelling of component as independent entity. Each component has internal basic events, logical gates, and input and output ports which connect to other components. Components without input ports can be analysed alone. Instead of fault trees, Directed Acyclic Graphs, called “Cause Effect Graphs” (CEG) are used. CEGs differ from traditional fault trees in the sense that

repeated events are represented only once in CEG and CEG may contain several top events (more than one failure mode). Figure 3 shows an example of a CFT, with "BE" representing basic event:

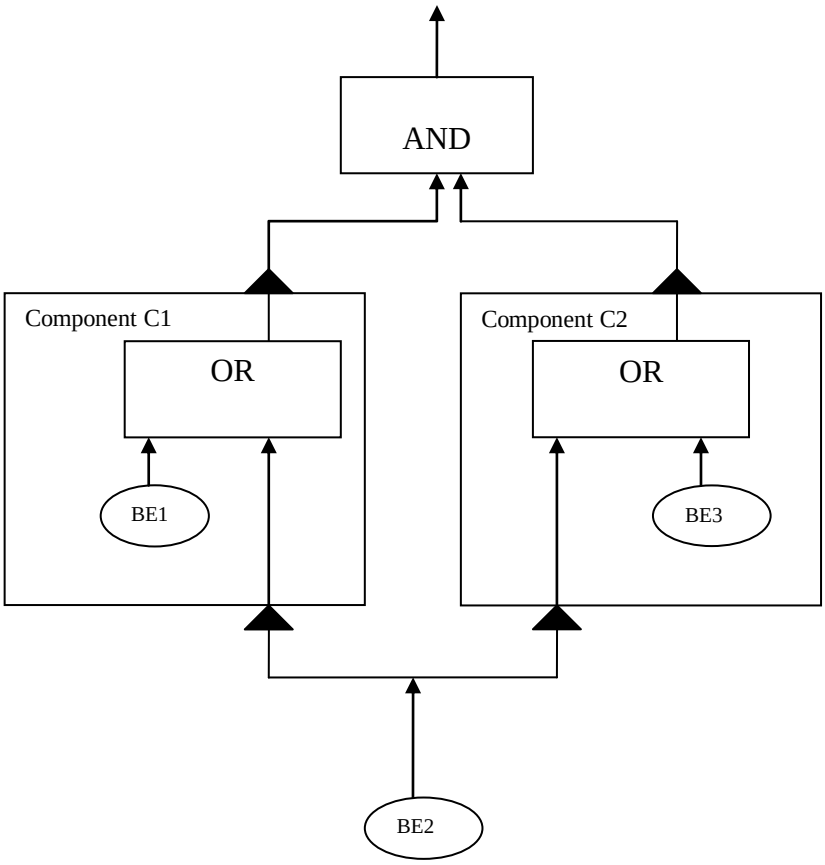


Figure 3: CEG in a Component Fault Tree

2.4.2 State Event Fault Trees

State Event Fault Trees (SEFTs) (Grunske *et al.*, 2005) are the youngest compositional technique. They aim to extend traditional FT capability by distinguishing the notions of "states" and "events" notion to better capture sequence of action and state history. Traditional FTs do not differentiate states (a system condition that last over a period of time) from events (sudden phenomena, especially state transitions). So semantically, a SEFT is an extended state machine model instead of a true combinatorial model (i.e. like a traditional fault tree). Like many other state machine based models, the states and events that appear in SEFT are not necessarily failures.

SEFTs deal with a finite state space for each component, where each component is exactly one state at each instant of time. The notion 'state' in SEFTs indicates the condition a component is in for a given interval of time, while 'event' indicates the instantaneous phenomena that do not take time to occur (e.g. state transitions). System failure can be represented as either top-events (which happen instantaneously) or top-states (which last over a period of time). In SEFTs, the commonly used gates fall into the following categories:

- NOT gates, which have one state input and one event output. There is no negation of an event;
- OR gates, which combine either states or events (state OR state / event OR event). There is no OR gate that mixes states with events;
- and lastly, AND gates, which join states and/or events (state AND state / state OR event). There is no simple (event AND event) except for History AND and Sequential AND. This is because an event is assumed to occur over {a very | an infinitesimally} short time interval, thus only one can occur at a time. Gates need to be converted to match state inputs to event outputs and vice versa.

2.4.3 Embedded Systems Safety and Reliability Analyser

ESSaReL (Embedded Systems Safety and Reliability Analyser) (Kaiser *et al.*, 2007) is a recent development that aims to integrate different models (Markov Chains, Fault Trees, State charts) and support the new State/Event Fault Tree (SEFT) approach (Kaiser *et al.*, 2004). ESSaRel takes SEFT models as input and produces probabilistic analysis results based on Deterministic Stochastic Petri Nets (DSPNs) as output. Main phases for safety analysis employing SEFTs are:

1. SEFT construction
2. Translation of SEFT into DSPN
3. Analysis of flattened DSPN.

SEFTs are constructed like traditional fault trees, but just like CFT, they are organized by components. A SEFT enables analysts to trace back and finds out which system

states or events initiate, propagate, or inhibit the failure behaviour. Figure 4 shows an example of SEFT fragments.

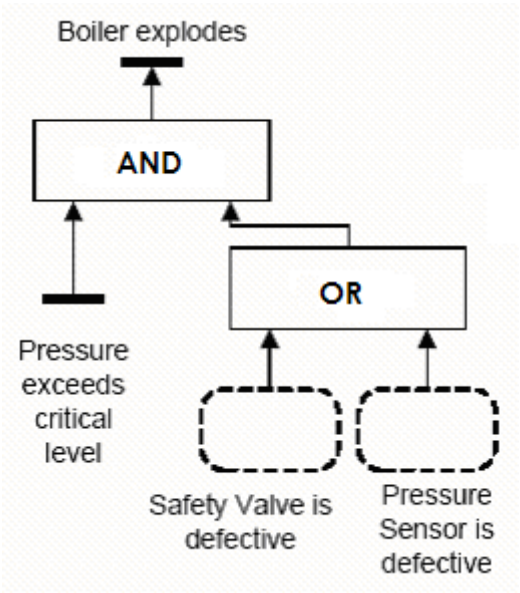


Figure 4 : SEFT fragment (source: Kaiser, 2007)

Being a state-based model, a SEFT cannot be evaluated by traditional combinatorial FTA algorithms, and therefore needs to be translated into formal notation where known algorithms exist (Kaiser *et al.*, 2007). Deterministic and Stochastic Petri Nets (DSPNs) (Ciardo & Lindermann, 1993) are chosen as they are better suited to analyzing dynamic models of this sort (German, 1995). Each SEFT state is mapped to a DSPN place and each SEFT event to a DSPN transition. SEFT gates, however, are translated as a whole by looking up the corresponding DSPN structure in a dictionary (Kaiser *et al.*, 2007).

For quantitative probabilistic analysis of SEFTs, the component SEFTs are translated into DSPN and then merged (flattened) into one flat net. Then an existing Petri Net analysis tool, like TimeNET (German & Mitzlaff, 1995) is used and it offers both transient and steady-state analysis. Currently the translation to DSPN and its analysis is carried out manually.

2.4.4 Hierarchically Performed Hazard Origin and Propagation Studies

Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) (Papadopoulos *et al.*, 2001) - on which this project/thesis is developed - is a compositional safety analysis technique currently being developed in the University of Hull, which pioneered semi-automated fault tree analysis and IF-FMEA (Interface focused FMEA). HiP-HOPS models the propagation of failures through the system by constructing hierarchical component failure logic into a network of fault trees.

The HiP-HOPS tool can work in conjunction with commonly-used system modelling tools, such as Matlab Simulink or Simulation X. Failure editors can be integrated in these modelling tools which allow the system designers to annotate the model components with failure information.

The failure information describes how the component fails and its relationship with other component failures in the system. HiP-HOPS then takes this information and examines how the component failures propagate through the system topology, producing sets of interrelated fault trees and eventually an FMEA. This approach also enables the hierarchical structure of the system to be captured neatly in the fault trees.

HiP-HOPS consists of three main phases: a model annotation phase, a fault tree synthesis phase, and the generation of minimal cut sets and FMEA (the analysis phase). Figure 5 illustrates the concept and steps involved in HiP-HOPS. The process starts with the system designer (or analyst) annotating the components with failure information. This stage provides information to HiP-HOPS on how the components can fail. Local failure information takes the form of a set of expressions which are manually added to each component. A failure class which occurs on a port (input or output connections of the component) is known as *deviation*. These local failure expressions describe how deviations of the component output can be caused by a combination of deviations received at the component inputs and/or by failure modes (internal malfunctions) of the component itself. For example, in this figure, we assume failure in component C1 can be caused by its internal malfunction C1BE. Failure O-S1 which can occur in S1 is said to be the system failure.

HiP-HOPS uses the local component failure behaviour and the topology of the model to generate a network of fault trees that connect output deviations of the system to internal failures of individual components. These fault trees show how the component failures propagate from one component to another and affect the system or subsystems individually or in combination with other component failures. Here, to maintain simplicity, component failure C1BE is assumed to be a direct cause of system failure O-S1.

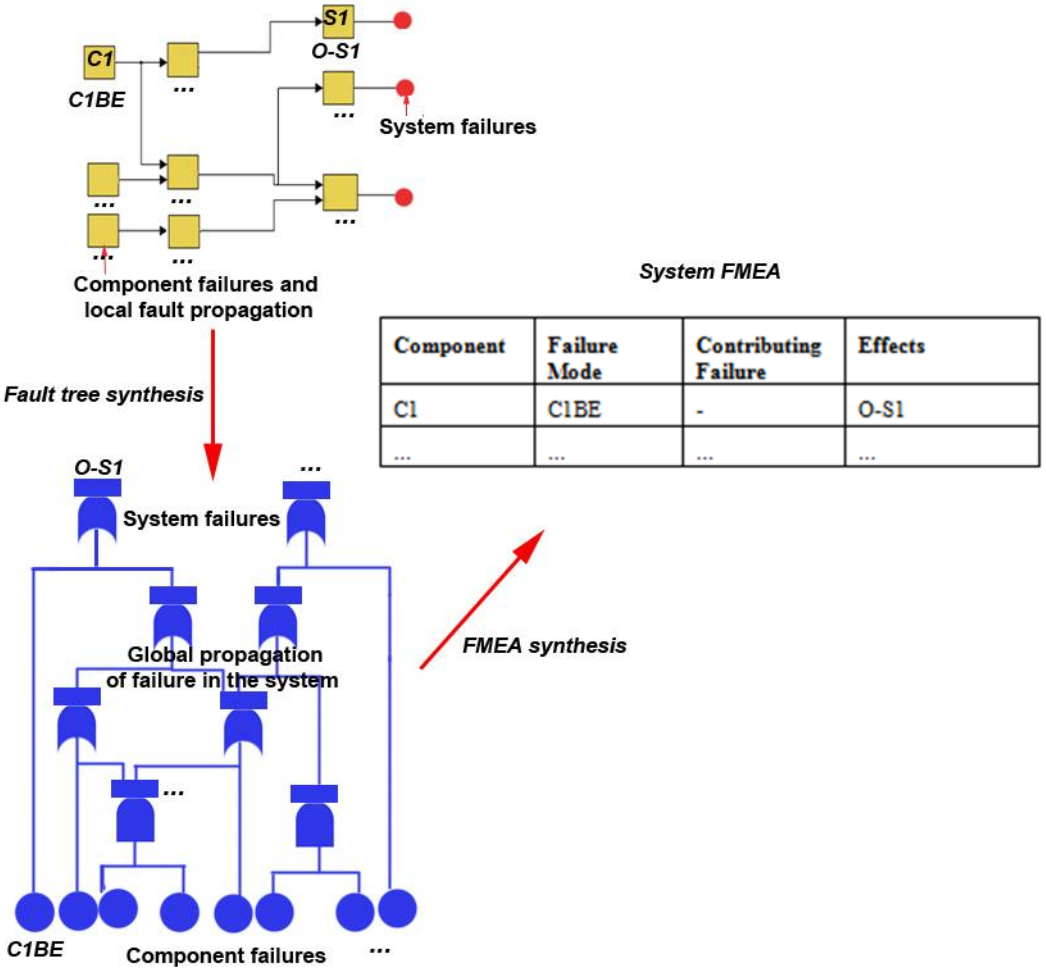


Figure 5: Main phases in HiP-HOPS

In the analysis phase, the synthesized fault trees are analyzed and an FMEA is generated. Both qualitative and quantitative analysis can be performed depending on the amount of information provided. Qualitative analysis is performed through the

implementation of selected FTA algorithms and minimal cut sets are obtained from this analysis. Eventually, the data is combined into a multiple failure mode FMEA which shows both direct effects of failure modes on the system as well as the further effects of the failure modes (i.e. the effects a failure mode can have on the system when it occurs in conjunction with other failure modes). The resultant FMEA is presented in a table that can be conveniently displayed through a web browser. In this example, the FMEA table shows how component failure C1BE is a direct cause of system failure O-S1.

HiP-HOPS not only provides a consistent and robust model throughout design and analysis, it also takes the pressure off the designer through the application of effective analysis early in the lifecycle – by detecting potential design flaws early on, they can be quickly remedied before they become serious problems. HiP-HOPS is flexible and scalable, and is therefore well-suited to be performed iteratively throughout the design phase.

2.4.5 Summary of CSA Techniques

Having reviewed the aforementioned CSA-based techniques, we have selected HiP-HOPS to facilitate CSA in the IACoB process based on the following reasons. Firstly, HiP-HOPS has been considerably developed in the recent years. It has been tested on several industrial systems (Papadopoulos *et al.*, 2005), (Hamann *et al.*, 2008), and has recently been extended with the capabilities to enable the analysis of temporal fault trees (Walker, 2008) and multi-objective optimisation (Parker, 2010). HiP-HOPS also provide tools implementation which allows practical support. In the context of this thesis, it is also a natural choice because of the support and expertise available on site, as well as the access provided to source code for any necessary expansion of the tool (please see work on chapter 6).

2.5 Behavioural Safety Analysis

In the Behavioural Safety Analysis (BSA) approach, system-level effects of failures are established by injecting faults into the formal specification of the system, and the effects of these faults on system behaviours are observed. The BSA technique employs model checking to allow formal verification. Model checking formally verifies safety

properties which represent safety requirements and enables the assessment of dynamic behaviour. The model checking process is performed when a detailed formal model is established. Formal models are expressed as *state automata* (or "*finite state machines*") in the language of the particular technique (e.g. Altarica language for Altarica and NuSMV for FSAP/NuSMV). Model checking performs exhaustive exploration to check whether a safety property – which is usually expressed in temporal logic – holds. The tool produces Boolean output with a counterexample when safety properties do not hold to show traces of 'simulation' on how the breaching condition is reached.

The strength of this approach lies in its ability to facilitate automated formal verification and capture the system's dynamic behaviours. It is also possible to differentiate between transient and permanent failures and model the temporal ordering of failures. However, this technique also has a number of drawbacks including the fact that most model checker tools require the system model to be expressed in that particular model checker input language. Valuable safety artefacts like fault trees produced from a model checker generally have 'flat' structures representing a disjunction of all minimal cut sets, which can hamper understanding of the fault trees. The analysis is also typically qualitative in nature and not probabilistic. Other challenges of model checking techniques can be found in (Holzmann, 2005). Formal models (which are required as input to the model checker) are only developed at later stages where designs are more mature, detailed and stable. Lastly and perhaps most critically, model checking based approaches are computationally expensive and inductive in nature which means that the exhaustive assessment of the effects of combinations of component failures is infeasible in any non-trivial system.

2.5.1 Introduction to Model checking

Model checking (Clarke & Emerson, 1980) tools explore all possible system states to check if a condition holds true. This way, it can be shown that a system model truly satisfies certain safety requirements (properties). If a model state is encountered that violates the property, a counterexample is generated to show how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. By studying it, sources of the errors can be identified.

For a model checking tool to do this "state-exploration", the input model needs to be represented in the right format. Most real-time embedded or safety-critical systems are control-oriented, and for control oriented systems, finite *state machines* (FSM) or finite state automata are widely accepted as the abstract notation for defining the system model. To model real-life complex industrial systems, the system model needs to be represented in different level of detail (sub-systems, components) that can be combined and integrated. Most model checking tools have their own rigorous formal language for defining input models.

Typical safety properties that can be checked using model checking are of a qualitative nature. For example: "Both processes can never be in their failed state simultaneously", "memory overflow can never occur", or "as long as the plane is not on ground, the engine should never stop".

These properties (safety requirements) need to be expressed in a precise and unambiguous statement, and *temporal logic* is employed to do this. Temporal logic is a form of logic specifically tailored for statements and reasoning which involve the notion of order in time. In model checking, it serves to formally state properties concerned with the execution of systems. PLTL (Propositional Linear Temporal Logic) and CTL (Computation Tree Logic) are the two most commonly used temporal logic in model checking.

In temporal logic, classical Boolean combinators are necessary: the constants *true* and *false*, the negation $\neg$, Conjunction *and* $\wedge$, Disjunction *or* $\vee$, logical implication $\Rightarrow$, and double implication $\Leftrightarrow$ (*if and only if*). These combinators enable the construction of complex statements by relating various simpler sub-formulas.

In addition to Boolean operators, temporal logic also includes the additional temporal connectives. The table below shows some of the common temporal connectives in CTL. "E" (for some paths) and "A" (for all paths) are path quantifiers, while "F" (for some states) and "G" (for all states) are state quantifiers for states in a path. "X" indicates *next*.

Temporal Connectives	Description
EX φ	True if formula φ is true in at least one of the next states
EF φ	True if there exist some states in some path that satisfies formula φ
EG φ	True if every state in some path satisfies formula φ
AX φ	True if formula φ is true in every one of the next states
AF φ	True if there exist some state in every path that satisfies formula φ
AG φ	True if every state in every path satisfies formula φ

Figure 6: Commonly used temporal connectives table

An example of a safety requirement specified in CTL is the statement: "`AG (ComponentA = activated)`" which specifies that component A must be activated all the time.

A system state is defined by a tuple of values for each of the variables. For example:

```
state1 = (componentA=off, componentB=off, level=low).
```

Most model checker tools convert a state model of the system provided as input into a particular state transition model called a *kripke structure* (Kripke, 1963). This conversion process removes hierarchies in the finite state machine, as well as parallel compositions, guards and actions on transitions. Each state in a kripke structure contains one value for each state variable, and transition in a kripke structure indicates changes in one or more state variable values. A given property is checked against the kripke structure, which is further unfolded into an infinite tree where each path in the tree indicates a possible execution or behaviour of the system.

Figure 7 below shows an example of an execution tree. In an execution tree, the states of the system are arranged so that the root is the initial system state and the children of any state denote the next possible states. The definitions of how the system changes from one state to another, and what states it can be in next, are defined in the input model. If, for example, in Figure 7 the first variable of each node represents the state of ComponentA, the second represents the state of ComponentB, and the third represents that value of requirement `AG (ComponentB = off) ⇒(level = low))`, which means: "every time component B is in its off mode, the level state is low", then through

the inspection of this execution tree, a model checker can determine that the requirement is clearly not true.

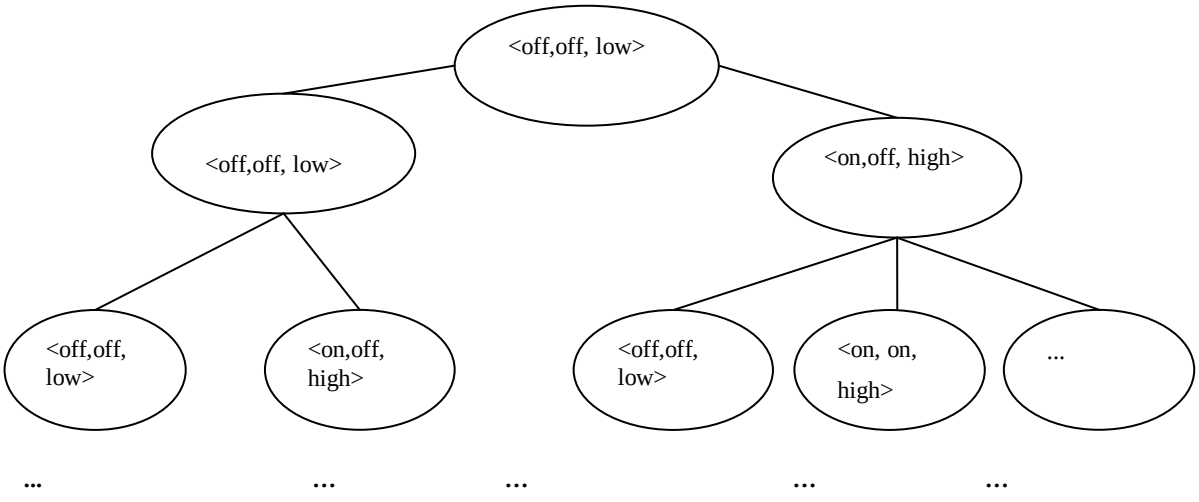


Figure 7 : Execution tree showing next possible states

2.5.2 FSAP/NuSMV-SA

FSAP/NuSMV-SA (Bozzano *et al*., 2003) is a safety analysis technique developed within the ESACS project and consists of two main components: 1) FSAP (Formal Safety Analysis Platform) which provides a graphical user interface 2) NuSMV-SA which performs the safety assessment and is based on the NuSMV model checker.

FSAP/NuSMV-SA takes system models in NuSMV format as input and produces analysis results as well as trace information like simulation results, counterexamples, property verification results, minimal cut sets and fault trees as output. The following phases describe how safety analysis is performed in FSAP/NuSMV-SA:

1. Model capturing
2. Failure mode capturing and model extension
3. Safety requirement capturing

4. Model analysis

5. Results extraction and analysis

Model Capturing: The starting point is the formal model of the system, which can be modelled as a system model (written by a design engineer) or a formal safety model (written by a safety engineer). These models are written using the NuSMV (Cimatti *et al.*, 2000) input language and entered using a text editor. An example of simple bit adder written in NuSMV is shown in Figure 8:

```
MODULE bit(input)
VAR
output: {0,1};
ASSIGN
output:=input;

MODULE adder(bit1, bit2)
VAR
output: {0,1};
ASSIGN
output:=(bit1 + bit2) mod 2;
```

```
MODULE main
VAR
random1: {0,1};
random2: {0,1};
bit1: bit(random1);
bit2: bit(random2);
adder:
adder(bit1.output,bit2.output
;
```

Figure 8: Fragment sample of NuSMV model for one-bit adder (source: Bozzano *et al.*, 2003)

Failure Mode Capturing and Model Extension: Failure modes which describe how various components of the system can fail are defined using a GUI. Here the safety engineer can specify which nodes of the system can fail, how they fail, and with what parameters. The failure modes can be stored and retrieved from a Generic Failure Modes Library.

Once the failure modes are defined, they are then inserted into the models and the result is called the extended model. “Injection” of this failure mode also produces a new piece of NuSMV code that is automatically inserted into the extended system model. Figure 9 shows a sample of NuSMV model extended with failure modes:

```
VAR
Output_nominal: {0,1};
Output_FailureMode: {no_failure, inverted};
ASSIGN
Output_nominal :=input;
DEFINE Output_inverted := !Output_nominal;
Output Output := case
      Output_FailureMode = no_failure : output_nominal;
      Output_FailureMode = inverted : output_inverted;
esac;

ASSIGN
next(output_FailureMode) := case
output_FailureMode  = no_failure: {no_failure, inverted};
output_FailureMode = inverted : inverted;
esac;
```

Figure 9 : NuSMV model extended with failure mode (source: Bozzano *et al.*, 2003)

Safety Requirement Capturing: During this stage, the design and safety engineer define functional and safety requirement that will be used to assess safety behaviour of the system. The safety requirements are expressed in temporal logic and the input process is simplified through an available requirements library from which safety patterns can be chosen and instantiated. Requirements can be subsequently verified using the NuSMV model checking verification engine.

Model Analysis: FSAP/NuSMV performs simulation of both system model and extended system model. The behaviour of a system is assessed against the functional and safety requirements. The model analysis phase is performed by running the model checker on the system properties. Two main verification tasks are performed:

1. Model checker NuSMV tests the validity of a system property and generates a counterexample if the system property is not verified. At the moment, the model checking tool is BDD-based.

2. FSAP/NuSMV generates fault trees. The FSAP/NuSMV-SA tool is able to perform failure ordering analysis (Bozzano & Orita, 2003) which provides information on timing constraints (where applicable) among the events in a minimal cut set.

Result Extraction and Analysis: The analysis results are displayed in formats compatible with traditional commercial tools. Trace results obtained from a simulation task or counterexample are bound to a system verification property or minimal cut set, and can be displayed in textual, structural (XML), graphical (gnuplot utility) or tabular display. Fault trees generated can be viewed in commercial tools like FaultTree+.

Joshi (Joshi *et al.,* 2006) discuss several limitations of FSAP/NuSMV-SA which include the "flat" structure of generated fault trees (fault trees generated are only two levels deep and can be very broad). This might hamper the understanding of the system via the fault trees. A normal fault tree shows multiple levels of causation, and in the CSA approach also indicates the propagation of failures through the system. There is also limited flexibility in defining the fault model, as there is no good way (in capturing the hierarchy) of specifying fault propagation or simultaneous or dependent faults.

2.5.3 ALTARICA

The AltaRica language (Arnold *et al*., 2000) is designed to formally specify the behaviour of a system. AltaRica models can be assessed through fault tree generators or model checkers. The process takes in system models (AltaRica models) as input and generates fault trees and model checker verification results as output.

The main phases for safety assessment with AltaRica are as follow (Bieber *et al.*, 2002):

1. System Modelling
2. Formal Safety Requirements
3. Graphical Interactive Simulation
4. Safety Assessment : Fault Tree generation and Model checking

System Modelling

The AltaRica language is a hierarchical specification language based on constraint automata used to formally model system specifications and behaviour. Formal syntax and semantics of the language are described in (Point & Rauzy, 1999). AltaRica

describes complex systems consisting of interacting components with semantics expressed in terms of an interfaced transition system. Components can be defined hierarchically and composed together (synchronized) to create more complex components. AltaRica provides a general synchronization mechanism and other features like bidirectional flow, broadcast vectors and transition priorities.

An AltaRica model of a system consists of hierarchies of components called nodes. A node gathers flows, states, events, transitions and assertions.

Flows: visible variables of the component which are used to exchange information with the environment (other components of the system).

States: local/internal variables which are inaccessible by the environment.

Events: occurrences that change the state of a component (e.g. failures). Transitions: describe how internal states may evolve. They are characterized by a guard, an event name and a command part. A guard is a Boolean constraint over the component flow or states. An event is the trigger for transition. Lastly, the command part is a set of values assigned to some state variables, which describe the actions or results of the transition.

Assertions: Boolean formulae that describe the constraints linking flows and internal states. These constraints express mutual dependencies on/between the states of the components.

Consider the following example in Figure 10 of a simple component called “block”. A block represents a basic energy provider and receives two Boolean inputs, I and A. Input I is true every time the component receives energy and input A is true whenever the component is activated. The component has a Boolean output O that is true whenever it produces energy. It has an internal state S that is true whenever the component is working properly (the safe state). Initially, S is assumed to be true. A transition for the block can occur if the component is safe (S is true) and the event ‘failure’ occurs. After this transition, the component is no longer safe (S is false). The block produces output O only when both of the inputs are true and the component is safe.

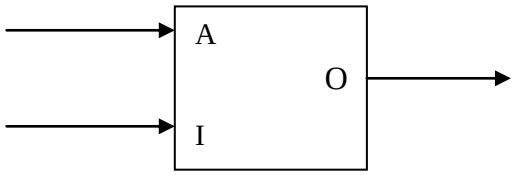


Figure 10: Sample of block

The following shows the representation of block in the form of an AltaRica node :

```
Node block
      Flow
            O : bool : out ;
            I, A : bool : in ;
      State
            S : bool ;
      Event
            Failure;
      Trans
            S |- Failure → S: = false;
      Assert
            O = (I and A and S) ;
      External initial state = S = true ;
Edon
```

The whole system node (main node) is built by connecting basic nodes. Components are combined together by two means: assertions and synchronizations. Global assertions allow the definition of the flow connections (for example: stating that input flows of a node are the output flows of another node). Connections can also be related to events shared by a set of nodes (synchronization of events). Recently a time extension has been introduced to AltaRica to enable the verifying of real-time AltaRica specifications (Pagetti *et al*., 2003)

Formal Safety Requirements:

In this phase, the safety requirements are formalized with the use of linear temporal logic operators (bieber *et al.*, 2004). A library can be defined to store (and retrieve) useful safety formulae.

Graphical Interactive Simulation:

Safety engineers can check the effect of failure occurrences on the system architecture using the graphical interactive simulator. It enables the safety engineer to choose an event and the resulting state is computed by the simulator.

Safety assessment

Once a system model is specified in the AltaRica language, it can be compiled into a lower level formalism for verification purposes. Compilers available for AltaRica could produce automata, fault trees and stochastic Petri Nets. Figure 11 Illustrates the main phases of Altarica and its safety artefacts.

- Automata: An AltaRica program can be compiled into a finite state automaton on which formal verification techniques like model checking can be performed by the MEC 5 (Arnold, 1994) model checker.

- Fault trees: another compiler could produce a fault tree on which reliability analysis can also be carried out through the ARALIA program/tool (Groupe ARALIA, 1996). Compilation of AltaRica descriptions into Boolean formulae (i.e. a fault tree) is discussed in (Rauzy, 2002) where a mode automaton is introduced as the underlying mathematical model. An extended type called AltaRica Data-Flow which is based on mode automata is introduced.

- Stochastic Petri Nets: the third compiler produces a stochastic Petri Net on which performance analysis can be performed.

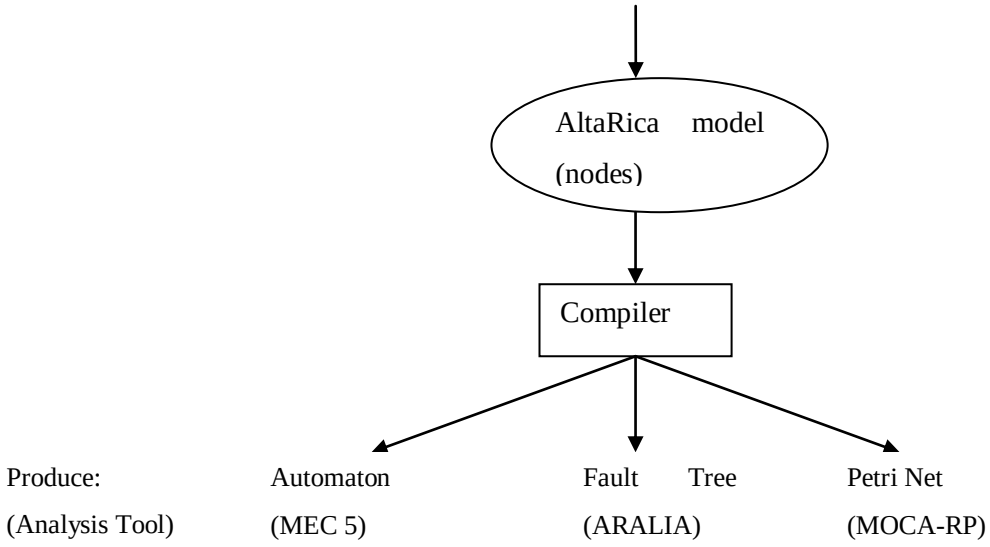


Figure 11 : Analysis phases of Altarica

2.5.4 Summary of BSA Techniques

NuSMV has been selected as a BSA method to complement HiP-HOPS in IACoB based on the following considerations. Firstly, NuSMV is fundamentally a symbolic model checker. Symbolic model checkers are generally more scalable and therefore, are recommended for larger real-life systems. NuSMV has a strong advantage over Altarica, as (Bieber, 2002) highlighted that Altarica's MEC model checker is limited on the size of systems it can handle. NuSMV is also more suitable for Boolean-based data (as opposed to enumerated type) (Miller, 2007). Considering that most of the failure data obtained from HiP-HOPS are Boolean-based, NuSMV is a logical choice. The NuSMV support tool is also available as an open-source program which allows it to be tailored more effectively into a future integrated support tool.

2.6 Relevant Work on Other Integrated Approaches

To propose an integrated approach, it is first of all, important to understand the notion of 'integration' in this context. We believe that the generic primary characteristics which constitute an integrated approach (methodology, tool, or both) include:

- The application of these different techniques (CSA and BSA) within the same process
- The link between relevant models (model representations in CSA and BSA) should be well-established
- The integrated approach should provide better analysis capabilities (and results) compared to a single technique

Here, we also briefly reviewed recent model-based techniques which incorporate the critical elements of CSA and BSA, namely the capabilities to perform FTA and model checking within its application.

Techniques like Altarica and FSAP/NuSMV are able to perform model checking and generate fault trees. However, the primary characteristic shared between these techniques is that they start with a BSA-based technique, and the fault trees are produced as a result of the model checking analysis. As previously mentioned, this approach has the drawback of having a "flat" fault tree structures.

ForMoSA (Formal Methods and Safety Analysis (Ortmeier *et al.*, 2004b)) proposed a combined use of traditional safety analysis (FTA) and formal verification via the use of 'failure-sensitive specification'. Failure-sensitive specifications are used to derive more complete failure modes by first generating all possible scenario combinations. It then removes implausible behaviours and behaviours that do not fulfil specification rules which govern the intended behaviour. The extracted behaviours results in a list of failure modes, which is then separated from intended behaviour (nominal model). These failure modes along with results obtained from independently-constructed fault trees, are used to extend the nominal model. The main challenge this approach faces is the state explosion problem in its generation of 'failure-sensitive specification', in which all combinations are first to be produced.

The "failure injection" nature of model checking in BSA can also be used to validate results of CSA. Failure injection approach introduces failures and observes the changes in the system behaviour in response to these failures. Lisagor (2006) recommends the use of results from failure injection to verify the completeness of minimal cut sets produced from FTA in CSA.

Here, we approach the integration from a radically different angle. Instead of starting from a BSA process (which tends to be performed at the later stage of design development), we believe it is possible to start the integrated process from CSA. We also demonstrate a way in which the results of the CSA can be used for the systematic construction and refinement of state automata that describe dynamic behaviour and can be further subjected to BSA. In the following chapters, we discuss further the proposed approach and demonstrate how these characteristics of integration can be achieved.

2.7 Chapter Summary

In this chapter, the background and context of this research work in system modelling was briefly discussed. Attention was paid to the early functional design as we hope to fill the gap in providing a more-robust safety assessment at (although not limited to) this stage. MBSA techniques have been recently developed to cope with the rising complexity of modern systems. The two most widely used MBSA-based techniques are the CSA and BSA. CSA is generally based upon classical techniques like the FTA and FMEA. It is widely used in reliability engineering, and its Boolean-based and compositional nature makes the analysis efficient and scalable. However, CSA is mainly limited to static analysis and is not capable of formal verification. This chapter reviewed several CSA- based techniques, for example CFT, SEFT, and HiP-HOPS.

BSA, on the other hand, is based upon formal techniques like model checker. It relies on exhaustive state exploration and allows formal verification of the model. BSA limitations include the fact that it requires a relatively mature model, and therefore it is often applied only at the later design stage. Example of BSA-based techniques reviewed here are Altarica and FSAP/NuSMV.

This chapter provided an overview of the working mechanism, strengths, and limitations of these techniques. We also studied the different objectives, and complementary of aspects of CSA and BSA. In the next chapter, we proposed a method to combine their applications.

CHAPTER 3. A method for Integrated Application of Compositional and Behavioural Safety Analysis (IACoB)

3.1 Introduction

This chapter develops a method for safety analysis which integrates the application of CSA and BSA techniques. Its application is mainly explored in two contexts: early functional design, and more detailed architectural design.

In early functional design, the method is applied to an early model where design details are not yet mature. At this stage, focus is drawn to the benefits yielded by the method in enabling systematic derivation of abstract behavioural models via CSA and then useful application of model checking on such models. Application of the method is also demonstrated in a later stage of design where the model includes more detail about the architecture of the system. It is shown that the method is generic and applicable as an iterative process that can span across the design lifecycle.

The key steps involved in IACoB analysis are illustrated in Figure 12. The method starts with a given set of system functional requirements and safety specifications. From this, a functional model of the system is established, which shows input processing and output functions and dependencies among them, e.g. the data exchanged among functions (or material and energy in the general case). In the next step, design engineers are asked to examine further this model in order to evaluate the severity of failures of output functions, i.e. functions provided by the system to users and its environment. Each function is then annotated with its local failure behaviour in the style of HiP-HOPS, enabling automated preliminary FTA to be conducted via application of CSA analysis. The result is the generation of an FMEA of the system model. This FMEA is then studied and interpreted, leading to recommendations for design improvements, and additional safety measures in particular.

With the introduction of new safety measures, the requirements and system model are updated. The severity of the failures of output functions and the local failure behaviour of all functions are revised, and the next iteration of FMEA can be performed. This might again lead to further iterations, until the design is deemed satisfactory. At this point, results from FMEA are analyzed and interpreted to assist the further development of the design via construction of state machines that represent system dynamic behaviour. Model checking is then used to verify whether this dynamic system model conforms to the requirements and specifications. If conformity is verified, the process proceeds to either further refinement of the model and iteration of the above process or its implementation. Otherwise, counter-examples are produced to show how the model fails to fulfil certain requirements and to point out to useful revision of the model. Each of these stages is discussed further in the next section.

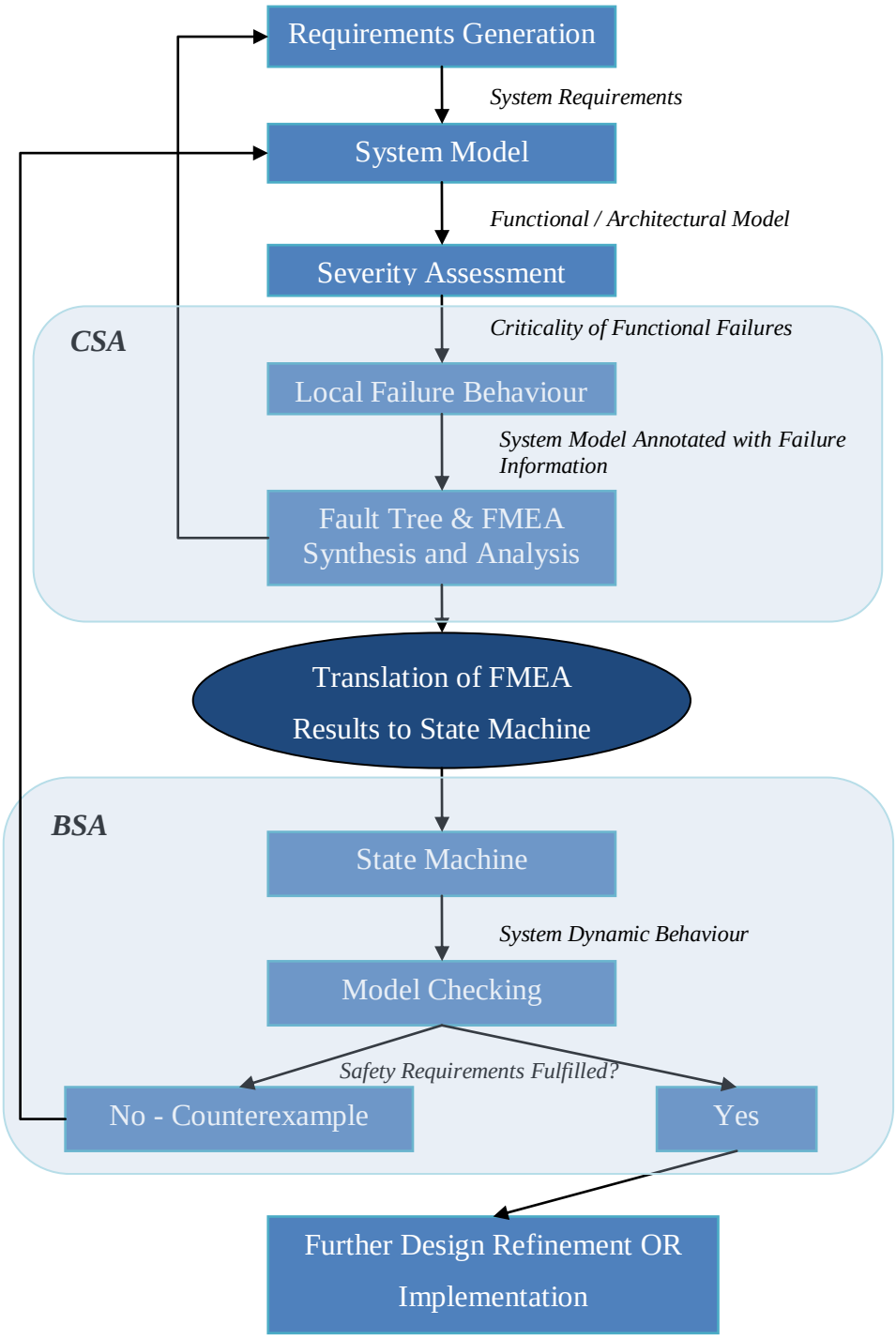


Figure 12: Process outline of IACoB method

Figure 13 shows how it is believed the IACoB method could fit into the traditional safety assessment process (adapted from (ARP 4754)). The inclusion of IACoB in this process enables techniques like FTA, FMEA and formal verification to be performed earlier (following FHA once functions are allocated) rather than being applicable only during or after a more detailed PSSA.

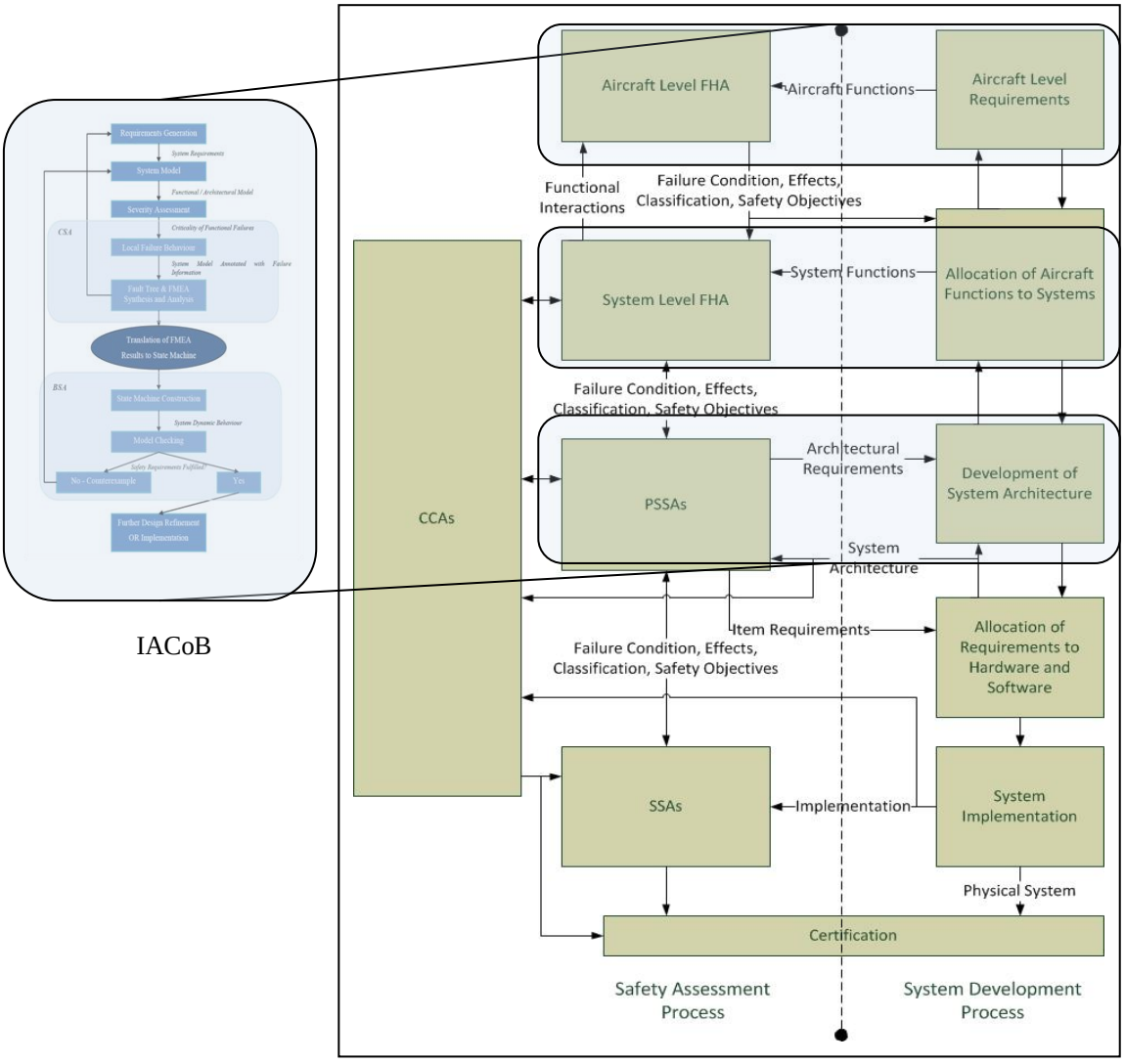


Figure 13: System development and safety assessment process (source: ARP 4754)

The IACoB Process

This section presents a detailed description of IACoB. It gives a series of steps in which various analyses and their results help to transform a basic initial functional model of the system into a more robust, better prepared model which eventually becomes the blueprint for system architecture and thus the foundation of development. For each step, an accompanying table is given that summarises the input, primary activity, and output of the processes that take place in the given step.

3.2 Functional Model

The essential element of early conceptual and preliminary design is the development of a functional model. A functional model is the *representation of the system functional architecture that fulfils the system requirements.* From a list of requirements, functions are initially derived from the identification of 'processes' that need to be performed by the system. The task of identifying and organizing the system functions depends on the application and the experience of designers. However, in general, functions would fall into three categories: input, processing, and output functions. A Functional model can then be seen as a function-oriented pipeline where data or control gets transformed as it flows from input to output.

Functional models are popularly represented as functional flow block diagrams, an example of which is shown in Figure 14.

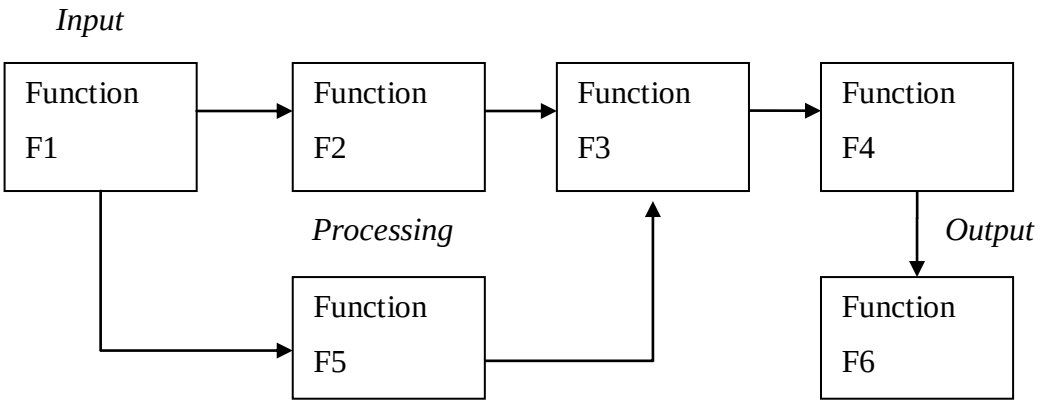


Figure 14: Functional model in basic block diagrams

An *input* function refers to a function that acquires the input parameters needed by the system. In architectural designs, input functions are typically implemented with sensors or communication controllers reading data on communication buses. A *processing* function is a function that describes how the input data is processed, and finally an *output* function provides function to the user or environment based on information received from processing functions.

In situations where more information is available in the early stage, it is possible not only to model the higher level functions of the system, but also to show hierarchical decomposition of functions in networks of sub-functions.

One popular diagrammatic technique used in functional modelling is the Functional Flow Block Diagram (NASA, 2007). A Functional Flow Block Diagram (FFBD), also known as a Functional Flow Diagram or a Functional Block Diagram, is a step-by-step flow diagram consisting blocks connected through lines, and it is used to represent functional flow in a system.

FFBDs are a general tool and can define operational and support sequence for systems, but also describe the processes for developing and producing systems. In FFBDs, functions are organized according to their logical order of execution, and might depend on the execution and completion of other functions. To manage complexity, functions are decomposed into several levels. This functional decomposition defines the lower-level functions and their sequential relationship allows traceability throughout.

Basic elements of a FFBD include: function blocks, directed lines and connection logic symbols. Each function block in FFBD represents a single defined function. The block contains information like the function name (which is generally expressed as verb) and the function identification number (which establishes relationships and traceability between levels). Reference functions which are denoted as bracketed blocks can also be used to show reference to other functional diagrams. Figure 15 shows an example of a functional block.

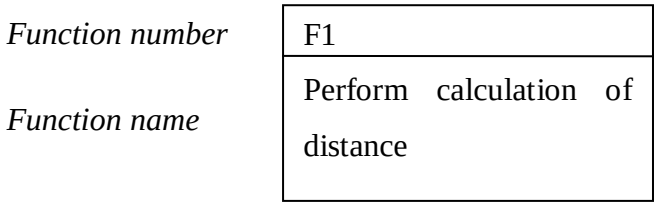


Figure 15: Example of a functional block

Function blocks are connected by directed lines, which depict function flow and flow direction. Usually the function blocks are structured so the flow is directed from left to right, as shown in Figure 16

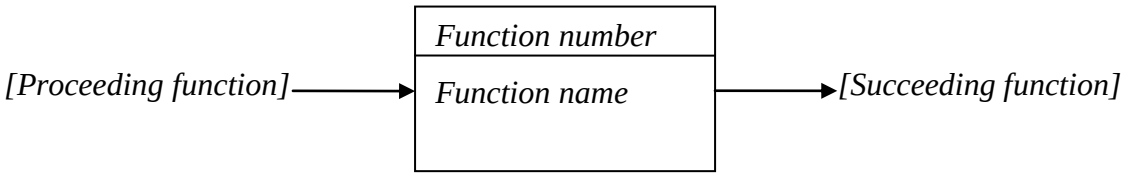


Figure 16: Connection flow between functions

In summary, the input, process and output of this stage of the construction of functional model are:

Input	Initial (textual) requirements
Process	• Identify, define and relate functions • Translate requirements into functional model
Output	Functional model/ functional architecture

3.3 Severity Assessment of Output Function

Once the high level functional model is developed, it is important to assess the severity of deviations at output functions. We define an 'output function' as the following:

A function that interacts directly with the environment of the system by providing information material and energy.

The 'environment' of the system refers to the users and other external elements outside the system boundaries. The interaction with the external environment is not exclusive to output functions. An input function naturally accepts its value and 'input' to be processed from its environment. Here however the focus is placed on output functions because of their effects on the environment and their potential contributions to hazards.

The classification of failures is assisted through the use of guide words similar to those used in HAZOP, (Bonadavali & Simoncini, 1990) and (Pumfrey, 1999). These guide words help categorize failure classes and their use depends on the level of details available. *Omission* and *Commission* are commonly used at this stage. 'Omission' of an output function indicates the condition in which function output is not provided when expected, while 'commission' indicates the provision of unwanted output. It is also possible to use more to indicate timing failures (*late, early)* and value failures (*more, less*).

The categorization of failures in terms of severity is based on the IEC-61508 (IEC 61508, 1998) and is presented in Table 3. According to this, the severity of failures can be classified, according to their consequences for humans (or for the quality of services provided in the more general sense), into the following categories: *Catastrophic*, *Critical, Marginal* and *Negligible.*

These severity classes are assigned to the failures using simulation, testing or experience. The classification can be assigned as part of information presented in FFA (please see the example FFA in Table 2) and allows the safety analysis to be focused correctly, especially when there is any conflicting priority in the functional design. Failures at functional outputs under the 'catastrophic' or 'critical' categories need to receive higher priority compared to those which have 'marginal' or 'negligible' effects.

Table 3: Allocation of severity category based on consequences to people and service (IEC-61508)

Description	**Consequence to human stakeholders**	**Consequence to service**
Catastrophic	Fatalities and/or multiple	-

	severe injuries	
Critical	Single fatality or severe injury	Loss of major system
Marginal	Minor injury	Severe system damage
Negligible	Possible minor injury	System damage

In summary, the input, process and output of this severity assessment for output functions process stage are:

Input	System functional model
Process	Estimate risk and classify the severity of output function failure based on their consequences
Output	• Severity analysis of output functions • Prioritisation of output failures using severity as criterion; Identification of higher priority (critical) functions

3.4 Local Failure Behaviour

Apart from deciding the severity of failures of output functions, it is also important to determine the potential causes of these failures as these can be seen to arise from the specified functional model of the system. Qualitative analysis which identifies these causes could provide valuable feedback towards improvement of the functional architecture design by pinpointing weak parts in the system model, for example single points of failure that can lead to severe output failures. To achieve this, local failure behaviour of each function needs to be established. Failure behaviour can be described using *deviations.*

A deviation contains information on the failure type and the 'port' (i.e. input and/or output) where it occurs. Failure of output function can be defined by output deviations. An output deviation describes a set of Boolean expressions that represents the causes of the output failure. These causes can consist of internal failures, input deviations, or both. When representing deviations, the dash "-"symbol is used to separate the failure type from the input or output parameters. Failure causes are connected by logical operators. Commonly used logical operators are the disjunctive operator ("OR", "∨", "+") and the conjunctive operator ("AND", "∧", ".").

For example, the following expression:

```
Omission-Output = InternalFailure OR Omission-Input
```

defines an output deviation for Function F1 (which is shown in Figure 17) where an internal failure (`InternalFailure`) of the function or an omission of the input (`Omission-Input`) can cause an omission of the output (`Omission-Output`) in the function.

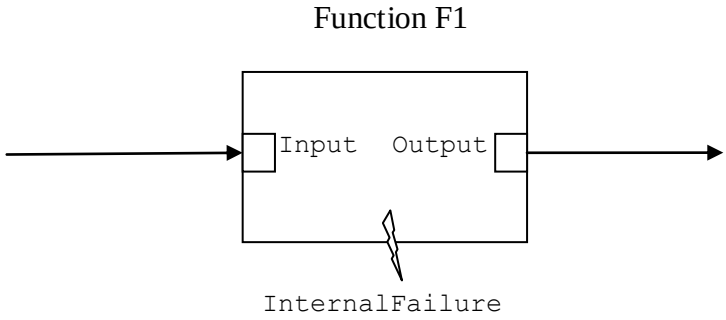


Figure 17: Local failure behaviour for Function F1

This annotation of failure behaviour for each function, in addition to the topology of the functional architecture, allows failure logic to be developed and propagation of failure to be subsequently established. The synthesis and analysis of fault trees are employed to achieve this.

In summary, the input, process and output of this local failure behaviour annotation of function are:

Input	Functional model
Process	Establish failure information for each functional block
Output	• Functional model with failure data information • Establishment of causes (internal failure and failure of input) of function failure

3.5 Fault Tree and FMEA Synthesis and Analysis

Compositional safety analysis techniques like HiP-HOPS can be applied to perform the automated construction and analysis of fault trees from the functional model.

The global view of failure propagation in the functional architecture can be captured by traversing and following the causal links defined in each function's local failure information. The process starts from a failure in an output function and moves backwards progressively to record failures from other functions which contribute to this particular output failure. This results in a set of fault trees that represent the relationships between failures of output failures and their root causes in the functional model of the system.

These fault trees in HiP-HOPS can be analyzed qualitatively, and the results are summarized in an automatically generated FMEA table. The FMEA table shows the direct links between potential failures of all functions in the model and the output function failures which represent the hazardous failures of the system. Traditional FMEA shows only the *direct effects* of a single failure on the output functions, but because of the way the FMEA is constructed by HiP-HOPS from a series of fault trees, it also captures the effects of a functional failure when it occurs in conjunction with failures from other functions. These are termed the *further effects* of the function failure.

The FMEA table generally contains information on the list of functions, failure modes, effects of the function failures in terms of the failures of the output functions, and other contributing failures that need to occur collectively to cause failures in output functions. It is also possible to include information on severity of the affected output function, recommended treatments and other general comments.

The FMEA table essentially shows how internal failures of functions can contribute to the hazardous failures of output functions. By determining these relationships between failures in functions and failures in output functions, it is then possible to establish the criticality of the function in the functional architecture.

Figure 18 illustrates this point by showing a functional architecture that produces three output functions: Function F7, Function F8 and Function F9. For simplicity we assume that every function has a single output failure - omission - and that this is caused by

internal failure of the function or omission of its input. For this reason, we do not explicitly define the obvious annotation of each function. Severity assessment performed during FFA identified that the severity of omission failure in output Function F8 is catastrophic, while the severities of Function F7 and Function F9 are marginal.

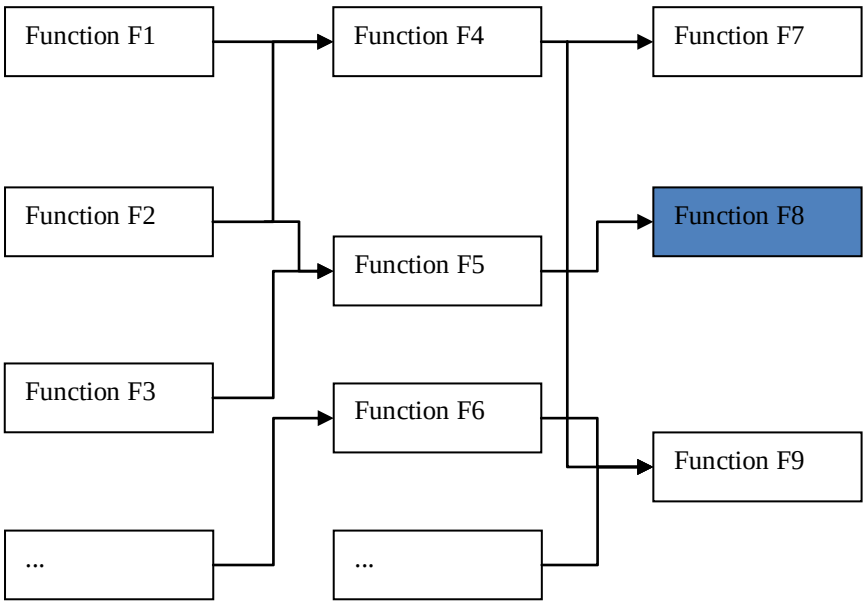


Figure 18: Example of the functional architecture

HiP-HOPS analysis of the above model with its simple failure annotations creates the fault tree of Figure 19 for the failure Omission of Function F8. The fault tree is analyzed and an FMEA table (as partly shown in Table 4) is generated. The FMEA table identifies those functions (Function F2, Function F3 and Function F5) whose failures play a vital role in contributing to Omission of Function F8 failure. These are shown in shaded function blocks in Figure 20.

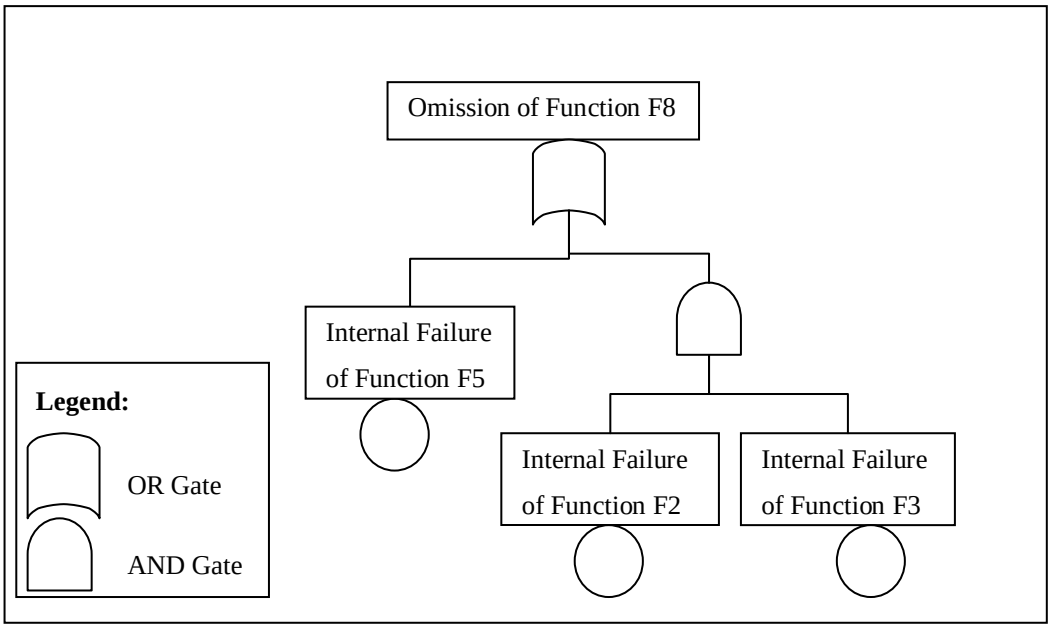


Figure 19: Example of generated fault trees

Table 4: Example of FMEA table

Function	**Failure Mode**	**Effects**	**Contributing Failure**	**Severity**
Function F2	Internal Failure	Omission of Function F8	Internal Failure in Function F3	Catastrophic
Function F3	Internal Failure	Omission of Function F8	Internal Failure in Function F2	Catastrophic
Function F5	Internal Failure	Omission of Function F8		Catastrophic
...	...	...	...	...

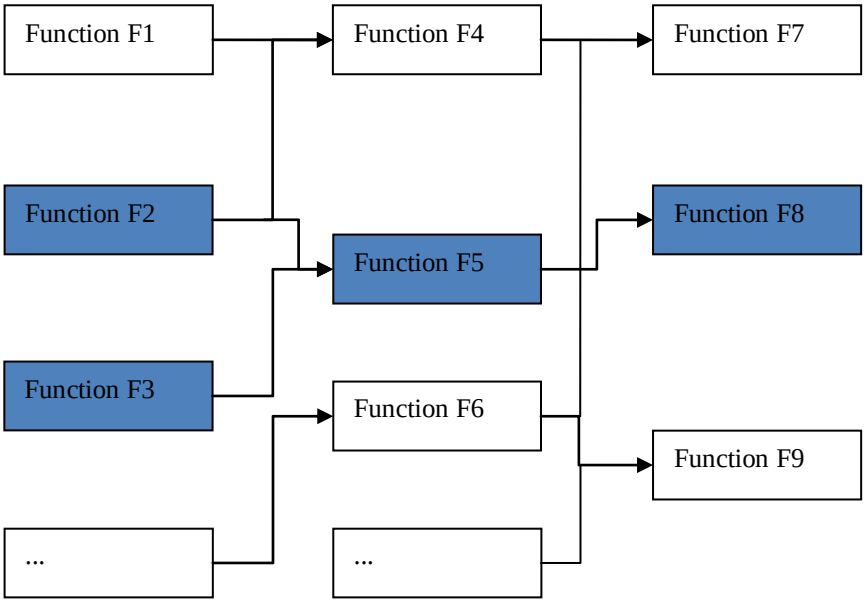


Figure 20: Identified critical functions based on failure propagation

Interpretation of the table allows the failure behaviour of the functional architecture to be checked against safety requirements. By examining the FMEA table, safety measures can be devised (for example, by revising the design structure or introducing safety mechanism). Focus is placed especially on functions whose failures contribute to hazardous effects, as they need to be prevented by design or at least have their impacts minimized. Further discussion on the common techniques and solutions employed to divert critical failures are discussed in Chapter 6.

While the design solution ultimately relies on the engineer's decision and experience, this identification of criticality for each function offers assistance in the management of effort allocation and design modification. For example, apart from identifying that focus should be placed on Function F2, F3 and F5 due to their failure criticality, the fault tree also shows that Function F5 is a single-point of failure that might need additional attention.

To achieve a safer design, modification of the system structure, for example through incorporation of backup or redundant components (in later versions of the designs) for fail-safe purposes, is often necessary. The introduction of these safety mechanisms might result in new modules (functions in an earlier design, or implementation

components in a detailed design) and brings about the need to iterate the process to generate updated FTAs and FMEAs.

From this process, more refined design and safety requirements can be derived. The identification of lower level failures leading to output failures can be evaluated, and this helps derive more refined design requirements. This results in fewer late design changes in comparison to traditional practice where assessment at this stage is often limited to FHA. In classical safety assessment, FTA and FMEAs are performed manually, making safety analysis a laborious process while often meaning it is deferred until the PSSA stage where the details of the design are more concrete.

In summary, the input, process and output of fault tree and FMEA synthesis and analysis are:

Input	Functional model with local failure behaviour information
Process	• Generate FTA and FMEA from functional model (HiP-HOPS is applicable for this) • Identify weak points in system design - contributing function failures that leads to (severe) output function failure : by linking failures in output functions to their causes
Output	• Effect of functional failures on output functions • Better understanding of the criticality of input, processing and output functions in the system

3.6 Generation of State machines and Their Translation into Model Checker Input Language

One important aspect in this research is the investigation on how application of CSA and BSA techniques can be integrated constructively. To achieve this, we need to establish an effective association between the primary elements of CSA and BSA techniques, namely the FTA/ FMEA results (output of CSA) and state machines (input of BSA) respectively.

In IACoB, the results of the FMEA are used to construct behavioural models that can be subjected to BSA. Indeed application of the method leads to synthesis of state machines that describe the dynamic behaviour of the system in conditions of failure. Iterative

application of the method starts from “abstract” state machines which progressively become more “refined” as they contain more details about the behaviour of the system. In general, these state machines show how functional failures assessed in the FMEA move the system to degraded or failed “modes” where there is reduced function or no function at all. We use the term *mode* as in (Papadopoulos, 2000) to indicate an abstract functional state in which the system delivers a set of functions. We also use the term “*mode chart*” to indicate a state machine which shows transition between modes.

To create such mode charts, in IACoB, an FMEA-ModeChart assistance table is constructed to help organise state machine elements and create the “abstract” state machine. Transitions in this state machine are then refined to produce a more “refined” state machine. Traceability between abstract and refined state machines allows the understanding of how transition in a more-refined level affects the higher level state of the system. The refined state machine can also be produced directly from HiP-HOPS failure annotations to model system failure-related dynamic behaviour. Both abstract and refined state machines can be represented in the NuSMV model, and can be extended with nominal behaviour. Figure 21 illustrates the process of generating state machines from FMEA results.

This process is further discussed in the following section. First, we investigate the representation of abstract state machines, their purpose and application, how they can be constructed based on information gathered from results of previous process, and the value of their analysis.

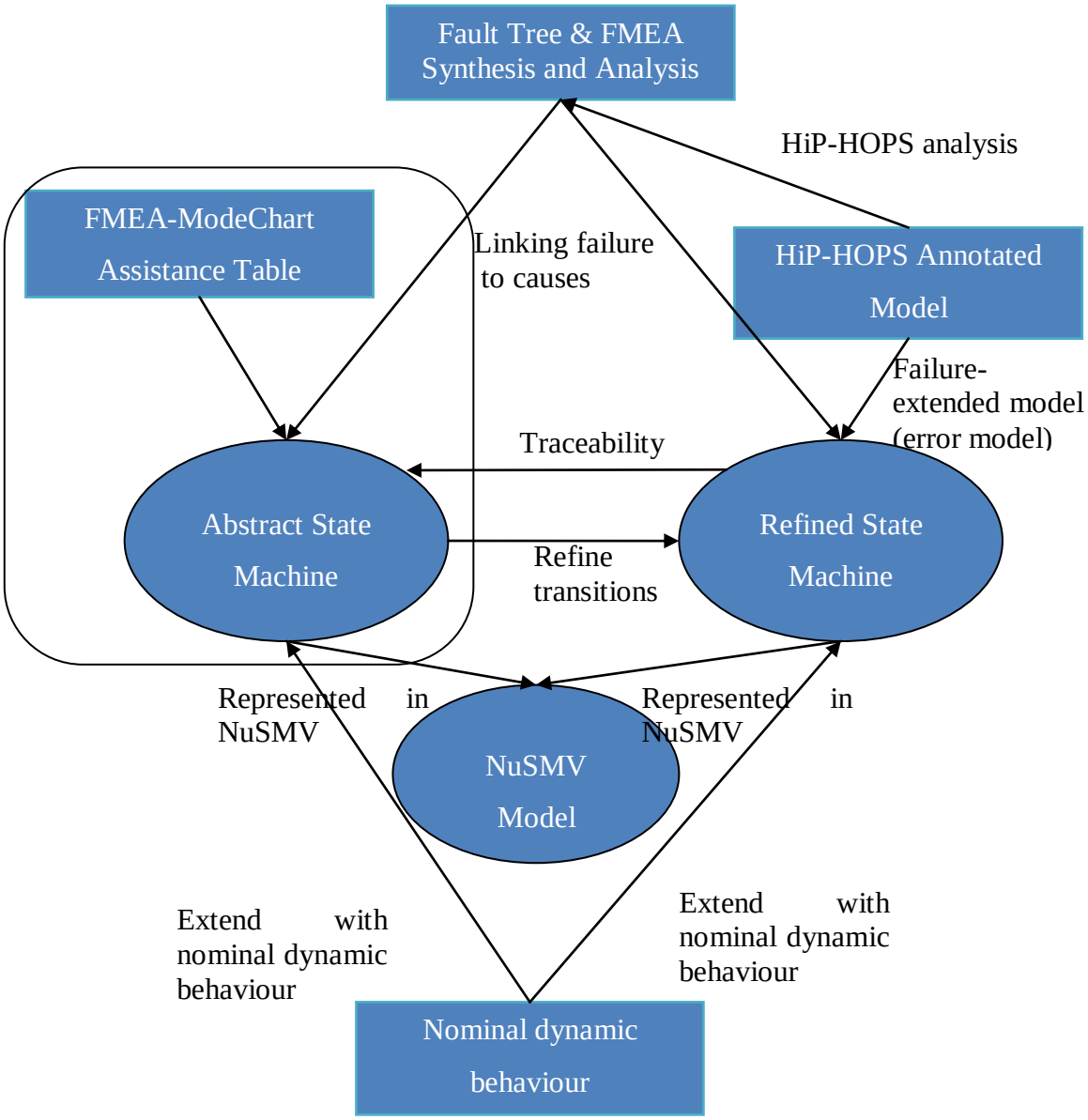


Figure 21: Generation of state machines

3.6.1 Modelling the Dynamic Nominal Behaviour of a System

State machines can be used to model the dynamic behaviour of discrete event systems. In the early functional design stage, the state machines are usually abstract and are used to model high-level system behaviour. In later stages of development, they can be used to model more refined behaviour.

Overview of State Machine fundamentals

Traditional (finite) state machines are flat and sequential. Such state machines have proved to be a useful theoretical tool in computer science, but are unsuitable for representation of large or complex system. David Harel introduced state charts as a language to describe state machines by extending finite state machines with additional capabilities, including hierarchy, concurrency and priority (Harel, 1987). While the approach we discuss here is not tied to any particular commercial support tool, the modelling of state machines in this thesis is based on the general semantics of state charts. Being an unofficial language, many variants of state charts have been proposed in the literature - as reviewed in (Von der Beek, 1994). One of the most widely known implementations of state charts is the STATEMATE tool, the semantics of which are described in (Harel & Naamad, 1996). The complete discussion of semantics and syntax of state charts is out of the scope of this thesis, and readers are referred to (Harel, 1987). This section presents the key concepts of state charts and discusses how these foundations enable state charts to be a prominent notion in modelling complex system behaviour, and how its extension can be adopted as part of early design analysis.

State

A state is defined by Weilkiens (2007) as *the representation of a set of value combinations for the underlying system elements*. It describes the system internal behaviours at a given time (and when a state is active, the system is said to be 'in' that state). Apart from the internal behaviour which is executed based on defined events, a state can have three other behaviours that are triggered by predefined events:

1) entry behaviour – which is executed immediately once the state is entered
2) exit behaviour – which is executed immediately before the state is exited
3) do behaviour – which is executed while the state is active

Figure 22 below illustrates an example of a state chart. States are shown in rounded-corner rectangles, and charts are shown with soft greyed dash border. Sometimes states and charts are not distinctively/uniquely named, for example S1 is both a chart and a state. A state may itself host and contain other state charts and this creates the relations of 'parent-chart'/'sub-chart' and 'parent-state'/ 'sub-state'.

States in state charts are categorized into two types: OR states and AND states. OR states (for example S1, S4 and S5) are states that have sub-states related to each other by 'exclusive-or', i.e. they are mutually exclusive and are reached sequentially. Basic states (for example S2, S6, S7, S8, and S9) are states that are at the bottom of hierarchy and do not contain any sub-states. Basic states are considered OR states. AND states (for example S3) are states that contain at least two sub-charts that are reached simultaneously when the parent-state is activated, and thus AND states are used to model concurrency.

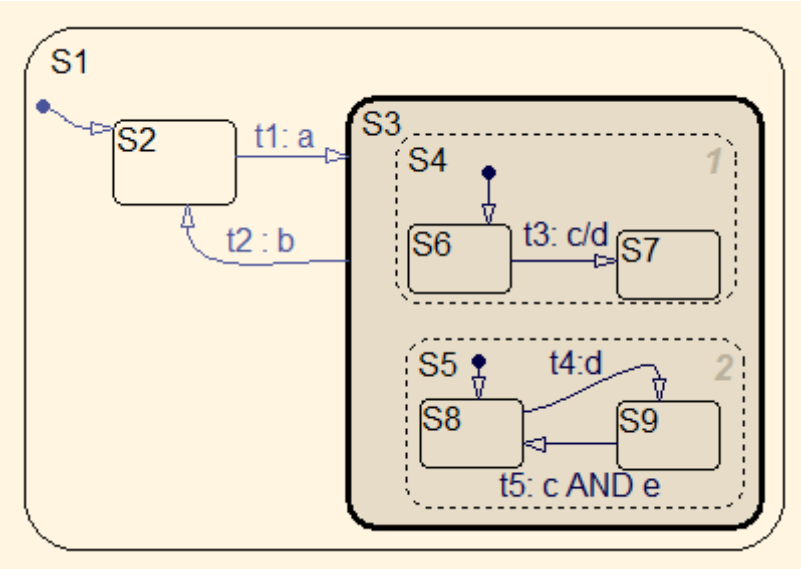


Figure 22: Sample of state chart

Transition

Transition (t1..t5) defines the trigger and condition of the directed relationship between states. These are expressed by a 'transition label' which can be defined in the form of

‘**e[c]/ac**’, where *e* is the *event* that triggers the transition, *c* is the *condition* that guards the transition, and *ac* is the *action* that is performed when and if the transition occurs. A *default* transition defines the state that is entered once the chart is active. In the Figure 22, the defaults are the transitions to S2, S6 and S8, and are denoted by the circle-ended arrow.

Events can be generated externally or inside the same sub-chart. Additionally, events can also be predefined (as mentioned earlier) and be generated when a state is entered “*en(s)”* or exited “*ex(s)”*or when the value of a Boolean variable “*variable”* becomes true “*true(variable)”* , false “*false(variable)”* or changes “*change(variable)”*.

Conditions are used to guard the transitions. A condition can contain expressions on data values or expressions on elements of state charts. The combination of events and conditions is called the *trigger* of the transition, and the trigger is fired only when the Boolean combination of these events and conditions are true. A condition persists until the instance when the inverse condition holds.

Transitions can generate actions which control other charts. These actions are categorized into basic actions, which form basic events, and compound actions, which modify state chart elements (i.e. data variables). Referring to the Figure 22 for example, transition t3 which is triggered by event ‘c’, will cause action ‘d’ to be fired, which in turn triggers transition t4 and causes a transition from state S8 to S9. Transition actions will be executed after the source state is deactivated, but before the target state is activated. Similar to events, actions can also be executed when a state is entered or exited in addition to appearing along transitions.

The following is an excerpt of the transition label syntax grammar customized from (Loer, 2003):

```
<label>      ::= [<event>]/{<action>;}
<event>      ::= E                                  event variable
                 |      (<event>and<event>)        Boolean combination
                 |      (<event>or <event>)
                 |      not(<event>)               negation
                 |      en(<state>)                chart entered state
                 |      ex(<state>)                chart left state
                 |      [<condition>]              condition is true
                        tr(<condition>)            condition became true
                 |      fs(<condition>)            condition became false
                 |      ch(<condition>)            condition changed
<condition>  ::= C                                  condition variable (Boolean)
                 |      not <condition>            negation
                 |      (<condition>) and (<condition>)        Boolean combination
                 |      (<condition>) or (<condition>)
                 |      in(<state>)                chart is in 'state'
<action>     ::= E                                  event variable
                 |      tr!(C)                     set  C to true
                 |      fs!(C)                     set C to false
                 |      C:=<condition>             assign the value of (Boolean) condition to C
                |D:=   <condition> assign the value of (data) <arithmetic expression> to D
```

In the state chart semantics system behaviour is described as a set of possible *runs* (Harel & Naamad, 1996). Runs represent the system responses to external stimuli, and consist of a sequence of *status*. A status is the set of all currently visited model states and may contain information on: active states, values of data items, conditions, generated events and scheduled actions. The transition from one status to the next is defined by *steps*. In addition to external stimuli, changes occurring during and since previous steps would trigger transitions between states and as a result the system moves to a new status, as illustrated in the figure below:

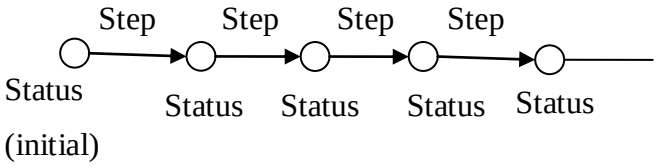


Figure 23: Status and steps in state charts semantics (source : Harel & Naamad, 1996)

General Principles and Language Restrictions

Although currently there is no agreed common standard that defines formal semantics for state charts, (Harel & Naamad 1996) describe the general principles to define the semantics of state charts:

1) Changes that occur in a step, and reactions to internal and external events, can only be sensed after completion of the step
2) Events 'live' for duration of one step only and are not remembered in subsequent steps
3) Calculations in one step are based on the situation at the beginning of the step (i.e. the states the system is in and the value of data items)
4) Greediness property: the maximal subset of non-conflicting transitions are always executed
5) Execution of a step takes 0 instances of time, i.e. it is instantaneous

3.6.1.1 *Overview of NuSMV*

As introduced earlier in Chapter 2, NuSMV is a newer version of Symbolic Model Verifier (SMV, (McMillan, 1993)). It automatically verifies if a system (which is expressed as a finite state machine) satisfies its specifications.

A NuSMV model describes system behaviour by declaring a set of variables. The initial values for these variables and how the variables change are explicitly defined. This description can be grouped into a set of modules with one main module. Modules are generally used to define or distinguish separate physical (sub) systems. A NuSMV module can consist of a set of variable declarations, assignments of variable initial values and definitions, and property assertions. The variable declaration section contains the local variables names and their types in the form '`variable_name : variable_type`'. Variables type can generally be of Boolean, numerical or enumerated types. The assignment section contains a set of assignments of variables into their initial value or its value in the next execution step, describing how a variable can change value. This can be expressed in the form '`variable_name := value`'. Various operators are available for variable assignments, including Boolean logic operators (`and, or, not`), conditional operators (`case, switch`), arithmetic (`+, -, *, /`), and

comparison (=, <, >, <=, >=). To assign a value of a variable in its initial and next execution step, operators `init` and `next` are used. The next value of a variable is defined using operators and constants from the range of values that the variable can have, as described in the declaration. Variables that do not have an assignment change non-deterministically. The assertion section is where safety properties (written in LTL or CTL) are defined, and these properties should hold over all executions.

Each module can also have input parameters (which are assigned outside the module) and output parameters (which are assigned inside the module). An excerpt of an example NuSMV model is presented below, showing relationship between input parameters and how they affect the internal variables:

```
MODULE functionF1 (inputParam1, inputParam2)
VAR
functionStates: {state1, state2, state3 };
functionEvent1: boolean;
functionEvent2: boolean;
ASSIGN
init (functionEvent1):= 0;
functionEvent2 := !functionEvent1;
functionStates := case
functionEvent1 & inputParam1 : state1;
functionEvent2 & inputParam2 : state2;
1: state3;
esac;
next(functionEvent1):=case
functionEvent1 = 1: 1;
1: {1,0};
esac;
```

AND states, however, require each of the state values to be defined independently as separate variables to allow the states to run simultaneously. For example:

```
VAR
state1: ...
state2: ...
state3: ...
```

defines that `state1`, `state2` and `state3` run in parallel, and each can hold value of its own (i.e. sub-modes, which will be discussed in the next section).

3.6.1.2 *Hierarchical Modelling in State Machines*

Contemporary systems are often required to perform large and complex functions in different stages of operation. For example, functions in an aircraft may vary from critical functions such as flight management, communications, and engine control to secondary electrical domestic and comfort/entertainment facilities. These functions involve large numbers of behavioural states, transitions, structural configurations, and interactions, and managing them is no trivial task.

One way to help the management of this large complex labyrinth of dynamic behaviour is through hierarchical modelling. Hierarchical modelling manages the decomposition of a state machine relating to a system by breaking it down into smaller parts, similar to those in static decomposition of systems and subsystems.

The activity of a state depends on the hierarchy of its parent-state. Hierarchy enables the states to nest, allowing the parent-state and sub-state relationships. (Drusinsky, 2006) outlines roles of hierarchy, mainly:

1) Refinement of states in a top-down manner
2) Reduction of transition clutter
3) Maintaining orthogonality, where parent-states are to be place holders for independent, irredundant activities (concurrency)
4) Enabling shared actions, where all sub-states shares the action of parent-state

Consider the example in the figures below. State machines in Figure 24 and Figure 25 describe the states and transitions in System S1. Both figures are semantically equivalent, but Figure 25, in which states belonging to State 3 are grouped, is more readable and less cluttered. Transition triggered by Event 8 in State 3.1 is required to be represented once in the parent-state State 3, as opposed to every sub-state in Figure 24. This significance is especially clear when there is need for the decomposition to be constructed into several levels (e.g. State 3.1.1, State 3.1.2 ...).

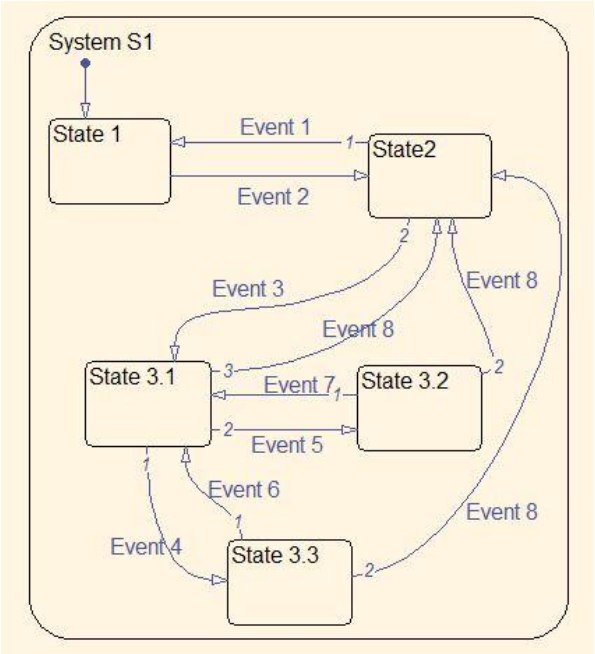


Figure 24: Simple state machine without hierarchy

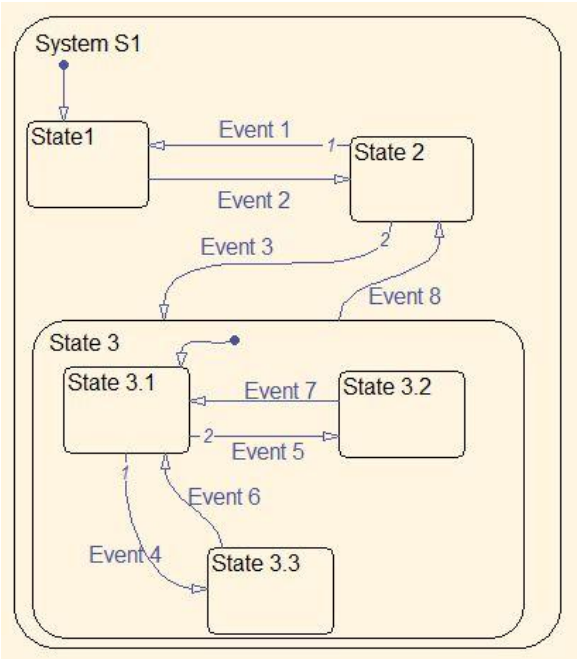


Figure 25: Simple state machine with hierarchy

Decomposition of states into sub-states is useful, but a state machine hierarchy should ideally also capture the physical and logical decomposition of the system into subsystems and components, which as mentioned earlier, can be represented in the functional model in the early stage or architectural model in the later stage of development. (Papadopoulos, 2000) describes how decomposition of a dynamic model

can be framed around the decomposition of its static structural model. For each (sub) system in the static hierarchy, a state machine is constructed to describe their behavioural transformations. For example, Figure 26 illustrates this relationship between static hierarchical model of System S and its subsystems, and their dynamic hierarchical model in state machines. System S can be structurally decomposed into subsystems S1, S2 and S3; while subsystem S1 is further decomposed into component C1, C2 and C3. The top level of the dynamic model represents the main operational states of the system S1, and transitions between them; the second level represents the behavioural states of the subsystems S1, S2, and S3. And the lowest level represents behavioural states of component C1, C2 and C3.

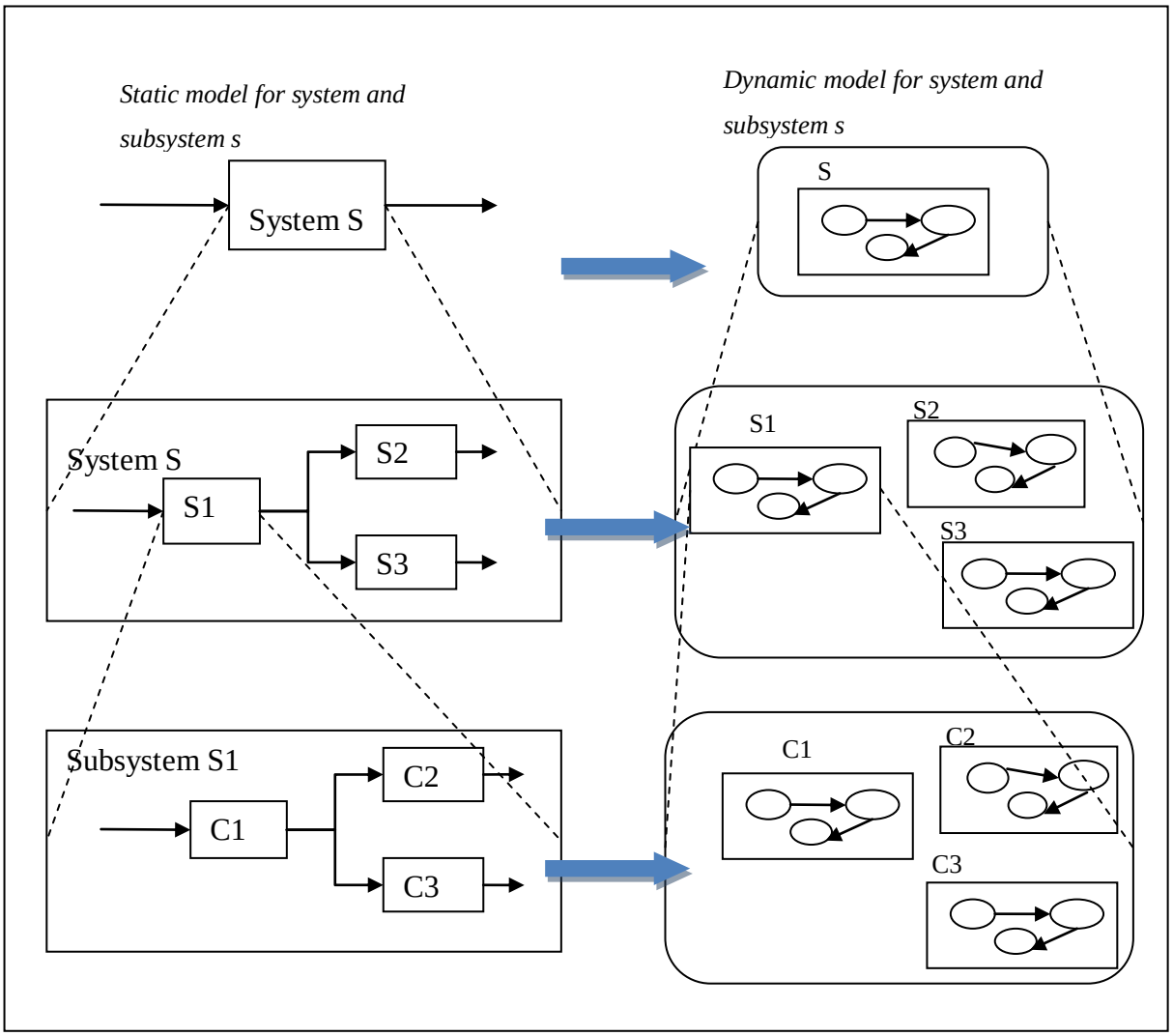


Figure 26: Relationships between static and dynamic models hierarchy of the system

3.6.1.3 *Hierarchical Modelling in NuSMV*

Hierarchy and decomposition in NuSMV models are managed through modules. The top-most module of the hierarchy needs to be declared as the `main` module. Modules, except for the `main` module, can instantiate multiple modules; and likewise a module can be instantiated by one or more other modules. Variables in a module can be local or global, and they can be accessed globally using path names.

For example, Figure 28 shows an excerpt of a NuSMV model representing the state machine shown in Figure 27. Sub-state `St1` is modelled in a separate module, and sub-state `St1a` can be referenced as `St1.St1a`. Events can be managed locally or globally. Events which are managed by other modules can be passed to corresponding modules as input parameters. Other parameters can be included to allow management of transitions and control. For example, additional variables can be assigned to manage activation of states (i.e. to inform sub-states whether parent-state is active) or to define which sub-state becomes active initially when the parent-state is activated. These allow transitions and control in AND/OR states to be managed accordingly.

Further discussion on semantics of NuSMV can be found in (Cavada *et al.*, 2005).

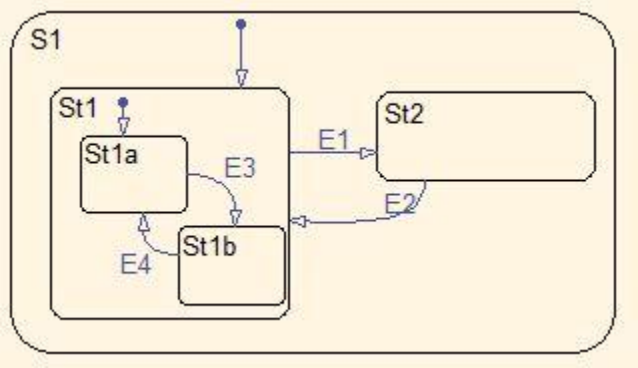


Figure 27: Sample state chart for S1

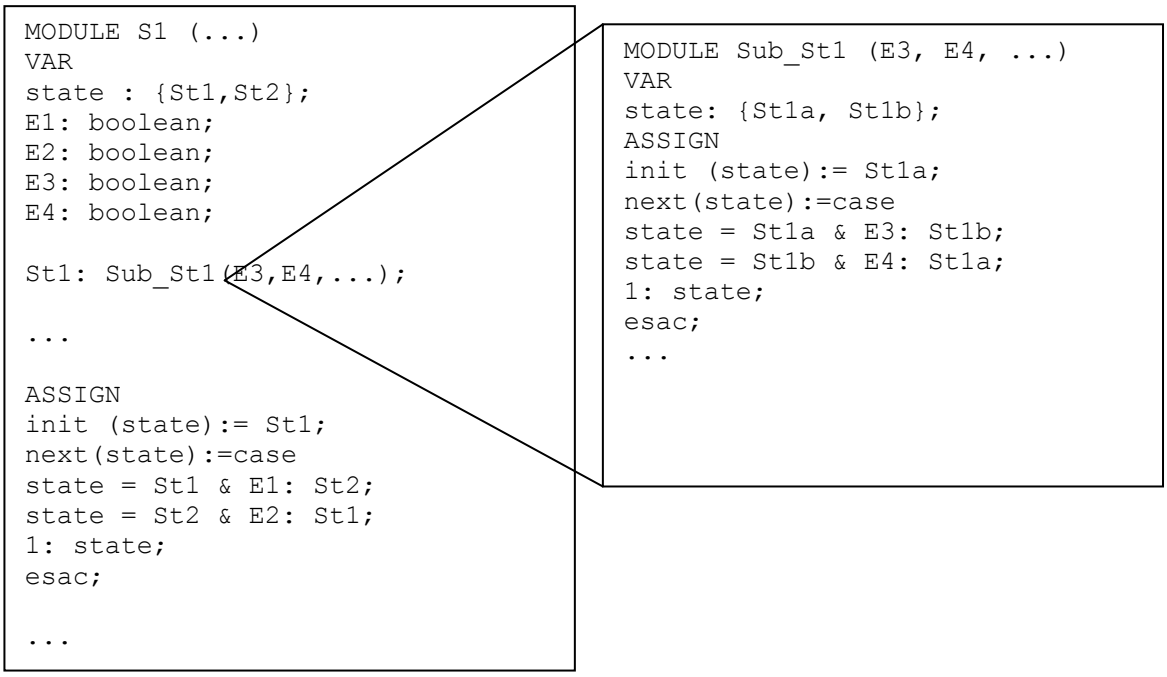


Figure 28: Modules to model hierarchy in NuSMV

3.6.2 Modelling the Dynamic Failure Behaviour of a System

The ability to capture system dynamic behaviour as part of the overall safety analysis process is vital because we need to understand how the system behaves not only during its normal operational conditions, but also in the presence of failures. In this section, we explore further the dynamic modelling of system failure behaviour, particularly in the construction of dynamic failure models and the information that can be obtained from their analysis.

Depending on the level of detail that they contain and their position in the IACoB design lifecycle, state machines in this discussion can be loosely grouped into two types: “abstract” state machine and more “refined” state machines. Abstract state machines are generally used to refer to the state machines that are created at early design stage (e.g. during development of functional model). Refined state machines refer to the state machines constructed at later stage (e.g. during development of architectural model). Construction of an abstract state machine generally focuses on modelling the delivery of the system functions. Construction of a refined state machine,

on the other hand, focuses more on exploiting hierarchical links and the topology of the system to manage the failure-related transition triggers between state machines of the components. These are also discussed further in the following sections.

3.6.2.1 ***Abstract Functional State Machine***

Early development process revolves around the construction and analysis of the functional model. At this stage, dynamic behaviour can be expressed as a set of different functional states of the system and transitions between them. A functional state in turn is defined by the set of functions delivered by the system in this state. (Papadopoulos, 2000) calls such states "modes" and defines a process for the construction and analysis of abstract state machines (or mode charts) that contain transitions among such modes.

This type of abstract state machine modelling plays a major part in the IACoB process as the introduction of safety-driven system mechanisms assisted by interpretation of the FMEA table brings new challenges in its safety analysis process. Failure in a function can cause occurrence of failure in other functions, or trigger the activation of other dormant functions. This in turn, changes the structure, interrelations and dependencies between the functions, and inevitably the failure propagation. The modelling and study of these new system dynamics pose new challenges for static assessment techniques like FTA. To address these problems and help model the dynamic behaviour, abstract state machines are used to describe the transition of the system from one state to another as the functional characteristics change. The advantage of including state machines here is twofold:

- Firstly, it helps to identify the fault tolerance mechanisms that can be introduced to the design by showing how the system can experience transition gracefully into the non-critical states after experiencing failures.

- Secondly, the abstract state machine captures dynamic system behaviour in a higher level manner. It retains the state/transitions information that enables it to provide input to formal verification/model checker tools.

3.6.2.2 *Mode chart to Represent Abstract State Machine*

Mode

In this thesis, the term *mode* is adopted to define the notion of a "*functional state in which the system maintains a stable functional profile*" as in (Papadopoulos, 2000). Mode is thus used to describe different phases of operations, in which the system behaves and functions in different ways. In a similar way, 'mode' is a more precise term that can be used to replace 'state' in an abstract state machine. Therefore it is adopted in this section to describe the application of an abstract state machine. The term '*mode chart*' is subsequently used instead of state chart to more precisely represent this type of state machine.

General types of mode that are used in the modelling of abstract state machines can be categorized as into the following:

1. Normal mode
2. Degraded mode
3. Failed mode

The system is said to be in normal mode when it *delivers its predefined set of functions*. Degraded mode describes the condition where a system delivers *part* of the intended functionality *safely*, whereas failed mode refers to the condition where there is complete loss of function or the system behaves in an *unpredictable* and *hazardous* manner. This implies that in cases when the system loses even only one of its many functions, if the lost function happens to be critical and has catastrophic effects on the system as a whole, the system is said to be in failed mode.

Modes and Their Roles in Fault Tolerance

Although this general classification of system modes is based upon delivery of functions, degraded and failed modes can be further divided into sub-categories as there are several well-established ways to categorize failures (and subsequently how these modes can possibly be further classified in relation to the response or nature of causative failures). For example, the general fault classification table presented in (Suri, 1995) outlines different types of faults according to different criteria such as: activity

(latent and active), duration (transient and permanent), causes (random and generic) and so forth. Here however, focus is placed on time-based classification. Degraded and failed modes can be categorized into *temporary* and *permanent*. A system is said to be in temporary degraded or temporary failed mode when the system has lost its normal functionality, but action can be taken to restore the normal mode. The system is said to be in permanent degraded or failed mode when it is no longer recoverable.

This classification and introduction of degraded modes is part of the effort to gain better understanding of the implementation of fault tolerance in early designs. Failures in (sub) systems should be compensated and managed in such way that their impacts leading to hazardous system failure are minimized. An abstract state machine is therefore designed to capture how degraded modes can act as potential buffers to divert hazardous failures.

One way to achieve this goal of fault tolerance is through introduction of redundant structures. In a more detailed design, redundancy can be implemented in the hardware, software, or information domain. For early design, we assume these are encapsulated as a more generic entity referred to as a *module*, which represents a function that can be refined accordingly into a system or component at a later stage.

Basic approaches to redundancy can be classified into static and dynamic redundancy. Static (also known as passive) redundancy does not detect or perform active action to control failures, but rather masks the failures to prevent failure propagation. Dynamic (also known as active) redundancy employs fault detection, diagnosis and reconfigurations. Hybrid redundancy combines both static and dynamic where masking is used to prevent propagation of failures and error detections, diagnosis and reconfigurations are also used to handle faulty components.

In static redundancy, modules are replicated according to the desired fault tolerance capability. Majority voting is typically used as the selection mechanism to decide on the correct output. To avoid single points of failure, voters can be duplicated and moved to the inputs of the modules.

Dynamic redundancy, on the other hand, uses less module duplication at the cost of heavier information processing. A minimal configuration consists of two modules (one main module and one standby module) performing the same functions. Fault detection and reconfiguration modules can be included for support. A fault detection module

monitors the outputs and when failure is detected in the main module, a reconfiguration module switches from delivering the output of the main module to delivering the output of the backup module. There are two types of standby in dynamic redundancy: hot standby (where standby module is continuously active) and cold standby (where standby module is activated only when needed).

In the context of the IACoB process, these fault tolerant mechanisms are often formulated after the CSA phase (FTA/FMEA). The construction of abstract state machines (and subsequently identification of degraded modes) essentially provides a state where these fault-tolerance strategies can be considered and taken into account into the overall system behaviour, and these strategies can be refined within the design progress.

Events and Transitions

Transitions between modes can be caused by:

- Normal events that cause the system to deliver different sets of functions. Such events cause a phase change in a phased-mission system.
- Failure events that causes the system to lose part or all of its functionality (e.g. normal transforming to degraded mode).
- Event that indicates restoration of functionality following failure (degraded modes back to normal mode).

Note that transitions are not only triggered by external events, i.e. stimuli from users and the environment. A transformation at a higher level of a mode chart can occur because of an event that occurs in the lower level, or by the occurrence of logical combinations of lower-level transitions. This allows us to capture the failure propagation of the system because as we move upwards from the lower level to the higher level, the mode charts capture how deviations or failure in the lower level (sub systems) affects the mode changes in higher level. Figure 29 illustrates this type of transitions triggered by internal events. At the higher level (level 1), the system changes its mode from normal to degraded when failure in subsystem S1 occur

(S1_Fail). Lower-level (level 2) state machines looks into how subsystem S1 reaches its fail mode after failure in C1 and C2 occur.

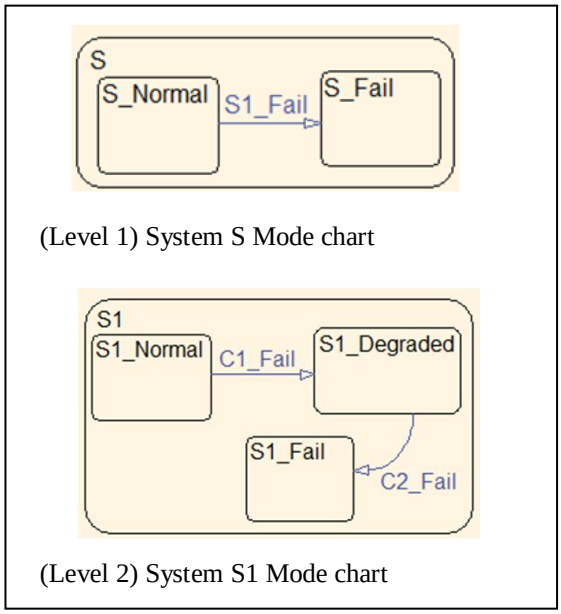


Figure 29: Mode charts showing high level and low level of system state transitions

Communications between mode charts can be established between different levels in these ways:

- Horizontal communication (peer charts) – transition results in an action that triggers other transitions in the same level of the chart.
- Vertical communication (Parents and sub charts) – transition in lower level mode chart triggers a transition in the higher level charts, and vice versa. Upwards communication where lower level charts can initiate an event that acts as the trigger for an action that activates a transition in higher level charts is common when the model aims to show how failures in subsystem trigger higher-level system failures.

One benefit of this organization technique is the ability to efficiently identify the relationship between transitions. Transition labels can be categorized according to the failure propagation points. One systematic method to identify possible failure-related

events that trigger transitions from one mode of a system to another is through the observation of the system's:

1. Peer-system's failure (same level) – a transition event in an immediate structural (input) block can trigger a transition event in another block.
2. Subsystem's failure (lower level) – a transition event in a lower level subsystem can trigger a transition event in a higher level system
3. Internal failure – internal malfunction of a block itself can be the basic cause of the transition.

3.6.3 Translation of FMEA Results to an Abstract State Machine

To construct system modes at an early functional level we need to identify system functional configurations and their possible transformations. To construct the events, we need to determine possible transitions between these configurations. At the same time, the FTA/FMEA results derived from previous stages provide information on failure relationship between functions. These results allow us to establish failure propagation and shows the effects (and criticality) of a failure event on the output function.

3.6.3.1 *FMEA-MODECHART Assistance Table*

Here we propose the construction of an assistance table to effectively identify and capture significant variables from the FMEA results for the main elements of mode chart. The table aims to organize information gathering from FTA/FMEA into a more systematic process of mode chart construction, as opposed to the traditional ad-hoc process. This assistance table is organized to identify: system modes, severity of each mode, output functions delivered in that mode, failure event(s) causing transition, and target mode(s) this transition leads to.

This information can be obtained from the previous IACoB processes. "Modes" are derived from previous FHA analysis where output function failures have been categorized according to their failure severity. The severity assessment process allows us to establish which function failures are tolerable, and which function failures are intolerable. It is then possible to categorize the delivery (or not) of these functions into

different modes. Failure to deliver functions that do not lead to hazardous effects is tolerable, and generally leads to degraded mode. Hazardous failure is intolerable and leads to a failed mode. This essentially allows us to establish graceful degradation for the system in the presence of failures.

The "Functions Delivered" column outlines lists of (output) functions delivered in the particular mode. This information can be obtained during the grouping of modes according to the functions delivered. "Functional Failure Causing Transition" describes the type of failure event that can occur (i.e. deviation in each of the corresponding output function). This information can be obtained from each of the output function which has been annotated with failure behaviour. Finally, "Target Mode" defines the mode a particular failure event leads to during a transition.

With this key information (modes and events which trigger transformations) now gathered in the assistance table, the process of constructing the abstract state machine is relatively straightforward.

Table 5 shows an example of an assistance table for the sample system presented in Figure 18. The first mode identified is System_Normal, where all output functions (Function F7, F8 and F9) are delivered. Each output function is susceptible to an omission failure which results in the inability of the system to deliver the particular function. From the earlier FFA, Function F8 is identified as a critical function, and this brings us to the second mode, System_Degraded. In System_Degraded mode, output function F8 is delivered regardless of the condition of function F7 or F9. System mode goes to System_Fail when omission in Function F8 occurs. Please note that even in the System_Degraded or System_Fail mode, Function F7 and/or Function F9 can still be delivered. It is also possible to include other degraded modes to further define the delivery (or not) of Function F7 or Function F9 if necessary.

Table 5: FMEA-ModeChart Assistance Table

Mode	**Severity**	**Functions Delivered**	**Functional Failure Causing Transition**	**Target Mode**
System_Normal	-	Function F7	Omission of Function F7 (O-F7)	System_Degraded
		Function F8	Omission of	System_Fail

			Function F8 (O-F8)	
		Function F9	Omission of Function F9 (O-F9)	System_Degraded
System_Degraded	Marginal	Function F8	Omission of Function F8	System_Fail
System_Fail	Hazardous	-	-	-

Figure 30 illustrates an example of the mode chart which can be constructed based on the assistance Table 5 above.

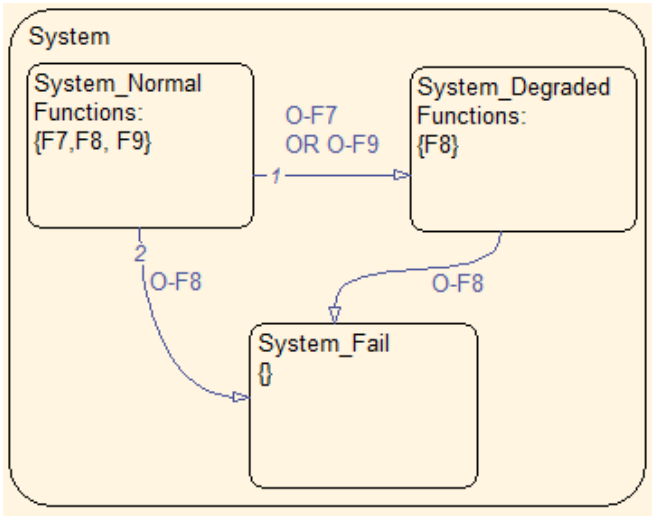


Figure 30: Example of mode chart constructed from FMEA-ModeChart Assitance Table

It is important to note that events that can occur in a mode are not limited to the ones listed in "Functional Failure Causing Transition". This column helps to draw focus on events that are significant enough to affect delivery of functions (thus causing transition between modes). In some cases, it is possible to have dormant failures in a mode where occurrence of a failure doesn't cause transition from a mode until another failure occurs. This can potentially cause almost-immediate transition from normal mode to fail mode. One way to manage this is by fully taking into account all possible failure occurrence in a mode and if necessary, by creating another intermediate degraded mode to manage dormant failure (for example, where alarm was raised) so that system does not move from normal to fail mode in succession. An example of this is shown in Chapter 4.

3.6.4 Translation of HiP-HOPS Failure Annotations to a Refined State Machine

Once the information on lower level components become more available, it is possible to refine transition triggers in the once-abstract mode charts. At this stage, it is no longer as significant, although it is still possible, to define the 'modes' based on delivery of component outputs (compared to the definition of mode according to the delivery of system functions earlier).

While the analysis of FTA/FMEA in earlier stages and the use of assistances table can help in the generation of an abstract state machine, the generation of a more refined state machine at a later stage involves a slightly different approach. This is because, unlike abstract state machines, most information required for failure-relevant transitions in more refined state machines can be obtained directly from HiP-HOPS component failure annotations.

It is important to note that our mode charts are not tied to any commercial state chart tool. It is possible to use available commercial tools like Matlab Stateflow or Statemate to provide graphical description. Converter tools are available (sf2smv (Banphawatthanarak *et al.*, 1999), (Bobbie, 2001), stm2smv (Loer, 2003), or mdl2smv (Juarez-Dominguez *et al.*, 2008)) to convert state machine models from these commercial tools into model checking input models. While the use of intuitive interface (Barfield, 2004) and graphical tools is helpful for acceptance, (Schatz *et al.*, 2002) highlights that it is not essential for the concept. Also, to perform model checking, the state machines eventually need to be converted into model checking input models.

For these reasons, here we explore how the more refined mode-charts representing behaviours of lower-level designs can be expressed directly as a NuSMV model.

Each component block is represented as a module in NuSMV. Information flow between blocks of components can be modelled through the use of module parameters. These parameters provide links between the output (port) of a source component to the input (port) of a target component. In a similar manner, these input parameters are also used to relay and model the failure propagation between components. It is important to

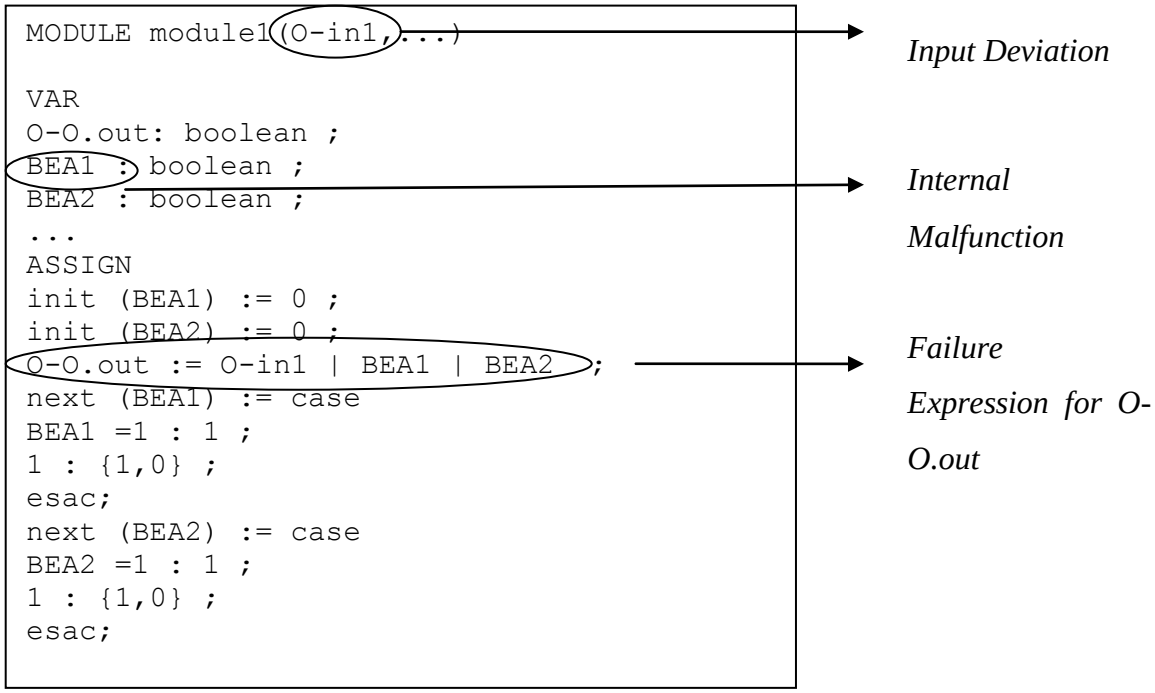
note that because multiple failure types could occur at one output, all of the deviations need to be passed to the target component input parameters.

The basic of HiP-HOPS failure expression can be represented as:

Output_deviation = Internal_malfunction AND/OR Input_Deviation

This expression can be incorporated into the NuSMV model by assigning the output deviations as the corresponding module internal variables. These output deviations are passed along as module parameters to other components/modules at the receiving end of the information flow. Internal malfunctions are defined within modules too, and once an internal malfunction occurs, it is assumed to be persistent throughout the entire run, unless correcting event is specified and triggered. Input deviations for failures propagated by other components are defined through the modules input variables.

For example, given HiP-HOPS failure expression: `O-O.out = O-in1 + BEA1 + BEA2`, the generated NuSMV model from the annotated HiP-HOPS model can be seen in the following NuSMV model excerpt. Output deviations and each component basic events are declared as Boolean data types. All basic events are initialised to hold value 0 as the system starts operation in normal mode. Lastly, as basic events are assumed to be permanent, its next value will remain as '1' if the current value is '1'.



It is perhaps important to note that the current translation from HiP-HOPS annotation to NuSMV model is performed manually. And because the extracted behaviour is based on failure annotation, the NuSMV module produced is essentially a failure-extended model (error model). This model can eventually be extended to include further relevant nominal behaviour, or be integrated with nominal model if the nominal model was developed in parallel (please see chapter on future work).

3.6.5 Refinement of Events to Maintain Traceability

Refinement refers to the process of providing a system solution with more details or precision in an incremental development process. This includes the process of adding more constraints and developing details of system/component attributes. Refinement of a design often traverses abstraction levels and captures sub systems. In a later design phase, these subsystems are further refined by adding more constraints, including non-functional aspects, and by improving the model solutions.

The refinement process will affect both structural and behavioural elements of a system. Dynamic behavioural models need to reflect and capture refinement of behavioural decomposition. Structural refinement is captured through the decomposition of the physical system into sub-systems. As discussed earlier, structural decomposition in a

NuSMV model is represented as individual NuSMV modules. Similarly, refinement for behavioural models can be achieved through the decomposition of modes into sub-modes.

A systematic management of the decomposition process helps to provide good traceability. Traceability refers to the ability to maintain and navigate the relations between different stages of the model and manage that information. Such relations should allow designers/analysts to follow the evolution of the design more closely and establish connections between earlier and later design models.

We believe that - in addition to facilitating decomposition - a systematic process of event refinement (especially those events relating to failure behaviours) contributes to better traceability. One way to achieve this is through clear communication and linking between events in earlier abstract design models and more detailed events in later models. Well-established traceability between early and later models is particularly useful in situations where errors are discovered through model checking, in which case it is possible to trace errors to earlier design decisions and eventually investigate and re-evaluate their effects on high-level design assumptions and goals.

Here we aim to provide methodological guidelines to assist event refinement systematically. This can be achieved by two main approaches: 1) refinement of events through minimal cut sets and 2) refinement of events through compositional annotation. They are discussed further in the following sections.

3.6.5.1 ***Refinement of events through minimal cut sets***

The first possible way to refine a state machine is by replacing the transition event expression with its causing events. As the transition events are losses of functions or malfunctions which form top events of fault trees in HiP-HOPS, the causing events can be effectively obtained and mapped from HiP-HOPS FTA/FMEA results. For each top event, its minimal cut sets can essentially be used to form the replacement expressions.

This approach works well in several scenarios. It is appropriate for situations where focus is placed more on the verification of behavioural modes in higher level abstract systems compared to behavioural modes in refined individual subsystems. This usually

means that the verification process aims to explore the effects of the lower level subsystem failures on the modes changes at the higher level, instead of exploring the nominal dynamic behaviour for each of the subsystem.

The following example is presented to illustrate this further. Figure 31 presents an abstract model that describes system A. System A is then gradually refined into subsystems A1, A2, A3, and A4. The refinement allows us to update the abstract dynamic model for system A to take into account failure events occurring in the lower level subsystems.

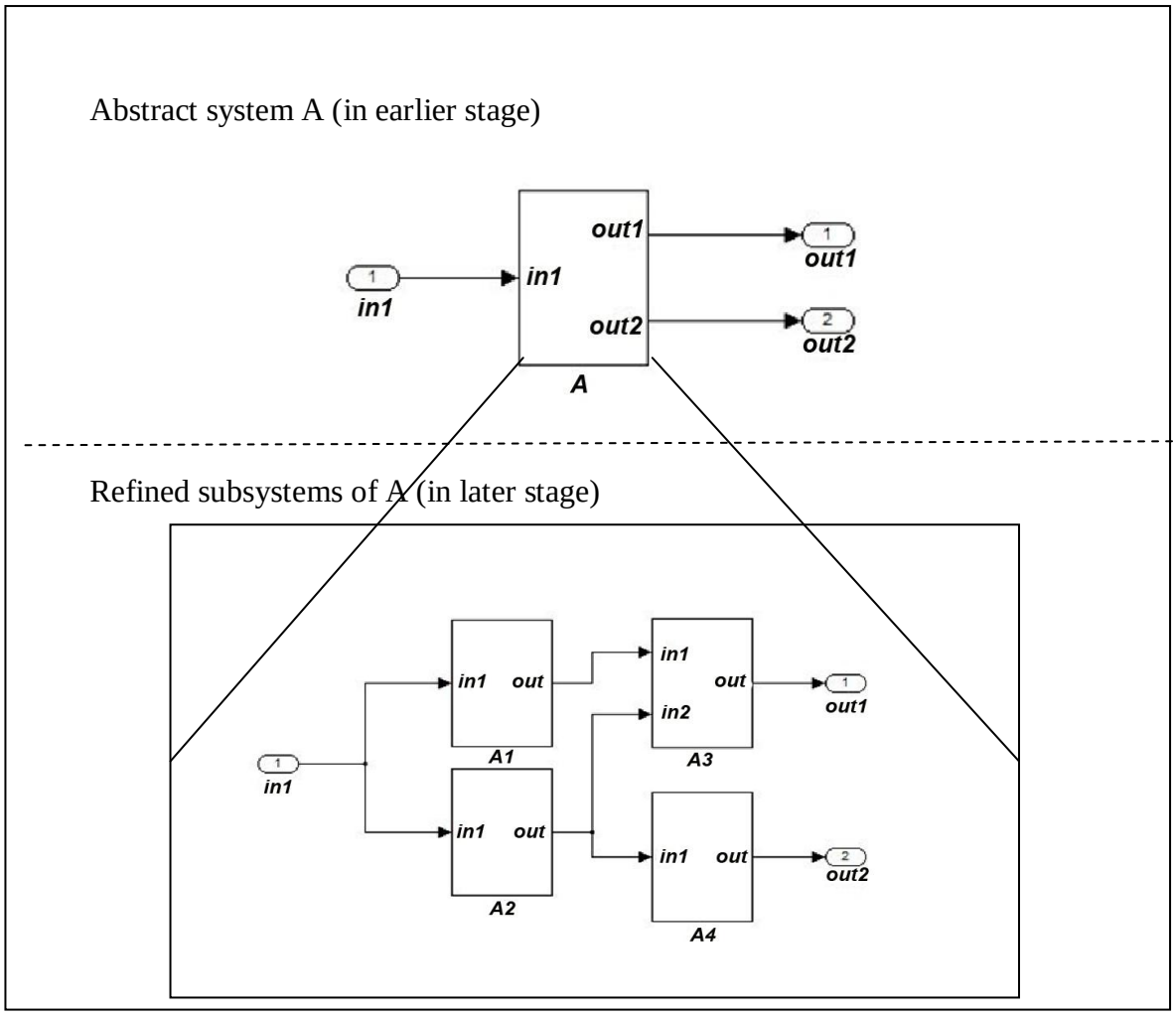


Figure 31: Refinement for system A

Presented in the following is the failure information of system A and subsystems A1, A2, A3, A4. Subsystems A1,A2 and A4 are assumed to simply have one failure type, Omission (O-A1, O-A2, O-A4 respectively) caused by corresponding internal failures BEA1, BEA2, BEA4. Subsystem A3 has both Omission and Value failure types which are caused by internal failures BEA3 and VBEA3 respectively. The following table summarizes the failure behaviour:

Table 6: Failure behaviour for System A and Subsystems A1, A2, A3, A4

System / Subsystem	Internal Malfunctions	Output Deviations	Description of Output Deviation	Causes of Output Deviation (Output Deviation Failure Expression)
A	-	O-A.out1	Omission deviation in output 1 (out1) of system A	O-A3.out
		O-A.out2	Omission deviation in output 2 (out2) of system A	O-A4.out
		V-A.out1	Value deviation in output 1(out 1) of system A	V-A3.out
A1	BEA1	O-A1.out	Omission deviation in output (out) of subsystem A1	BEA1
A2	BEA2	O-A2.out	Omission deviation in output (out) of subsystem A2	BEA2
A3	BEA3	O-A3.out	Omission deviation in output (out) of subsystem A3	BEA3 OR (O-A3.in1 AND O-A3.in2)
	VBEA3	V-A3.out	Value deviation in output (out) of subsystem A3	VBEA3
A4	BEA4	O-A4.out	Omission deviation in output (out) of subsystem A4	O-A4.in1 AND BEA4

System A can be operated in several abstract functional modes, namely Mode1, Mode2, Mode3, Mode 4 and Mode5. System A starts with nominal Mode1 when there is no failure occurrence. From Mode1, it either moves to Mode2 when O-A.out1 occurs, or moves to Mode3 when O-A.out2 occurs. If both O-A.out1 and O-A.out2 occur, it moves to Mode4. Mode5 occurs when system A experiences a V-A.out1 failure. We assume that the severity analysis process has identified Mode4 to be hazardous. The abstract mode chart for this abstract model can be seen in Figure 32:

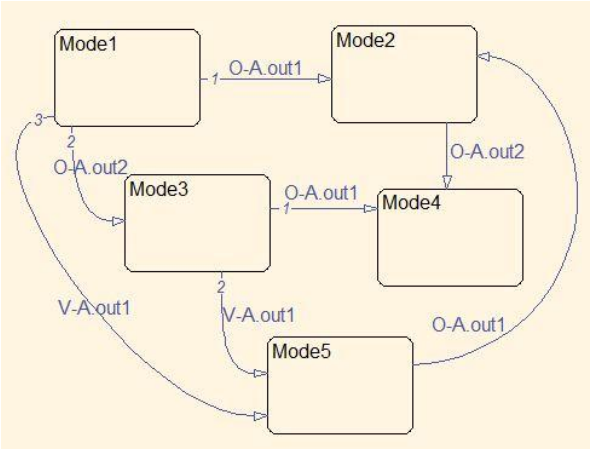


Figure 32: Mode chart for system A

From the tabulated failure information in Table 6, HiP-HOPS produces the following minimal cut sets for each of the system output failures:

O-A.out1 = {BEA1.BEA2 , BEA3}

O-A.out2 = {BEA2.BEA4}

V-A.out1 = {VBEA3}

These analysis results allow us to refine the abstract state machine of system A (Figure 32) into a more refined state machine (Figure 33) which takes into account failure propagations of its subsystems in the event transitions. The event transitions are now expressed fully in terms of the components internal malfunctions.

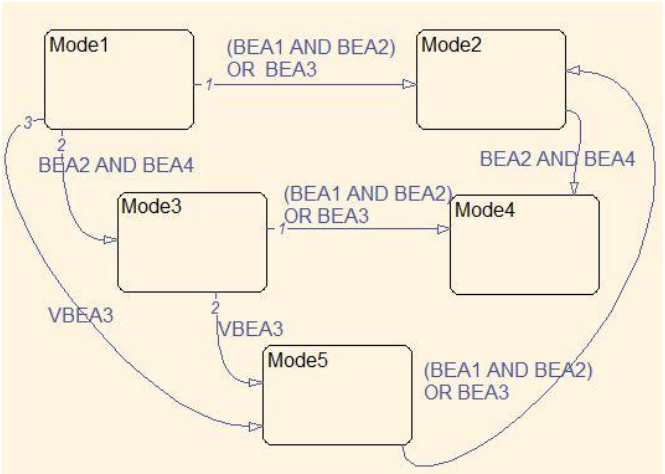


Figure 33: Refined transitions for System A

Subsequently, the expression of mode charts in the NuSMV input language enables us to verify various (generic) requirement specifications , for example "*If failure in subsystem A4 does not occur, hazardous Mode4 shall not occur*", which in this example, is relatively straightforward.

$$AG(!BEA4 \rightarrow !(SystemMode = Mode4));$$

The checking of this property can also be arguably performed through manual analysis of FMEA results table to establish the effects of causing events (and their combinations) on the corresponding output deviations. For example, by manually working through the FMEA table to decide if any combination of all basic events without BEA4 can lead to failure "O-A.out1 AND O-A.out2" (Mode4). It is also possible to perform this via FTA by studying the minimal cut sets. However, this could become inconvenient for larger systems with more complicated modes. The translation into the model checking input language allows verification to be done more quickly and automatically. In addition to that, this formal analysis via model-checking is also able to take into consideration other nominal behaviour which is not captured by the FMEA.

With this approach, focus is placed on the abstract high-level system mode chart, which is often sufficiently contained within the NuSMV Main Module. One of the advantages of adopting the results from CSA is the easy representation of both deviations and component basic events in NuSMV as Boolean data types. Here, failure logic is used instead of success logic, meaning that instead of defining output(s) of the system according to the outputs of subsystems, output deviations are defined by failures in subsystems. This is done by assigning to it the corresponding minimal cut sets.

One downside of this approach is the fact that focus is placed on the abstract mode chart and how the occurrence of internal malfunctions affects the abstract system modes. Little attention is placed on the other non-failure relevant behaviour of subsystems (although they can be included if necessary). Also, these internal malfunctions are modelled within the main module (therefore not benefiting from any hierarchical structure). An example of the generated NuSMV model from an annotated HiP-HOPS model can be seen in the following NuSMV model excerpt:

```
MODULE main

VAR
```

```
BEA1 : boolean ;
BEA2 : boolean ;
BEA3 : boolean ;
Mode : {Mode1, Mode2 };
...
ASSIGN
init (BEA1) := 0 ;
init (BEA2) := 0 ;
init (BEA3) := 0 ;
...
init (Mode) := Mode1;
O-O.out := BEA3 | BEA2 & BEA1  ;
next (BEA1) := case
BEA1 =1 : 1 ;
1 : {1,0} ;
esac;
next (BEA2) := case
BEA2 =1 : 1 ;
1 : {1,0} ;
esac;
next (BEA3) := case
BEA3 =1 : 1 ;
1 : {1,0} ;
esac;
next (Mode):= case
Mode = Mode1 & O-O.out : Mode2;
1: Mode;
...
```

3.6.5.2 ***Refinement of Events through Compositional Annotation***

This approach extends the previous approach by focusing not only on the abstract high level mode chart, but also by modelling each subsystem's behaviour in its own module. It captures and reflects the functional hierarchy by constructing independent mode charts and NuSMV modules for each function (subsystem), which in turn allows non-failure related behaviours of each subsystem to be effectively modelled and considered in their roles of contributing to system failures.

To effectively link failure behaviour to input modules and capture the structural topology, transition events (labels) are maintained in a similar structure similar to the ones in HiP-HOPS failure annotations for system and subsystem output deviations. This means they are expressed in terms of input deviations and internal malfunction events.

To illustrate this approach using the previous example (Figure 31), the failure behaviour for System A and Subsystem A1, A2, A3, A4 can be modelled in the following mode charts, each capturing their failure expressions in the transition labels:

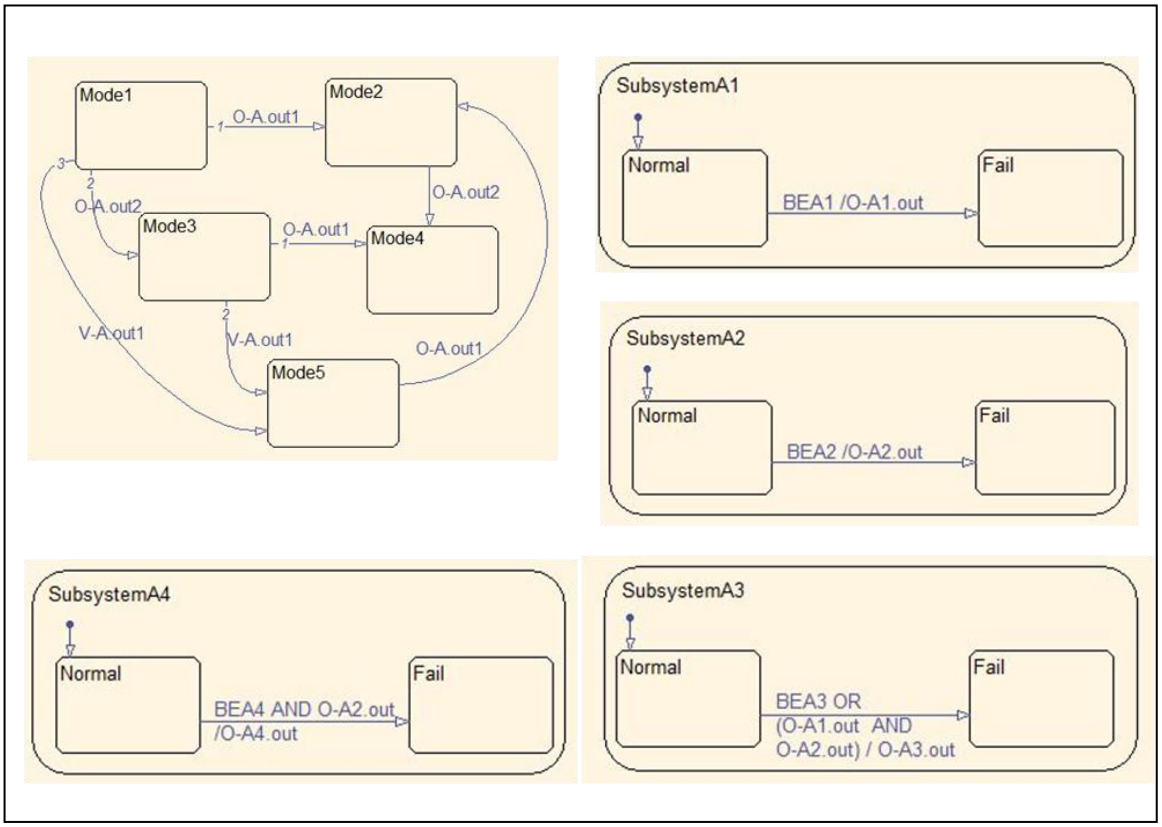


Figure 34: Mode charts for system (and subsystems of) A

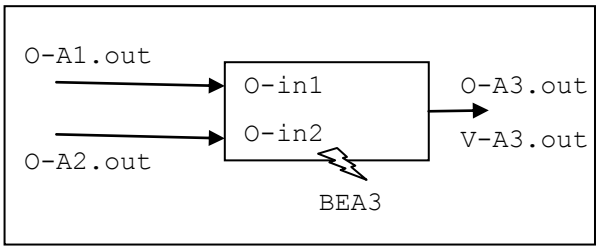
The compositional failure annotation in HiP-HOPS also allows systematic generation of a NuSMV model for the system and subsystems. Each NuSMV module contains information about internal malfunctions, input deviations and the definition of output deviations, all of which are obtainable from component failure annotations. Like HiP-HOPS, flow of information is obtained through the structural topology. The 'higher-level' module `SystemA` manages these connections and the flow of information between subsystems by passing the output variables of a source subsystem to the input parameters of target subsystem during module initiation. This allows linking between components to be established and subsequently connect input deviation to corresponding output deviations.

Figure 35 to Figure 39 illustrates the connection between components annotated with HiP-HOPS failure data and their corresponding NuSMV models which shows the hierarchical structure and failure propagations of these subsystems:

```
MODULE SystemA

VAR
A1: SubsystemA1;
A2: SubsystemA2;
A3:  SubsystemA3  (A1.O-A1.out,  A2.O-
A2.out);
A4: SubsystemA4 (A2.O-A2.out);
Mode  :  {Mode1,  Mode2,  Mode3,  Mode4,
Mode5};
O-A.out1 : boolean;
O-A.out2 : boolean;
V-A.out1 : boolean;

ASSIGN
init (Mode) :=Mode1;
O-A.out1 := A3.O-A3.out;
O-A.out2 := A4.O-A4.out;
V-A.out1 := A3.V-A3.out;

next(Mode) := case
Mode = Mode1 & O-A.out1 : Mode2;
Mode = Mode1 & O-A.out2 : Mode3;
Mode = Mode1 & V-A.out1 : Mode5;
Mode = Mode2 & O-A.out2 : Mode4;
Mode = Mode3 & O-A.out1 : Mode4;
Mode = Mode3 & V-A.out1 : Mode5;
Mode = Mode5 & O-A.out1 : Mode2;
1: Mode;
esac;
...
```

Figure 35: NuSMV model for system A

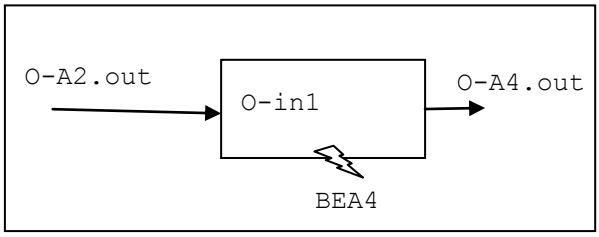
Failure propagation of SubsystemA3




```
MODULE SubsystemA3 (O-in1, O-in2)

VAR
O-A3.out: boolean ;
V-A3.out: boolean;
BEA3 : boolean ;
VBEA3 : boolean ;
Mode: {Normal, Fail};

ASSIGN

init (BEA3) := 0 ;
init (VBEA3):= 0 ;
init (Mode) := Normal;
O-A3.out := BEA3 | (O-in1 & O-in2) ;
V-A3.out := VBEA3 ;

next (BEA3) := case
BEA3 =1 : 1 ;
1 : {1,0} ;
esac;

next (VBEA3) := case
BEA3 =1 : 1 ;
1 : {1,0} ;
esac;

next(Mode) := case
Mode = Normal & O-A3.out : Fail;
1: Mode;
esac;
```

Figure 36: Failure propagation for subsystem A3

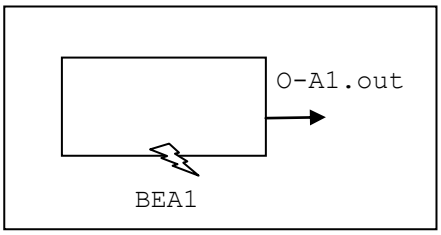
Failure propagation of SubsystemA4




```
MODULE SubsystemA4 (O-in1)

VAR
O-A4.out: boolean;
BEA4: boolean;
Mode: {Normal, Fail};

ASSIGN
init (BEA4) := 0 ;
init (Mode) := Normal;
O-A4.out := BEA4 & O-in1;

next (BEA4) := case
BEA4 =1 : 1 ;
1 : {1,0} ;
esac;

next(Mode) := case
Mode = Normal & O-A4.out :
Fail;
1: Mode;
esac;
```

Figure 37: Failure propagation for subsystem A4

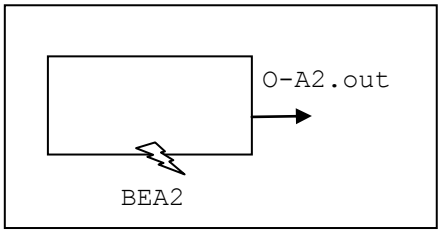
Failure propagation of SubsystemA1




```
MODULE SubsystemA1

VAR
O-A1.out: boolean ;
BEA1 : boolean ;
Mode: {Normal, Fail};

ASSIGN
init (BEA1) := 0 ;
init (Mode) := Normal;
O-A1.out := BEA1;

next (BEA1) := case
BEA1 =1 : 1 ;
1 : {1,0} ;
esac;

next(Mode) := case
Mode = Normal & O-A1.out : Fail;
1: Mode;
esac;
```

Figure 38: Failure propagation for subsystem A1

Failure propagation of SubsystemA2




```
MODULE SubsystemA1

VAR
O-A2.out: boolean ;
BEA2 : boolean ;
Mode: {Normal, Fail};

ASSIGN
init (BEA2) := 0 ;
init (Mode) := Normal;
O-A2.out := BEA2;

next (BEA2) := case
BEA2 =1 : 1 ;
1 : {1,0} ;
esac;

next(Mode) := case
Mode = Normal & O-A1.out : Fail;
1: Mode;
esac;
```

Figure 39: Failure propagation for subsystem A2

One of the benefits of systematic establishment of connections and failure propagation is a better traceability between design models. It allows the designer to visit earlier models to see if they are still correct and re-examine any design decisions that were made based on the analyses of these systems. This subsequently enables verification of early hypotheses as details become more available. If errors are discovered during model checking on these NuSMV models, it is possible to trace the errors to earlier design decisions and investigate the effects of early assumptions.

In summary, the input, process and output of this process are:

Input	• Functional model annotated with failure information • FMEA results showing relationship between failures
Process	• Identification of modes (states) and events • Construction of state machines from FTA/FMEA results
Output	• System state machines • NuSMV model

3.7 Application of Model Checking

Once state machines are constructed and translated into the model checking input language, formal methods can be used to validate system requirements.

The background of model checking has been introduced in chapter 2. In this section, attention is primarily drawn to the value of model checking as part of IACoB in early design stages; classes of requirements and properties that can be verified; and common errors discovered through early application of model checking.

General steps involved in this process are:

1. Generation of a NuSMV model
2. Creation of a specification that defines a property which is required of the model
3. Running of model checker
4. Model checker produces confirmation statement if the property holds or produces a counter example if the property is breached
5. Based on the results of the model checker, analysis takes place to determine whether modifications are required for:
    i. Design of model
    ii. Formulation of properties

System specifications and requirements which are expressed in temporal logic can generally be classified into different categories. (Bérard *et al.*, 2001) distinguishes these properties that can be verified by a model checker into: *reachability, safety, liveness* and *fairness* properties.

3.7.1 Reachability

Reachability properties define that a particular configuration φ (a state in the Kripke structure) of the model can be reached. Three possible variations of such properties can be distinguished, as shown in the following CTL logic:

- Reachability from the current state: `EF` φ

- Reachability from any execution state: `AG EF` φ
- Reachability from under some condition σ: `E[`σ U φ]

3.7.2 Safety

Safety properties define that under certain conditions, configuration φ never occurs. This can only be proven if all execution paths are explored, therefore a CTL logic that can be used to specify this is:

```
AG (! φ)
```

A common application of a safety property is the analysis of mutual exclusions. For example, a CTL expression that specifies σ and φ are to be mutually exclusive:

```
AG !(σ ∧ φ)
```

Safety properties can also be used to formulate configurations where a desired property holds:

```
AG φ
```

3.7.3 Liveness

Liveness properties define that under certain conditions, a Kripke state where configuration φ holds will eventually be reached.

One common application of a liveness property is the analysis of response to configurations. For example, a specification which states that whenever σ holds, eventually a state must be reached where φ holds:

```
AG (σ -> AF φ)
```

It is also possible to specify in a liveness property that whenever σ holds, some responding state where φ holds will be visited within *m* to *n* time units. Time in NuSMV can only be measured qualitatively in terms of execution steps. Abounded liveness property specifies when some response is required. In NuSVM, this bounded liveness properties can be formulated in CTL:

```
AG (σ -> ABF m..n φ)
```

3.7.4 Fairness

Fairness properties define that under certain conditions, states where some property φ holds will occur infinitely often.

Fairness properties are expressed in LTL, and are not expressible in CTL format because it is not possible to specify that some expression holds repeatedly (Huth & Ryan,2000). In NuSMV, fairness constraints can be introduced with the inclusion of "`FAIRNESS φ;`" which corresponds to

```
GF φ
```

in LTL, this expression defines that state φ holds continuously without interruption.

3.7.5 Common Errors Discovered Through Model Checking

Considering the fact that the general application of model checking has been primarily targeted at mature design models, it is important to understand and determine its values at earlier design stages. (Miller *et al.,* 2003) and (Tribble & Miller, 2003) presented case studies which demonstrate that formal models can be effectively used to find errors before implementation of the system. One common error found through model checking is inaccuracy in the original requirements (or how it was phrased). This generally leads to modifications to refine the requirements to be more specific and accurate.

Other errors could involve situations where more than one input arriving at the same time and in combination drives the model into an unsafe state. There are several ways

to deal with simultaneous input events, for example: stating a rule whereby only one input variable can change in any step (Miller *et al.*, 2003). Additional logic is included to assign priority to multiple events, and when simultaneous input events occurs, lower level priority events can either be discarded or stored in a queue for processing in succeeding steps.

(Juarez-Dominiguez, 2008) also highlights the importance of model checking in detecting hazardous interactions between system features (e.g. software components). These software components which control mechanical components are often developed in isolation, and their combination can sometimes cause unexpected or undesired system behaviour. Model checking can be used to detect these hazardous combinations in a design.

In chapter 4 and 5, we demonstrate how in practice model-checking can be usefully employed to verify or not the satisfaction of properties on behavioural models constructed using IACoB method.

3.8 Potential for Automation

Currently, the translation process between the different models in IACoB is performed manually. In the context of a larger, more complex system, this can become an error-prone process. To address this, we note the potential for automation in IACoB. The key aspects of the process which can be automated include the translation from HiP-HOPS annotated model to NuSMV model.

The construction of NuSMV models from HiP-HOPS annotated model can be achieved by mapping the failure information as discussed in section 3.6.4. As mentioned previously, this results in a failure-extended NuSMV model (error model). Basic states and transitions can be assigned by default for each module. The basic states generally include Normal state to describe states where component functions as intended, and Fail state(s) to describe states where failure(s) of component occurs. The default transitions between these basic states are described using the corresponding events which are the causes of the failure.

The following is a sketch of hips2smv algorithm. It presents steps which can be used to describe the translation process from HiP-HOPS annotation to the NuSMV model.

For each component, a NuSMV module is created within which the following steps are performed:

- Step 1: Identify input parameters

The input parameters of a NuSMV module are the input deviations of that component. To identify these input deviations, we use HiP-HOPS fault tree synthesis algorithm which provides a record of the deviations for each input port of the component.

- Step 2: Declare the internal variables

Internal variables which can be assigned automatically from HiP-HOPS models typically include the internal malfunctions, output, and output deviations. These are declared as Boolean data type.

- Step 3: Specify initial values for the internal variables

Initial value of internal malfunction and output deviations are set to 0 by default, reflecting the assumption that the system starts from normal state.

- Step 4: Define output deviation

Output deviation is defined according to the failure expression provided in HiP-HOPS annotation. It is described in terms of basic events and input deviations.

- Step 5: Specify next value for internal variables

The ‘next’ notion in NuSMV relates current and next state variables to express transitions. As mentioned previously, once internal variables occur (set to 1), the next value stays at 1 as it is persistent throughout the entire run. Next value of output deviation can also be defined here in relation to current value of internal malfunction and input deviations.

In addition to modules which represent components in the system, a MAIN module is also constructed for each NuSMV model to:

- Construct instances of all component modules.
- Define the connections between components modules. This is achieved by connecting the parameters of each component module's input ports (and supplying them as input parameters) to the corresponding output ports of other module which is connected to it.

Refinement of these state machine transitions (using ways described in sections 3.6.3 and 3.6.4) can also be automated. The algorithm for refinement through minimal cut sets works by constructing one main NuSMV module to model the internal malfunctions of all components. The initial state of the system is set to Normal, and all internal malfunctions are set to be absent. The output deviation is defined in terms of its minimal cut sets generated from the FTA, as opposed to defining it in terms of input deviation and internal malfunctions.

The refinement through compositional annotation can be achieved using the algorithm described above where one NuSMV module is constructed for each component. This way, the structural, hierarchical and failure propagation information are retained.

This automation is particularly useful for refined state machine when establishing failure connections between components are more crucial than describing system states and therefore it is sufficient to use basic states and transitions.

However, there are also several aspects of the process that require human intervention. In general, human intervention is required to obtain information on the system dynamic behaviour which is not captured in the initial CSA model. This may include:

1) Description of system states

   In addition to the basic states (Normal and Fail states) which are automatically assigned, other classifications of system states (for example, degradation states) may be required. These inputs need to be manually specified and defined. This is particularly important in the early abstract state machines where degradation states play important roles in the understanding of system high-level behaviour, which are not captured in CSA's HiP-HOPS model.

2) Description of system transitions

   In addition to basic transitions (from normal to fail states), specification on how the system moves from one state to another (for example, to a degraded mode) also need to be specified.

3) Requirement specification

   Requirement specification needs to be manually provided by the analysts in terms of CTL.

These processes which require human intervention can be assisted with improved support tool, for example, by extending current editor tool, storing frequently-used specifications in a library, or by introducing graphical tool for state machine (please see future work session).

3.9 Chapter Summary

In summary, this chapter describes the IACoB process which consists of a number of key phases, along with the following main activities involved in each phase:

Phase	**Input**	**Process**	**Output**
Construction of system functional model Or in later stage, architectural model	Requirements Or in later stage, a less-refined model	Identify, define and relate functions Translate requirements into functional model Or in later stage, refinement of model	Functional model Or architectural model
Severity assessment of output function (or component)	System functional model (or architectural)	Estimate risk and classify the severity of output function (or component) failures based on their consequences	Severity analysis of output functions (components) Narrowed focus on higher priority functions (or components)
Establishing local failure behaviour	System model	Establish failure information for each functional (or architectural) block	Functional model with data information

			Establish causes of output failure
Fault tree and FMEA synthesis and analysis	System model with local failure behaviour	Generate FTA and FMEA for system model Identify weak points in system design	Effects of failure on output function (or component) Better understanding of the criticality of function (or component)
Generation of state machine	System model annotated with failure information FMEA results showing relationship between failures	Identify modes and events Construct state machine from FTA/FMEA results	System (abstract) state machines NuSMV models
Model Checking	NuSMV model	Apply model checking to verify system	Affirmation or counterexample

The application of the whole process is illustrated in the next chapter.

Overall, IACoB combines the advantages of compositional safety analysis such as simplicity, efficiency and scalability, with the benefits of formal verification such as the ability to perform verification of safety requirements on dynamic models of the system. This helps increase confidence in the design and leads to an improved model-based safety analysis process compared to the reliance of only one technique. In terms of identifying potential failures, the part of IACoB which employs CSA focuses on the relationship (causes and effects) of failures between components. The application of BSA can potentially further uncover errors as it takes into consideration component dynamic nominal behaviour (and their interaction with failures).This can potentially uncover new failures which have not been anticipated in CSA. For example, weakness in the design of logical connections or flow of information between components. This is illustrated in the case study presented in Chapter 5.

CHAPTER 4. Case Study on Brake-by-wire

Functional and Behavioural Analysis of a Brake-by-wire System

This chapter demonstrates application of the IACoB method. We present a case study which explores the design and analysis of a simplified brake-by-wire system for cars. The study produces safety analyses which help us to gain better understanding of this system. The analysis deliberately starts from a simple model, where fault-tolerant functions and other well-established heuristics for good design in such systems have been omitted. The idea is to demonstrate how the proposed process could systematically help designers arrive at such measures.

The case study consists of two main system models, namely a purely functional model and a model where functions have been allocated to an architecture of components, to exemplify different application stages for the process.

The case study is structured as follows: in section 4.1, the vehicle brake-by-wire system is introduced. The safety assessment of this system is discussed in 4.2. Section 4.2.1 to 4.2.2 describe the construction and analysis of fault trees, FMEA and a mode chart for the design. Safety requirement properties are discussed in section 4.2.4 and the system design is checked against the predefined list of properties. Section 4.2.5 discusses the ways to refine transition events in the mode chart by exploiting results from FMEA. This is followed by section 4.3, which presents a scenario where a more detailed architectural model for the system is derived. In sections 4.3.1 and 4.3.2, both single failure and multiple failures FFA, FTA/FMEA are performed for this revised model, the relevant mode chart is constructed and the system is again verified against safety requirements.

4.1 Introduction to Brake-By-Wire System

Brake-by-wire systems are a recent drive-by-wire technology in the automotive industry. Drive-by-wire technology employs electronic control systems which use

electromechnical actuators to replace traditional hydraulic and mechanical control systems. Brake-by-wire systems replace traditional automotive braking components (like brake boosters, pumps, and master cylinders) with electronic sensors and actuators. Although the application of brake-by-wire is not as widely commercialized in relative comparison to its *x*-by-wire counterparts (*x* representing commanded action) like throttle-by-wire, or steer-by-wire (Langenwalter, 2004), many experts believe that brake-by-wire systems will eventually become common in the future (Carley, 2004). Nossal and Lang (2002) presented a model-based approach to building an *x*-by-wire application.

Brake-by-wire systems can be classified into two types: brake-by-wire with hydraulic backup and brake-by-wire without hydraulic backup. Brake-by-wire with hydraulic backup, also called Electric Hydraulic Brake (EHB) is realized through hydraulic pumps and additional electrically controlled valves. If the electronic control fails, the complete electric hydraulic system will be deactivated and the brake system will behave like a pure hydraulic system which delivers only emergency brake function with reduced brake force. Brake-by-wire without hydraulic backup is often known as Electric Mechanical Brake (EMB). EMB transfers electrical commands generated through the driver to computer controlled electro-mechanical actuators. EMB does not possess the fail-safe mechanics of hydraulic backup, and therefore must be developed with strict fault tolerant properties.

The brake-by-wire system used in this case study is based upon a model provided by Daimler, but also draws from designs in (Hedenetz & Belschner, 2008) and (Colombo, 2008). The system consists of one vehicle-level processor and four local-level wheel processors. The vehicle-level processor reads in brake command input from the driver, communicated through a human-machine interface (for example, the brake pedal or parking brake interface), and subsequently generates braking command for each local-level wheel processor based on high-level advanced brake functions such as an Anti-Lock Brake System (ABS) or Electronic Stability Program (ESP). This braking command is broadcasted using two replicated data buses. Local-level wheel processors are located physically close to the wheels. Upon receiving braking command from the vehicle-level processor, each local-level processor calculates the value of braking pressure, taking into consideration various local-level information including actuator

position and speed. This value of braking pressure is then fed to an actuator which then applies the actual braking pressure on the corresponding wheel of the car. These functions are distributed using the Time-Triggered Communication Protocol (TTP) (Hedenetz & Belschner, 2008) which is especially designed for safety-related applications. The system is usually powered by two independent power supply units. To maintain the simplicity of this example, communication architecture and power supply units are not included in the discussion. The physical configuration of the brake-by-wire system is illustrated in Figure 40, which depicts the system general topology.

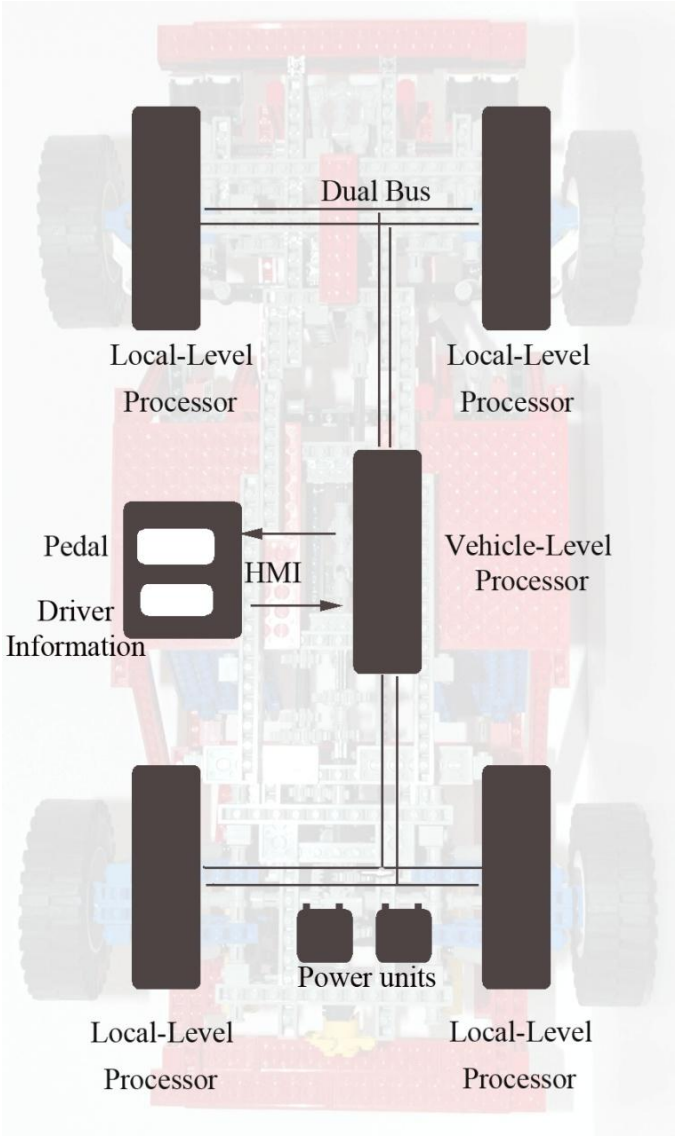


Figure 40: General topology of Brake-By-Wire system

4.2 Analysis of System Functional Models

In accordance with the IACoB method, we start the safety assessment process from a high-level functional model. For this simplified system, two initial main functions can be delivered: 1) Function which delivers basic braking 2) Function which delivers braking with driving assistance anti-lock (ABS). These two functions can arguably be combined into one as they are not physically distinct. In this early model, however, they are free from architectural detail and are modelled as two separate logical functions to facilitate the illustration of function delivery. If required, these functions can be combined with a conjoining function.

The Matlab Simulink model illustrated in Figure 41 represents a high-level abstraction of the brake-by-wire system. It is simplified to consist of input functions, braking command processing functions (vehicle-level and local-level), ABS command processing function, and output functions. As local-level processing provides identical function for each wheel of the vehicle, we assume it is sufficient to discuss and analyze one (instead of all four) in this initial model. There are four input blocks which read in driver's initiated braking demand from brake pedal (*Input_brakeDemand*), readings for wheels' speed (*Input_wheelSpeed*), external variable readings (*Input_external*) , and local-level feedback (*Input_local*). Information on brake demand, wheel speed and external environment is passed to the vehicle-level processing function (*VehicleLevelProcessing*) which calculates and generates the independent brake commands for each local-level processing (*LocalLevelProcessing*). It also relays the information needed for ABS calculation to the *ABSProcessing* function. The wheel local-level processing controls the output functions which provide basic braking or ABS braking. This early model does not yet incorporate any fault tolerance mechanisms.

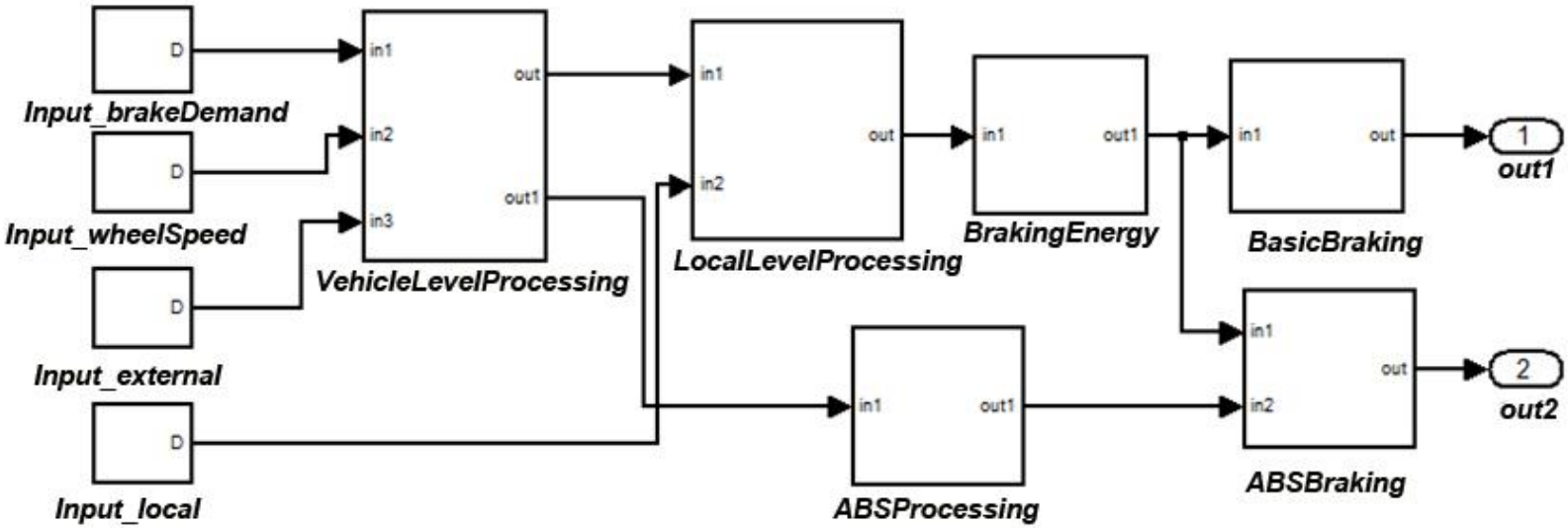


Figure 41: Abstract functional model for Brake-By-Wire

4.2.1 FFA

Once the model is constructed, we proceed to perform the FFA on the system. The main aim of this process is to classify and analyse the effects and severity of failures in the output functions, *BasicBraking* and *ABSBraking*. In this case the focus is placed on the omission and commission failure types, although it is also possible to perform analysis on value or timing failures. The following Table 7 presents an extended FFA which includes identification of detection, potential recovery plan and recommendation columns for each failure.

Table 7: Functional failure analysis of Brake-By-Wire

Function	**Failure Type**	**Effects on System**	**Severity**	**Detection**	**Recovery Plan**	**Design Recommendation**
BasicBraking	Omission	No brake force ; vehicle cannot be stopped; driver loses control.	Catastrophic	Using pressure feedback	Not possible	Redundant back up mechanism should be introduced
BasicBraking	Commission	Vehicle tends to drift; loss of stability	Critical	Comparing pedal input (demand) and pressure feedback	Release Pressure	Commission failure should not be allowed to propagate
ABSBraking	Omission	Loss of steerability ; less efficient brake	Marginal	Using feedback on wheel speed and pressure	Not possible	Situation can be compensated by driver
ABSBraking	Commission	No brake force available	Catastrophic	Comparing wheel speed and pressure feedback	Switch off ABS function	Commission failure should not be allowed to propagate

From the examination of this FFA table, it can be seen that the severity of an omission failure of function *BasicBraking* (O-BasicBraking) is categorized as having a catastrophic effect, and therefore should be mitigated with fault tolerant design. The second functional failure related to the provision of braking pressure is commission.

The commission failure in *BasicBraking* function (C-BasicBraking) is identified as having critical consequences and therefore should not be allowed to propagate and influence other functions in the wrong way. One way to achieve this is by detecting the commission failure, forcing the system to fail silent and then handling the omission accordingly by putting a fault-tolerant mechanism in place. The failure for the *ABSBraking* function is categorized as having catastrophic severity in its commission failure and marginal effects in its omission failure. This is due to the nature of the *ABSBraking* function which provides driving assistance rather than those of imperative role in braking. This suggests that it is more favourable for the function to fail in omission, and therefore the function should fail-silent when commission failure is detected.

To perform FTA and FMEA, these functional blocks are annotated with failure behaviour before being analyzed by HiP-HOPS. Table 8 summarizes the internal malfunction of each of the function. To maintain the simplicity of this example, output blocks are modelled to be free from internal malfunctions, and instead can only propagate failures.

Table 8: Functional blocks internal malfunctions

Function	Failure Mode	Description
Input_brakeDemand	BDBE	Internal malfunction in function which reads in brake demand, causing Omission failure.
Input_wheelSpeed	WSBE	Internal malfunction in function which reads in wheel speed, causing Omission failure.
Input_external	ESBE	Internal malfunction in function which reads in external measurements, causing Omission failure.
Input_local	LSBE	Internal malfunction in function which reads in local actuator measurements, causing Omission failure.
VehicleLevelProcessing	VLPBE	Internal malfunction in vehicle-level processing function or which causes Omission failure. Most probably a hardware failure.
	VLPBEc	Internal malfunction in vehicle-level processing function which causes Commission failure. Most probably a software failure.
	VLPBEabs	Internal malfunction in vehicle-level processing function which causes ABS to be absent.

	VLPBEabsC	Internal malfunction in vehicle-level processing function which causes anti-lock ABS to be instantiated without intention.
ABSProcessing	ABSBE	Internal malfunction ABS processing function which causes anti-lock ABS to be absent.
LocalLevelProcessing	LLPBE	Internal malfunction in wheel local-level processing function which causes Omission failure.
	LLPBEc	Internal malfunction in wheel local-level processing function which causes Commission failure.
Braking Energy	ActBE	Internal malfunction in Braking Energy which causes Omission failure.
	ActBEc	Internal malfunction in BrakingEnergy causes Commission failure.

4.2.2 FMEA

As discussed in the earlier chapter, once the model has been annotated with its local failure information, fault trees can be generated and analyzed, and an FMEA can be obtained automatically using HiP-HOPS tool. The following Table 9 summarizes the FMEA results. The table defines how failures in other functional blocks propagate and contribute to failure O-BasicBraking, O-ABSBraking, C-BasicBraking and C-ABSBraking. As the initial design does not include any fault-tolerant strategies, the table shows us how each internal malfunction in every function can become direct contributors to the omission and commission failures of the braking and ABS functions.

Table 9: FMEA for Basic Brake-By-Wire functions

Function	**Failure Mode**	**Direct Effect**	**Severity**	**Comments/ Recommendation**
Input_brakeDemand	BDBE	O-BasicBraking	Catastrophic	Redundancy required
Input_external	ESBE	O-ABSBraking	Marginal	
Input_local	LSBE	O-BasicBraking	Catastrophic	Redundancy required
Input_wheelSpeed	WSBE	O-ABSBraking	Marginal	-
VehicleLevelProcessing	VLPBEabs	O-ABSBraking	Marginal	-

	VLPBE	O-BasicBraking	Catastrophic	Redundancy required
	VLPBEc	C-BasicBraking	Critical	Should fail silent
	VLPBEabsC	C-ABSBraking	Catastrophic	VLPBE should not propagate and when detected, ABS should be deactivated.
LocalLevelProcessing	LLPBE	O-BasicBraking	Catastrophic	Redundancy required
	LLPBEc	C-BasicBraking	Critical	Should fail silent
BrakingEnergy	ActBE	O-BasicBraking	Catastrophic	Redundancy required
	ActBEc	C-BasicBraking	Critical	Should fail silent
ABSProcessing	ABSBE	O-ABSBraking	Marginal	-

To implement a more robust design, several advisable design changes can also be determined from an analysis of the FMEA table above. These are recorded in the recommendation column. Recommendation and Severity for each function correspond and reflect the severity and recommendation of the output function failures they cause. One important (and most obvious) technique to achieve fault-tolerance is the introduction of redundancy in the '*module*'. Module here refers to function for functional model or components for the more refined architectural model.

As an industry common practice, fault tolerant design for brake-by-wire systems can be implemented through either the inclusion of a hydraulic system (in an EHB system) or through replicated electronic components (in an EMB system). For this example, we introduce a hybrid system which implements both hydraulic as well as redundant electronic modules (with lower numbers of redundant modules compared to a pure electronic EMB). Due to the cost and space constraints in automotive x-by-wire systems, it is often important to reach a compromise between the degree of fault tolerance and the number of redundant components (Isermann, 2004). For this example, it is assumed that it is sufficient for us to adopt duplex (i.e. consisting of two elements) redundant structure, which would enable a system to tolerate single-point failures.

The analysis of FMEA in Table 9 therefore provides an insight that assists us in distinguishing critical functional failures that contribute to failures which have

catastrophic or critical consequences (O-BasicBraking, C-BasicBraking, C-ABSBraking) from those that contribute to failures with marginal effects (O-ABSBraking). This knowledge subsequently allows us to establish the appropriate resource management priority and design improvement. For example, we learnt that the failure in input blocks that detects braking demand (*Input_brakeDemand*) could have more severe consequences (causing O-BasicBraking) than other input blocks (*Input_external*) which failure only lead to O-ABSBraking.

First we examine the input blocks. Two input blocks, the *Input_brakeDemand* function and the *Input_local function*, are identified to be the contributing causes to O-BasicBraking which is catastrophic, and therefore it is necessary to configure these functions to be at least fail-operational by introducing a redundant module to backup each function. As mentioned earlier, failure in *Input_external* and *Input_wheelSpeed* only lead to O-ABSBraking and therefore in this example, will be tolerated. We also identified that there is a need to introduce redundant function for *VehicleLevelProcessing* as its failure also leads to O-BasicBraking. Additionally, *LocalLevelProcessing* can be connected directly to the function *Input_brakeDemand* to read raw braking command. This way, in the occurrence of a failure in the *VehicleLevelProcessing* function, basic braking command can still be obtained. Similarly, an omission failure in basic braking caused by internal malfunction in *LocalLevelProcessing* and *BrakingEnergy* can be mitigated by introducing redundant functions to support these critical functions.

In addition to this independent redundancy for individual modules, we could also include a hydraulic function which acts as the group backup mechanism to provide emergency braking in the presence of failures that affect the electrical-based functions.

Commission failures on both braking and ABS functions have been identified as critical and catastrophic respectively. It is therefore recommended that any function which leads to commission failure should fail-silent instead. This can be achieved by deactivating or switching off the function whenever commission failure is detected. This, in turn, transforms the commission failure into omission failure, which will then be treated accordingly.

To manage the redundancy for omission failure, we look into the redundancy technique mentioned earlier in Chapter 3. In our case, duplex dynamic redundancy configuration is adopted for two of the input functions, the vehicle-level processing, the local-level processing, and the braking energy functions. Figure 42 shows an example of redundant configuration for *VehicleLevelProcessing* which consists of main function *VLP A* and backup function *VLP B*. Third module *VLP O* is used to monitor their outputs, and in the case of failure, select to relay the correct output. To maintain the simplicity of this example, *VLP O* is assumed to be reliable enough to only propagate failures; and therefore their failure behaviours are not modelled. In practice, safety monitoring components like *VLP O*, although practically more reliable (with lower failure rate compared to modules they monitored), possess their own failure behaviours. More discussion on failure behaviours of fault-monitoring modules and detectability properties are presented in the chapter 6.

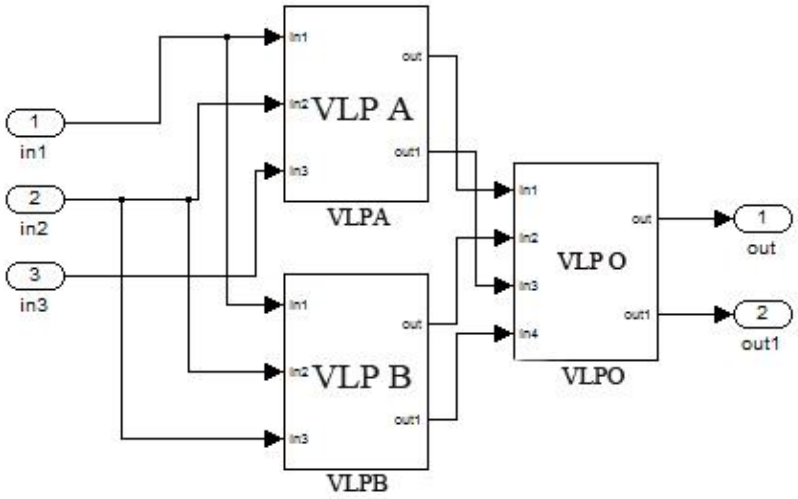


Figure 42: Redundant module for Vehicle-Level Processing function

The complete backup scheme structure for each function can be found in Appendix A. Figure 43 illustrates the revised model with backup components incorporated. Dark-coloured blocks signify redundancy. To summarize, several key changes as a result of the examination of FMEA in Table 9 are:

- Inclusion of redundant functions employing duplex configuration for *input* functions, *VehicleLevelProcessing, LocalLevelProcessing, and BrakingEnergy*

- Transformation of commission failure to omission failure in *VehicleLevelProcessing, LocalLevelProcessing, and BrakingEnergy.*
- *LocalLevelProcessing* can be connected directly to *Input_brakeDemand*
- Introduction of hydraulic backup mechanism.

These key changes illustrate the strength and contribution of CSA towards the improvement of the system design, in particular, the identification of the system critical points. By identifying and addressing design weakness in these critical points early, a more robust revised design can be formulated before the design progresses further.

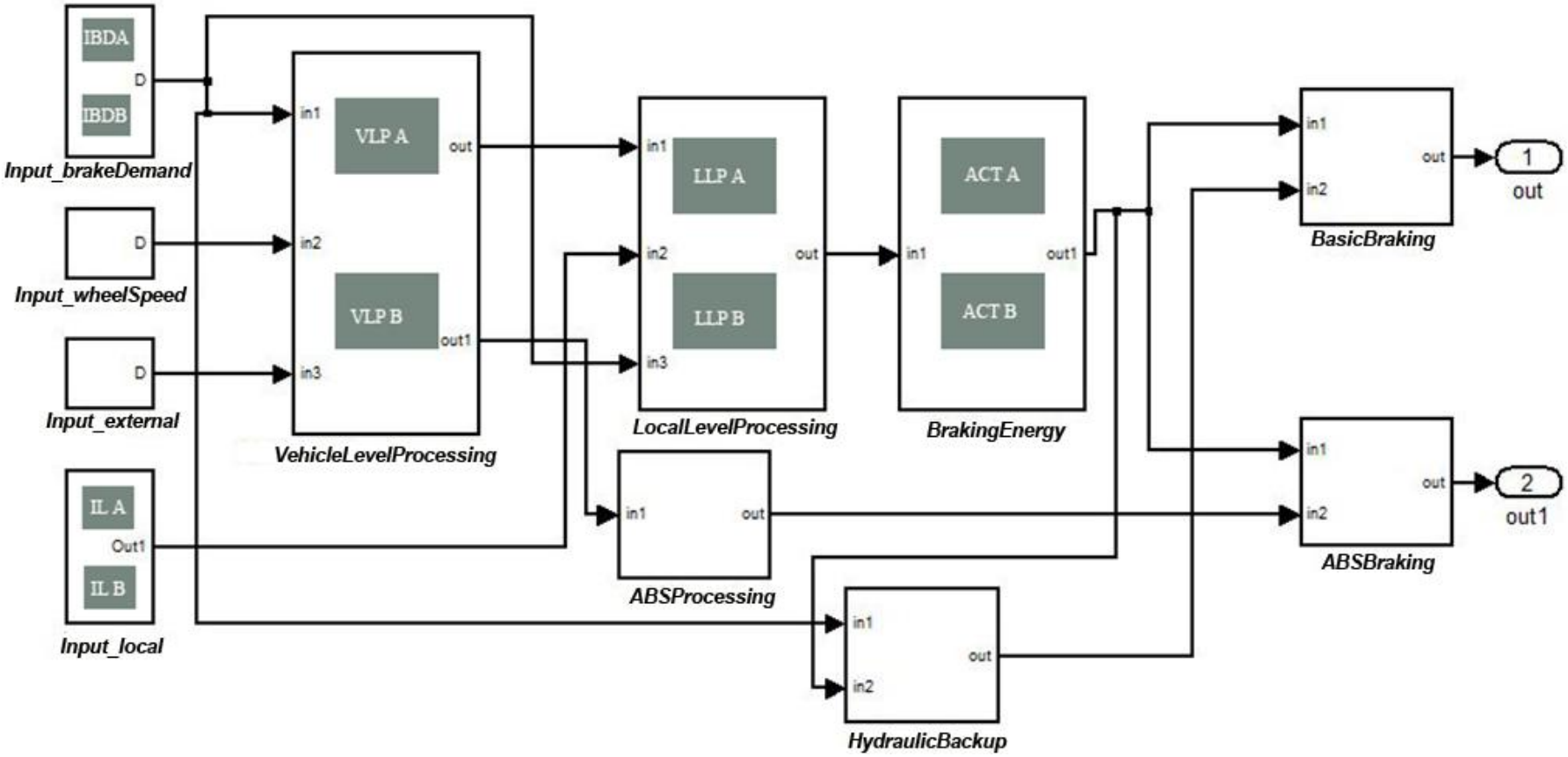


Figure 43: Revised model with duplex redundant mechanism

The inclusion of these new redundant mechanisms results in the introduction of new failure behaviours, which requires the FTA and FMEA to be updated. The new fault-tolerant redundant structure means that there are no longer any single-point failures which directly cause O-BasicBraking. As there is no backup for functions *Input_external* ,*Input_wheelSpeed*, and *ABSProcessing* they are shown to directly cause direct effect to O-ABSBraking in the updated FMEA table. The rest of the functional failures which causes O-BasicBraking in combination with other functional failures are recorded in FMEA Table 11:

Table 10: Direct ffects FMEA for revised model

Function	Failure Mode	Direct Effects
ABSProcessing	ABSBE	O-ABSBraking
Input_external	ESBE	O-ABSBraking
Input_wheelSpeed	WSBE	O-ABSBraking

Table 11: Further Effects FMEA for revised brake-by-wire for failure O-BasicBraking

Function	Failure Mode	Further Effects	Contributing Failure
ACT A	ActBEc	O-BasicBraking	ActB.ActBEc AND HydraulicBackup. HBBE
			ActB.ActBE AND HydraulicBackup. HBBE
	ActBE	O-BasicBraking	ActB.ActBEc AND HydraulicBackup. HBBE
			ActB.ActBE AND HydraulicBackup. HBBE
ACT B	ActBEc	O-BasicBraking	ActA.ActBEc AND HydraulicBackup. HBBE
			ActA.ActBE AND HydraulicBackup. HBBE
	ActBE	O-BasicBraking	ActA.ActBEc AND HydraulicBackup. HBBE
			ActA.ActBE AND HydraulicBackup. HBBE
HydraulicBackup	HBBE	O-BasicBraking	ActA.ActBE AND ActB.ActBE
			ActA.ActBE AND

			ActB.ActBEc
			ActA.ActBEc AND ActB.ActBE
			ActA.ActBEc AND ActB.ActBEc
			Input_local1.LSBEa AND Input_local1.LSBEb
			LLPA.LLPBE AND LLPB.LLPBE
			LLPA.LLPBEc AND LLPB.LLPBE
			LLPA.LLPBE AND LLPB.LLPBEc
			LLPA.LLPBEc AND LLPB.LLPBEc
Input_brakeDemand.IBDA	BDBEa	O-BasicBraking	Input_brakeDemand.IBDB. BDBEb
Input_brakeDemand.IBDB	BDBEb	O-BasicBraking	Input_brakeDemand.IBDA. BDBEa
Input_local.LSA	LSBEa	O-BasicBraking	Input.localSensor.LSB.LSBEb AND HydraulicBackup.HBBE
Input_local.LSB	LSBEb	O-BasicBraking	Input.localSensor.LSA.LSBEa AND HydraulicBackup.HBBE
LLP A	LLPABE	O-BasicBraking	LLPB.LLPBE AND HydraulicBackup.HBBE
			LLPB.LLPBEc AND HydraulicBackup.HBBE
	LLPABEc	O-BasicBraking	LLPB.LLPBE AND HydraulicBackup.HBBE
			LLPB.LLPBEc AND HydraulicBackup.HBBE
LLP B	LLPBBE	O-BasicBraking	LLPA.LLPBE AND HydraulicBackup.HBBE
			LLPA.LLPBEc AND HydraulicBackup.HBBE
	LLPBBEc	O-BasicBraking	LLPA.LLPBE AND HydraulicBackup.HBBE
			LLPA.LLPBEc AND HydraulicBackup.HBBE

4.2.3 Construction of Mode charts

FTA and FMEA can be iterated until the design model meets early predefined requirements, for example until a satisfactory level of redundancy configuration is achieved (i.e. system tolerant to n number of failures). In this case study, we assume that elimination of single point failures for O-BasicBraking is sufficient. As FTA and FMEA results have shown this, the design is deemed to be acceptable for the next stage of the process. This allows us to proceed and model the design dynamic behaviour by constructing an abstract state machine.

To construct the state machine, it is first of all, important to identify the primary elements: abstract states (as discussed in previous Chapter 3, referred to as 'modes') and transition events. Modes are derived based upon provision of system functions, which in this case are the *BasicBraking* function and the *ABSBraking* function. Each of the functional failures in the Table 7 then causes a transition to degraded or failed modes. Corrective measures can be identified through FFA or the fault trees which explore the causes of the failure. In general, these potential treatments can be classified into three categories: untreatable failures, failures that always require identical treatments, failures that require different treatments depending on root causes. In this example at this stage, the recovery plan is not taken into consideration, and therefore is not modelled in the mode chart.

1) *Normal (BBW_Normal)* mode where both Braking and ABS functions are delivered

2) *Permanent Degraded (BBW_PD)* mode where basic Braking is delivered, but ABS function can no longer be delivered

3) *Fail (BBW_Fail)* mode where no braking pressure is delivered.

The table summarizes system modes, related severity (whether mode is hazardous), functions delivered, potential functional failures that could occur in that mode, transition these failure could cause and the target mode after transition.

Transitions can be formulated according to the failures that could occur to each of the functions; in this case, all such failures are of omission type as commission failures have been transformed into omissions by design. As explained in section 3.8, default

modes (BBW_Normal and Fail) can be automatically assigned. Degraded mode BBW_PD and its corresponding transitions, however, need to be manually described.

Table 12 summarizes three modes the system that can be derived by considering the delivery of functions in which:

1) *Normal (BBW_Normal)* mode where both Braking and ABS functions are delivered

2) *Permanent Degraded (BBW_PD)* mode where basic Braking is delivered, but ABS function can no longer be delivered

3) *Fail (BBW_Fail)* mode where no braking pressure is delivered.

The table summarizes system modes, related severity (whether mode is hazardous), functions delivered, potential functional failures that could occur in that mode, transition these failure could cause and the target mode after transition.

Transitions can be formulated according to the failures that could occur to each of the functions; in this case, all such failures are of omission type as commission failures have been transformed into omissions by design. As explained in section 3.8, default modes (BBW_Normal and Fail) can be automatically assigned. Degraded mode BBW_PD and its corresponding transitions, however, need to be manually described.

Table 12: FMEA- ModeChart Assistance Table

Mode	Severity	Functions Delivered	Functional Failure Causing Transition	Target Mode
BBW_Normal		ABSBraking	O-ABSBraking	PD
		BasicBraking	O-BasicBraking	Fail
BBW_PD	Marginal	BasicBraking	O-BasicBraking	Fail
Fail	Hazardous	-	-	-

Based on this assistance table, we compose an abstract mode chart depicted in Figure 44 which models the system dynamic behaviour at this early stage:

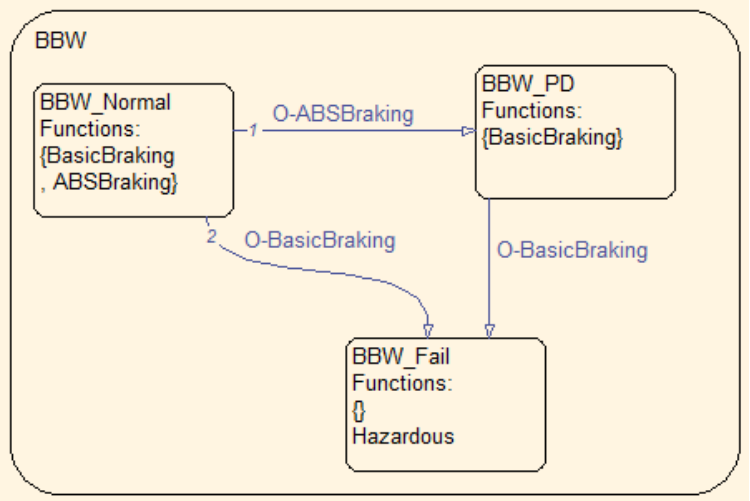


Figure 44: Mode chart for Brake-By-Wire

The level of safety assessment (requirements verification) depends on the level of detail provided in the mode chart. For this reason, it can be useful to refine the abstracted mode chart. Here, for example, to more closely reflect the inclusion of different type of pressure source, we could refine the function *BasicBraking* into *Electrical* and *Hydraulic*. This is made possible by the fact that we could utilize the current HiP-HOPS Matlab interface to set *Electrical* and *Hydraulic* blocks as 'system outport' therefore allowing fault trees and FMEA to be constructed for these functions. In the following Figure 45, additional blocks *Electrical* and *Hydraulic* are placed to illustrate this. For this reason, *Electrical* and *Hydraulic* blocks do not have failures of their own and only propagate failures. This break-down allows a more transparent functional distribution.

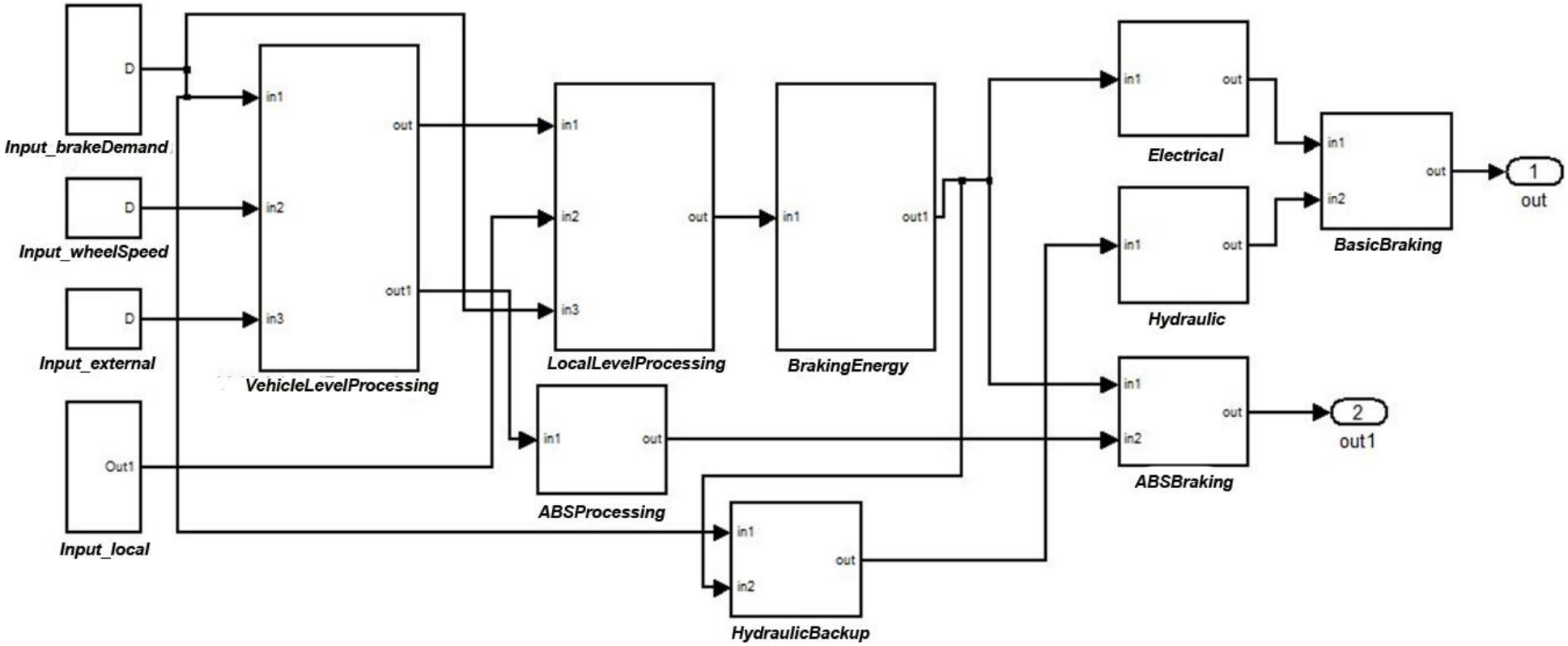


Figure 45: Brake-By-Wire revised model showing Electrical and Hydraulic sources

In response to this, the original Normal and Degraded modes are now extended to reflect the modelling of Electrical and Hydraulic modules. Subsequently, dynamic behaviour can now be modelled in the following modes:

1) *BBW_Normal* mode where both basic braking and ABS braking functions are delivered. Braking function in normal mode is delivered through the primary source, Electrical module.

2) *Permanent Degraded 1 (BBW_PD1)* mode where braking function is delivered by the Electrical module, but the ABS braking function can no longer be delivered.

3) *Permanent_Degraded2 (BBW_PD2)* mode where braking pressure is delivered by Hydraulic module, ABS function is not delivered.

4) *Fail* mode where no braking pressure is delivered. These are summarized in the updated FMEA-Mode chart assistance Table 13, and depicted in the following Figure 46 mode chart.

Table 13: Updated FMEA-Mode chart Assistance Table

Mode	Severity	Functions Delivered	Functional Failure Causing Transition	Target Mode
BBW_Normal		ABSBraking	O-ABSBraking	PD_1
		BasicBraking (Electrical)	O-Electrical	PD_2
BBW_PD1	Marginal	BasicBraking (Electrical)	O-Electrical	PD_2
BBW_PD2	Marginal	BasicBraking (Hydraulic)	O-Hydraulic	Fail
Fail	Hazardous	-	-	-

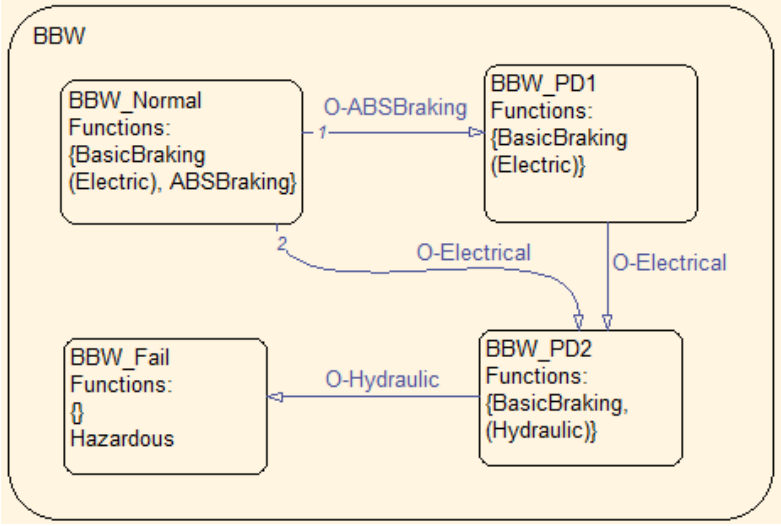


Figure 46: Updated mode chart

4.2.4 Requirement Verification

In keeping with the proposed process and to enable the verification of requirement properties, once the mode chart is constructed, it is converted into a NuSMV input model. For this high level NuSMV model, four modules are constructed to represent the system main module and each functional module (ABSBraking, Electrical, and Hydraulic). The complete NuSMV model can be found in Appendix B.1.

Among the requirement properties, safety requirements are often of primary concerns in this case study. The verification process here aims to investigate and verify that the design goals are achieved, while ensuring that the model conforms to the safety requirements. In this scenario, designers are provided with a list of 'safety requirement' (SR). These are presented and analysed throughout this section to exemplify the set of possible requirement properties. Possible general SRs which are expected to hold through the design are as follows:

SR1: *Driving assistance function(s) shall never hazardously interfere with the system state.*

SR2: *The system shall be able to withstand the occurrence of n failures, without entering a hazardous state.*

SR3: *Dormant functions shall only be activated when needed*

These requirements first have to be interpreted in terms of the behaviour specified in the mode chart model. One possible translation of SR1 for this model is that the driving assistance *ABSBraking* function, in its presence or absence, shall not cause the system to move into a hazardous mode. These can be expressed as the following SR1.1 and SR1.2:

SR1.1: "*The presence of the ABSBraking function shall not lead the system into Fail mode"*

SR1.2: "*The absence of the ABSBraking function shall not lead the system into Fail mode*"

Property SR1.1 can be interpreted as situation must not occur where the presence of driving assistance always results in the system entering Fail mode. Although relatively straightforward, this helps ensure the *ABSBraking* does not behave hazardously when selected. CTL property for this can be written as:

```
!(AG(absB.Output = 1 -> SystemMode = BBW_Fail));
```

Apart from assuring that *ABSBraking* function behaves as expected in its normal mode, SR1.2 property can be interpreted as situation must not occur where omission failure in *ABSBraking* function always results in the system entering Fail mode. The CTL property can be written as:

```
! (AG (absB.Output = 0 -> SystemMode = BBW_Fail));
```

The model checker confirms that these properties hold, and therefore we can be assured that as a non-critical function, failure in driving assistance *ABSBraking* will not dictate system failure.

Next, the SR2 requirements can be investigated. SR2 checks the robustness of the system and aims to ensure that the system can tolerate a certain number of failures. For this, SR2 can be further refined into:

SR2.1: "*If the system is in normal mode, a single functional failure shall not cause it to move directly into hazardous mode*"

This property aims to ensure that when a single functional failure occurs while the system is operating in its normal mode, the next state will be one of the degraded modes instead of the fail mode. To model this, a failure counter is introduced in the NuSMV model to record the number of functional failure occurrences. The highest possible number of the counter is three as at this stage we are keeping track of three functions (the ABS function, the Electrical function and the Hydraulic function), and failures are assumed to be permanent. This can be expressed in CTL as:

```
AG (((SystemMode = BBW_Normal) & (counter = 1)) -> AX !(SystemMode =
BBW_Fail));
```

This property is also verified to be true by the model checker.

One important thing to note is how inclusive the transition definitions are when modelling dormant functions. For example, the mode chart in Figure 46 is inclusive enough for the updated model if the hydraulic backup is a dynamic 'cold standby', where the hydraulic back up is only activated when O-Electrical is detected. However, for dynamic 'hot standby' where the hydraulic backup is continuously active, the transition definitions are no longer sufficient. This is because of the fact that if hydraulic backup is continuously active, it is possible for the system to experience a malfunction in the Hydraulic system (O-Hydraulic) when it is operating in BBW_Normal mode. If O-Hydraulic occurs in BBW_Normal, according to the mode chart in Figure 46, the system mode will stay in BBW_Normal, and when O-Electrical eventually occurs, the system will move to BBW_PD2 for one execution step before swiftly moving to BBW_Fail mode in the next step. Although it is not technically wrong, this could create a false sense of security because the system is not expected to fail by the occurrence of O-Electrical in BBW_Normal mode, especially as the mode chart aims to show a systematic degradation phase. For this reason, it can be helpful to take into consideration Hydraulic functional failures (if it is activated) in modes where Hydraulic output is not expected (in this case BBW_Normal and BBW_PD1).

One possible way to better address this is by introducing an additional temporary mode (BBW_TD1), to model the failures in the Hydraulic function when basic braking is provided correctly through Electrical system. This degraded BBW_TD1 mode could serves as a potential warning that the backup function has failed before the primary

function, a state in which potential recovery steps can also be included and performed. This can be illustrated in the mode chart in Figure 47 below.

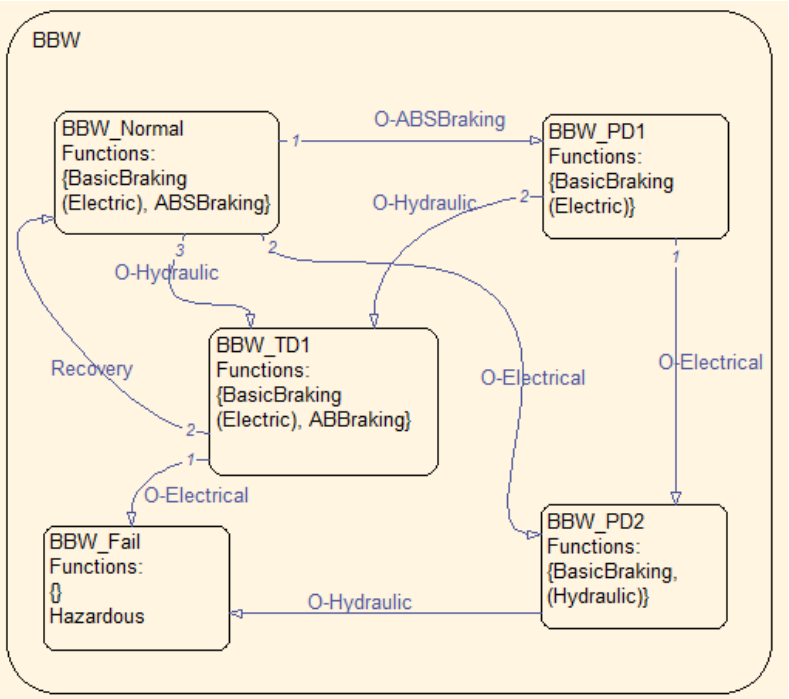


Figure 47: Modified mode chart for Brake-By-Wire

SR3, however, works with the assumption that the Hydraulic backup function is activated only when Electrical module does not supply any output presssure (Figure 46). It aims to ensure only either one or another is activated at the same time. This interpreation and its CTL specification can be expressed as follow:

SR3.1: *"Both Hydraulic and Electrical power shall not be activated at the same time"*

```
AG !((Hydraulic.state = ON) & (Electric.state = ON));
```

4.2.5 Refinement of Transition Events

As explained in the previous chapter, the level of detail in the verification is also dependent on the level of detail in the model itself. This phase of the process allows the brake-by-wire abstract mode chart (Figure 47) to be refined. The refinement of transition labels of this mode chart can be derived from either the FMEA directly or hierarchically through the model failure behaviour described in the failure annotations.

4.2.5.1 *Refinement of Transition Events through Minimal Cut Sets*

The first possible way to refine this mode chart is by replacing the transition event expression with its causing events, where the causing events can be effectively obtained and mapped from the HiP-HOPS FTA results. For each top event, its minimal cut sets can essentially be used to form the replacement expressions. In this case, Figure 48 presents the fault tree for the condition failure "O-Hydraulic". Figure 49 presents the accompanying mode chart incorporating the corresponding root causes as transition events. Compared to mode chart in Figure 47, the examination of this expanded mode chart allows analysts to establish direct links between internal module malfunctions and the effects of their occurrence on the system mode transitions. The expanded mode chart is considerably more informative and allows more verification properties to be checked (i.e. checking whether certain malfunctions or their combinations would lead to changes in system functionality modes). For example, instead of only being able to check the effects of O-ABSBraking, O-Hydraulic, and O-Electrical, this expanded mode chart allows us to ensure that malfunction events LSBEa and LSBEb will not always cause a transition to a hazardous state: "!(AG(LSBEa & LSBEb) -> (States = BBW_Fail));"

One of the main advantages of composing the transition events directly from their root causes is the fact that the mode chart and NuSMV models can be build without the need to model every level of the component or module behaviour. This is useful for effective iteration of abstract verification before details of module behaviours become available. Once details for each module are available and more dynamic behaviours are to be modelled (for example, to include non-failure related transitions), the mode chart can be refined as described in section 4.3.2.

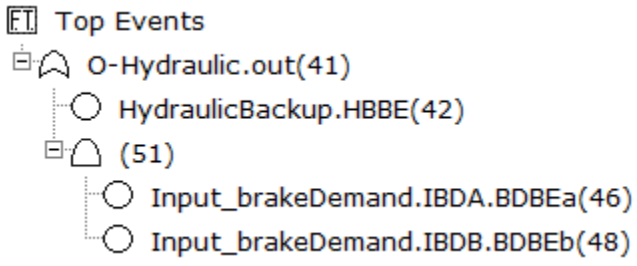


Figure 48: Fault tree for Omission of Hydraulic Failure

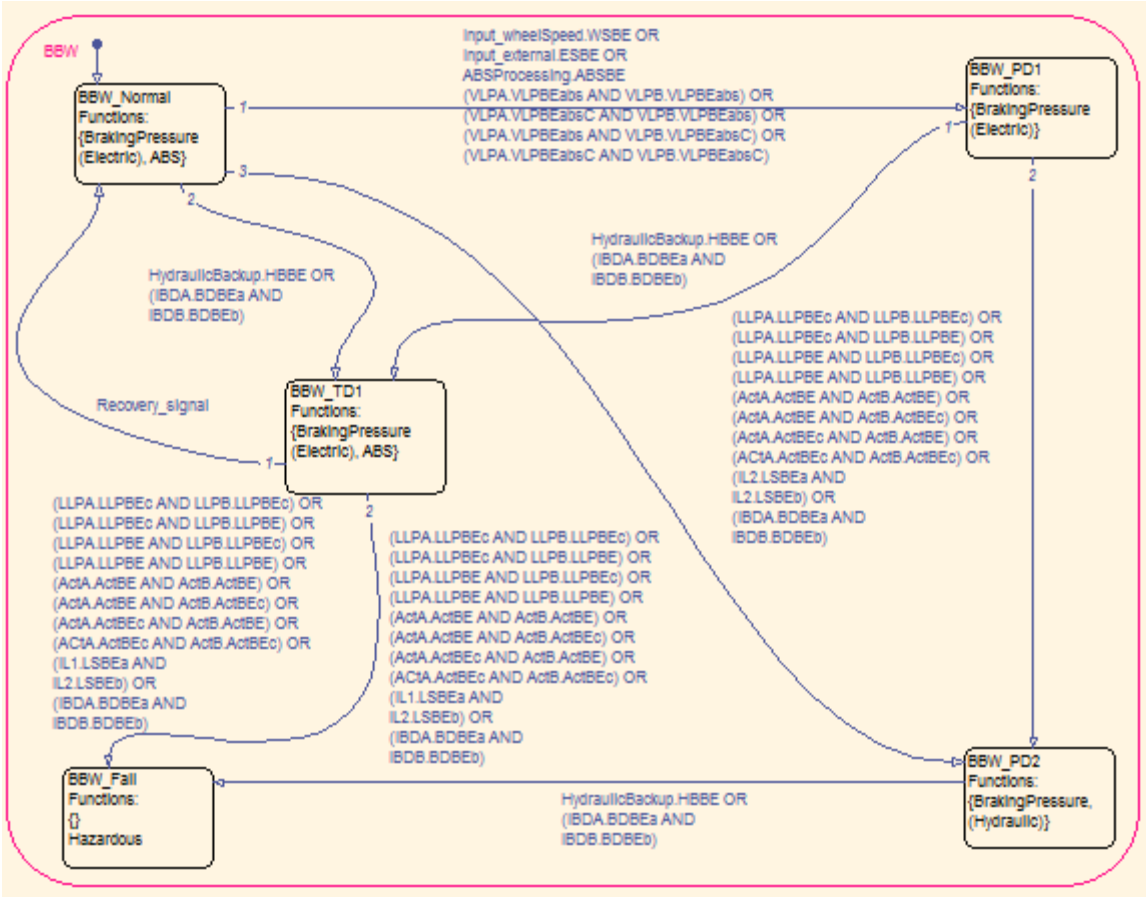


Figure 49: Expanded mode chart with minimal cut sets mapped to transition events

4.2.5.2 *Refinement of Transition Events through Model Failure Annotation*

As discussed in the previous chapter, it is also possible to construct the mode chart which captures and reflects the functional hierarchy by constructing independent mode chart and SMV modules for each function. To effectively link failure behaviour to input modules and capture the structural topology, transition events retain a similar structure to the ones of HiP-HOPS failure annotation. This means they are expressed only in terms of input functions and internal malfunction events. To illustrate this, Figure 50 presents the structural model of *BrakingEnergy (ACT)* module which consists of primary and backup modules *ACT A* and *ACT B*. *ACT* receives its input from *localLevelProcessings* (*LLP)*, and outputs the results of the process through *ACT O*. *ACT O* serves as the output module and only propagates failures.

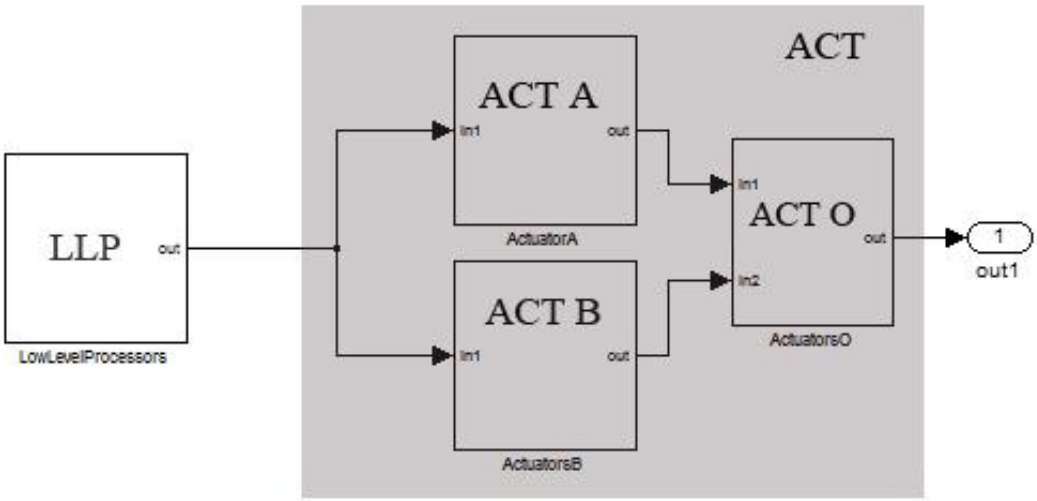


Figure 50: Structural model of Braking Energy

As with all other HiP-HOPS models, it is annotated with failure information which describes its failure behaviour. The failure annotation for *ACT A* (which is identical *ACT B)* for describes the causes of omission failure O-Out as:

O-Out **=** *ActBE OR ActBEc OR O-in1*

This failure behaviour is identical for *ACT B*. This can be mapped into the mode chart and subsequently the NuSMV model. The following Figure.51 illustrates how the

failure annotation can be represented as the mode chart transition. As *ACT A* receives its input from *LLP* which is in the same hierarchical level as *ACT A*, O-LLP can be used to directly replace O-in1 in the mode chart. *ACT A* can hold two failure-relevant modes: ActA_Normal (when it delivers its output) and ActA_Fail (when O-Out occurs and it fails to deliver its output). Once *ACT A* enters its ActA_Fail mode, it sends the appropriate global broadcast signal ("/O-ActA*")* to announce the occurrence of O-Out in *ACT A*.

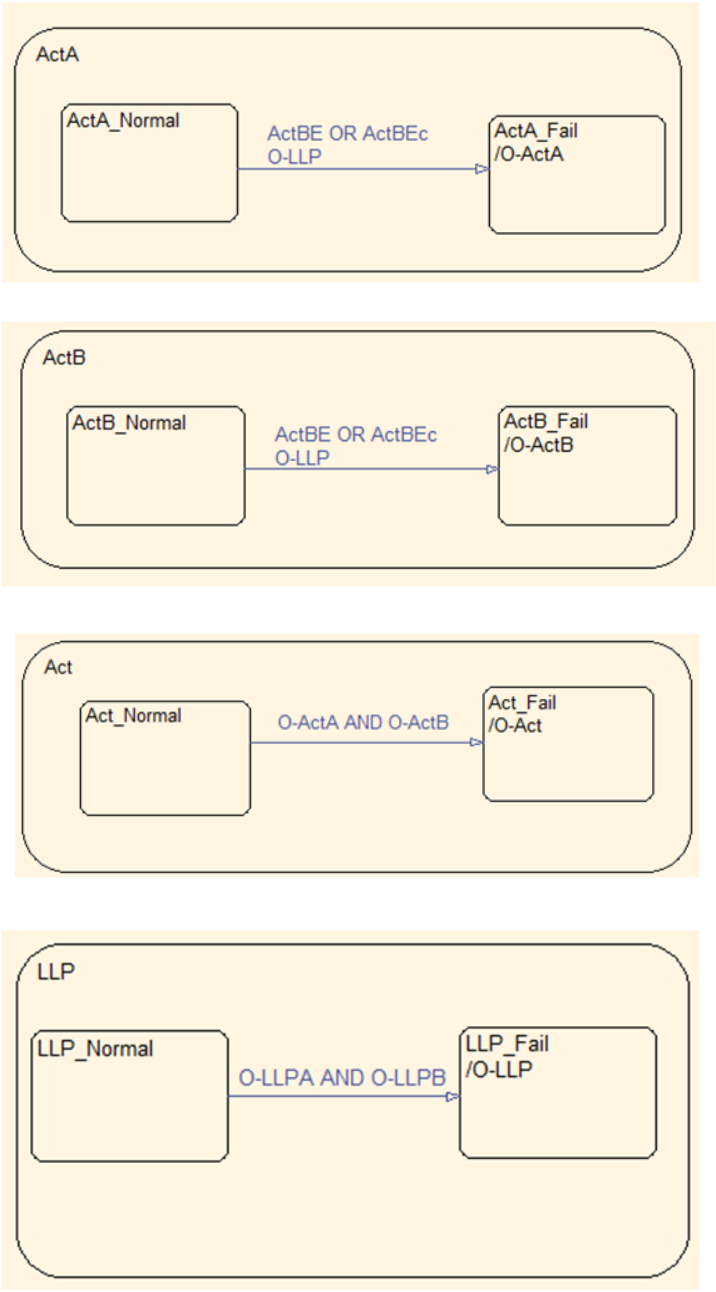


Figure.51: Mode chart for failure behaviour in *ACTUATOR*

The translation of this failure behaviour in HiP-HOPS to a NuSMV model can also be done in a structured way. Module *ACT A* input variables are used to represent input deviations, while its internal malfunctions are defined locally. This way, failure expression can be directly represented in the local definition of O-Out which is then used to affect the outcome of Out. Module *ACT B* can be constructed in a similar way. And as *ACT A* and *ACT B* are subsystems of *ACT*, output deviations O-ActA and O-ActB are passed as input deviation variables for module *ACT*. *Main module* manages the global architecture of the network and the broadcasting of events which enable transitions between modules. These are illustrated in the excerpt of a NuSMV model for an actuator module presented in the Figure 52:

ACT A

```
MODULE ACTA (O-in1)
VAR
O-Out: boolean;
Out: boolean;
ActBE: boolean;
ActBEc: boolean;

ASSIGN
init(ActABE) := 0;
O-Out := O-in1 | ActBE | ActBEc;
Out := !O-Out;

next(ActABE):=case
ActABE = 1 : 1;
1: {1,0};
esac;

next(ActABE):=case
ActABE = 1 : 1;
1: {1,0};
esac;
```

ACT B

```
MODULE ACTB (O-in1)
VAR
O-Out: boolean;
Out: boolean;
ActBE: boolean;
ActBEc: boolean;

ASSIGN
init(ActABE) := 0;
O-Out := O-in1 | ActBE | ActBEc;
Out := !O-Out;

next(ActABE):=case
ActABE = 1 : 1;
1: {1,0};
esac;

next(ActABE):=case
ActABE = 1 : 1;
1: {1,0};
esac;
```

ACT

```
MODULE ACT (O-ActAOut, O-ActBOut)
VAR
O-Out: boolean;
Out: boolean;

ASSIGN

O-Out := O-ActAOut & O-ActBOut;
Out := !O-Out;
```

LLP

```
MODULE LLP( LLP_inputDeviation_variables...)

VAR
Out: boolean;
O-Out : boolean;
C-Out : boolean;
LLPBE : boolean;
LLPBEc: boolean;

ASSIGN
init(LLPBE) := 0;
O-Out := LLPBE;
Out := !O-Out;
C-Out := LLPBEc;

next(LLPBE) := case
LLPBE = 1: 1;
1: {1,0};
esac;
```

MAIN

```
MODULE main

VAR

llp: LLP;
acta : ACTA(LLP.O-Out);
actb : ACTB(LLP.O-Out);
act: Actuator(acta.O-Out, actb.O-Out);

Other_local_variables ...
Other_definitions...
...
```

Figure 52: Excerpt of the NuSMV model for the Braking Energy

This refinement of the NuSMV model captures and retains the hierarchical composition of the model and allows more detailed verification to be performed. By examining the relationships between the dynamic behaviour of modules it is now possible to verify more safety related requirements, from more straight-forward ones like "*As long as Braking Energy ACT A is functioning, the Braking Energy function shall be present*", or for a cold-standby system which examines the electrical and hydraulic modules: "*Only either Electrical pressure or Hydraulic pressure shall be supplied at one time*", to the

effects of this function behaviour on the system modes: " *System shall not be allowed to enter hazardous mode when Electrical system is functioning*".

Although the processes of construction and refinement of state machines are currently manual, the potential for future automation has been outlined in Chapter 3.8. With IACoB, the construction of these state machines (presented in Figure 44, Figure 46, Figure 47, Figure 49, Figure.51, and NuSMV excerpt in Figure 52) are no longer *ad hoc*, but made systematic with the help of FTA/FMEA results. The ability to verify listed safety requirements (SR 1.1 to SR 3.1) also highlights the benefits of the application of BSA at this early stage.

4.3 Architecture-allocated Functional Model

To illustrate the iterative application of the IACoB process in a more detailed design, we present another phase of analysis in an architecture-allocated functional model of the BBW system. The architecture-allocated functional model extends the purely functional model by taking into account early system architecture and concisely represents allocation of functions to architectural elements without going into fine details of the architecture. Figure 53 illustrates the architecture-allocated model of the system for the corresponding four wheels of the vehicle. This model is developed based upon the earlier functional model (Figure 45) and allocates functions to components. *VehicleLevelProcessing* function is assigned to (and therefore from now onwards referred to as) an *ECU* (electronic control unit). Similarly, each *LocalLevelProcessings* function is allocated to a *BCU* (Brake Control Unit) which, together with an actuator, are assigned for each wheel. It is common that multiple architectural components are assigned to perform a single function, or for a single components to be shared between multiple functions. Here *ABSBraking* function is realized by sharing BCU and actuators. ABS command is fed directly from ABS processing components to the *BCUs* to reflect the correct value of braking pressure applied by each actuator.

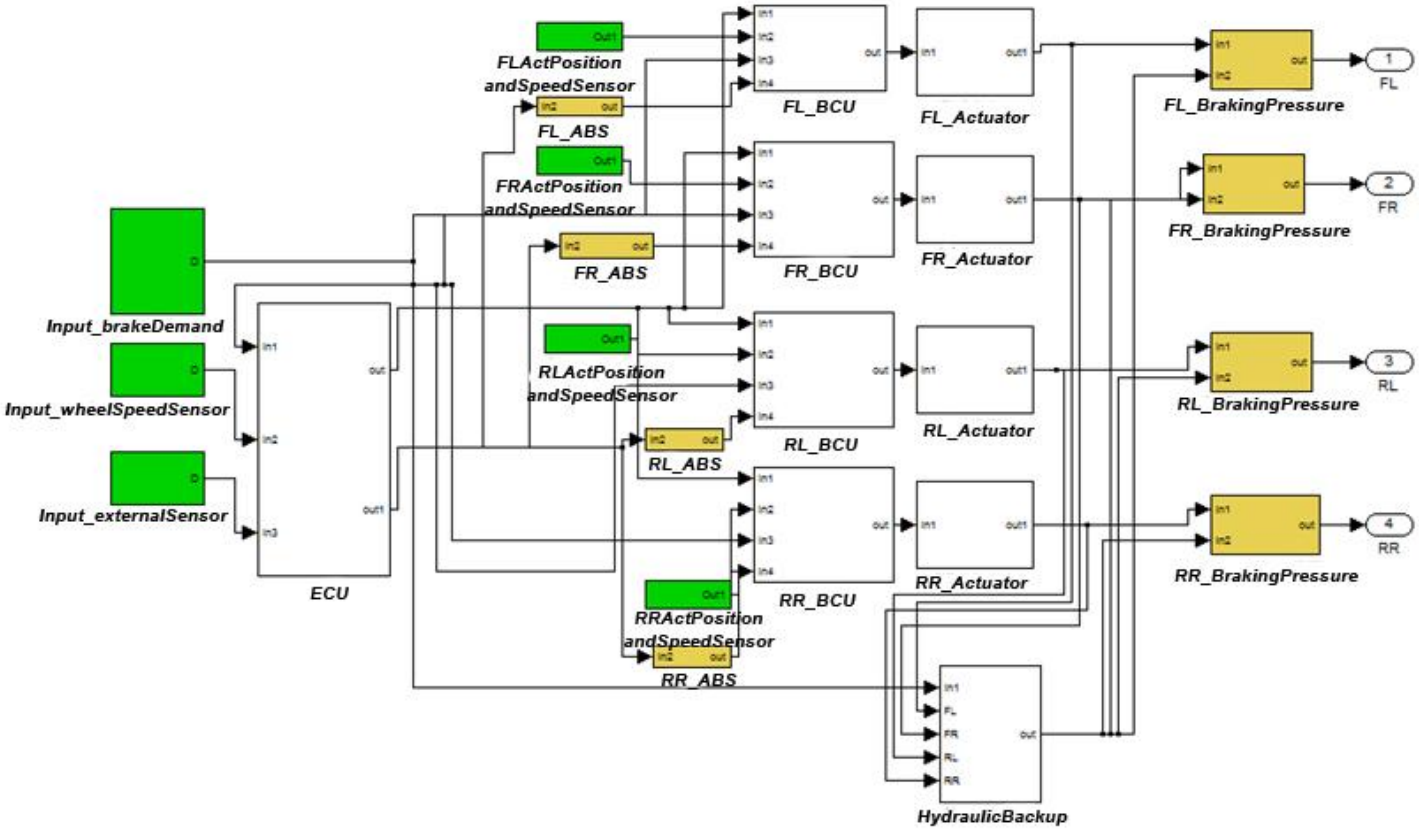


Figure 53: Architecture-allocated functional model for brake by wire system

In accordance with our proposed process, the iterative analysis begins again from the construction and analysis of an FFA, followed by an FTA and FMEA. At this phase of analysis, we place the focus on the *BCU* and the delivery of each wheel braking pressure as part of the whole brake-by-wire structure, and explore the relationship between the delivery (and absence) of this function from different wheels, as opposed to the independent analysis performed previously in section 4.2.

The system delivers four braking functions, each handled by a BCU delivering commands to actuator for each wheel. The longitudinal symmetry of this functional design means that the potential single functional failures on each side of the car and their effects of the systems are similar to those on the other side of the car. For this analysis, single functional failure and multiple combinatorial failures will be investigated.

4.3.1 Analysis of Single functional failure

The first part of the FFA identifies potential single functional failures of the wheel braking function. Here we introduce a new type of failure, *LockedWheel*, and investigate the effects of this failure on the system. The *LockedWheel* failure occurs when a wheel experiences rapid deceleration (causing it to 'lock') and stop much more quickly than the vehicle could. This is usually prevented by the *ABS* anti-lock function which alternately reduces the pressure to the brake until it sees acceleration, and increases pressure until it sees deceleration again. This is performed within a very short period of time, resulting in the slowing down of the wheel matching deceleration rate of the vehicle.

In this example, the supervision of relevant parameters (e.g. wheel speed reading) and the processing of ABS commands are shared between *ECU* and *ABS* processing component. To maintain simplicity, we assume that the *ABS* processing component only propagates failure, and the failure in ECU is enough to cause failure in producing correct ABS command. The *LockedWheel* (*L-BrakingPressure*) failure occurs when the new internal malfunction *LockBE* occurs in the *BCU* and at the same time the *ECU* fails to produce necessary command/information to instruct the *ABS* to prevent locking (Omission of ABS command or O-ABScmd). *LockBE* could represent an internal

failure in the BCU algorithm causing maximum brake pressure to be applied. This can be expressed in the following failure logic for the BCU:

$$L\text{-}BrakingPressure = O\text{-}ABScmd \ AND \ LockBE$$

Figure 54 presents a fragment of the architecture-allocated functional model to illustrate how failure can be propagated through the topology, eventually causing front-left (FL) wheel to lock:

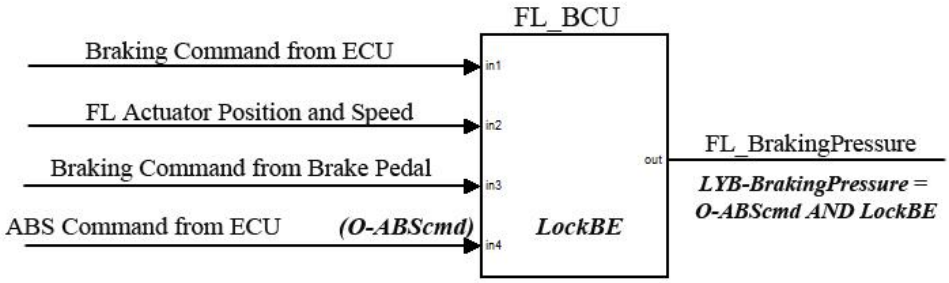


Figure 54: Failure propagation to BCU

Table 14 presents the FFA for single wheel braking function for front left wheel. Omission and commission failures were as addressed and treated in previous chapter (section 4.2), and the effects of wheel locking with or without braking intention are examined:

Table 14: Functional failures for single wheel braking function

Function	Failure Type	Effects on System	Severity	Detection	Recovery	Recommendation
FL_Braking Pressure	Locking-Com. Permanent wheel lock when there is no braking intention	Vehicle tends to drift to side. Severe lost of control as maximum brake is applied	Critical	Comparison of pedal input and pressure sensor feedback	-	Assume commission of brake pressure is transformed to omission of brake pressure.
FL_Braking Pressure	Locking-Om. Permanent wheel lock when there is braking intention	Vehicle tends to drift to side. Severe lost of control as maximum brake is applied	Critical	Comparison of pedal input and pressure sensor feedback	-	ABS algorithm to prevent permanent locking. *Additionally, intentional locking of diagonal wheel

4.3.2 Analysis of multiple functional failures

Apart from single functional failure analysis, the effects of combinations of multiple functional failures in the vehicle wheels can also be examined. The analysis involves conjunctions of two to four functional failures, and combinations of failures that require further examination are identified. As the system incorporates four braking functions (one for each wheel) and there are six corresponding failure modes for each function, there appears to be a large number of possible combinations. However, a systematic analysis of unique combinations yields a relatively small number. The reason is that due to the symmetry of the brake-by-wire system, only certain combinations are unique. Certain failure combinations are also inapplicable because they can only occur in mutually exclusive modes, for example, braking and absence of braking. Here, analysis of the *L-BrakingPressure* failure is performed in a scenario where the locking occurs

when brake pressure is required. The full analysis of the FFA is too long to include and will not contribute much to this discussion. We focus on the analysis regarding wheel locking which shows that:

- Severity of single wheel locking failure is critical and affects the stability and steeribility of the vehicle.
- Severity of two locking failures in diagonal wheels is marginal because stability is improved.
- Severity of three locking failures is critical.
- Locking in all four wheels is identified as less severe than locking in three, or in some cases, two wheels.

From these FFA results, we are able to identify recovery mechanisms against such types of failures by incorporating the ability to perform intentional locking: the intentional locking of a diagonal wheel can be performed in response to a single wheel locking failure, and intentional locking of all four wheels can be used as recovery mechanism to reduce the severity of a failure of three wheels.

Working with the assumption that the recovery plan to unlock the wheel (e.g. by releasing pressure in time) is not possible, it is decided that the ability to intentionally lock the wheel can be incorporated as an additional function to each *BCU*. This new function enables intentional locking by applying maximum braking pressure to the wheel. An additional module (*DL*) is used to monitor the output of each *BCU* to detect locking failure (*L-BrakingPressure*) and subsequently activates the locking of corresponding diagonal wheel. This *DL* module can be implemented as part of the *ECU* or as a separate independent module. It is also possible to further analyse the failure to provide this intentional locking (omission and commission failure of *DL*). To maintain the simplicity of this example however, we assume *DL* only propagates failures and focus the analysis on the degradation phases the system experiences in the occurrence of wheel locking, and ensuring required safety properties hold during these phases. The revised functional model of the *BCU* can be seen illustrated in the figure below. FL indicates Front-Left wheel, FR indicates Front-Right wheel, RL indicates Rear-Left wheel, and RR indicates Rear-Right wheel.

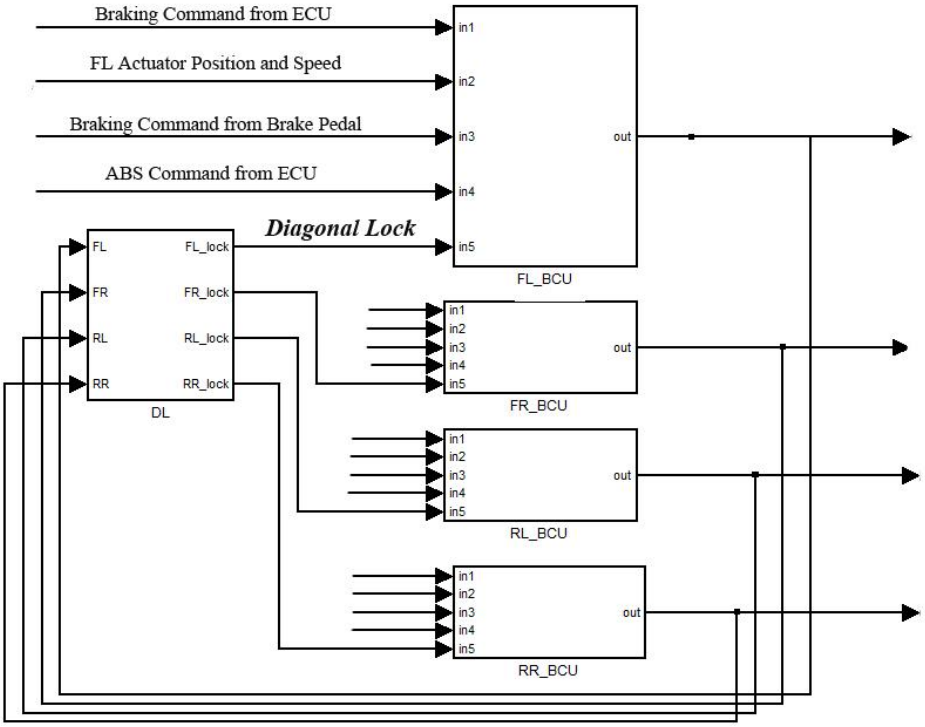
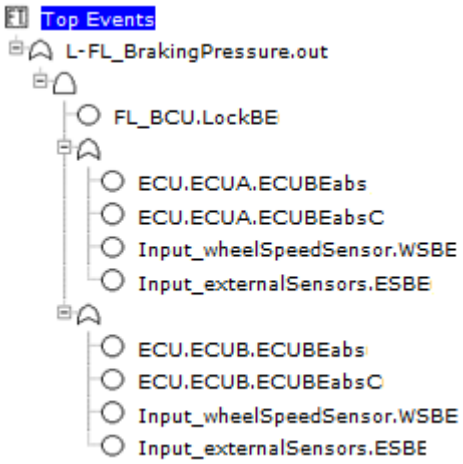


Figure 55: Updated BCU for wheels with Intentional Diagonal Locking (*DL*)

Following the introduction of the new function to enable intentional diagonal locking, we aim to analyze and verify that the system holds true the key safety requirements in its degraded mode. And as with the earlier example, FTA and FMEA results are used to assist in the construction of a mode chart where this dynamic behaviour can be analyzed. FTA/FMEA results are used to derive root causes of locking *L-BrakingPressure* for each wheel. This in turn enables us to study the how failures from different wheel propagates to cause the locking of the diagonal wheel, and if safety requirements still hold.

Figure 56 presents the fault tree for *L-BrakingPressure* for FL wheel and the list minimal cut sets derived:



Minimal Cut Sets For *L-FL_BrakingPressure*:

- FL_BCU.LockBE AND Input_wheelSpeedSensor.WSBE
- FL_BCU.LockBE AND Input_externalSensors.ESBE
- ECU.ECUA.ECUBEabsC AND ECU.ECUB.ECUBEabsC AND FL_BCU.LockBE
- ECU.ECUA.ECUBEabsC AND ECU.ECUB.ECUBEabs AND FL_BCU.LockBE
- ECU.ECUA.ECUBEabs AND ECU.ECUB.ECUBEabsC AND FL_BCU.LockBE
- ECU.ECUA.ECUBEabs AND ECU.ECUB.ECUBEabs AND FL_BCU.LockBE

Figure 56: Fault tree for L-FL_BrakingPressure

To understand how these root causes affect the changes in system modes and the activation of newly introduced intentional diagonal locking, we examine the dynamic behaviour of the *DL* module. In its normal mode, the *DL* module's function is to monitor for the occurrence of locking in any wheel and (when detected) instantiate the locking of the diagonal wheel. As mentioned earlier, the *DL* only propagates failures and therefore would only respond to external failures. So instead of modelling how it degrades in response to the failure of delivery of its monitoring and activating function, the modes are decided based upon the condition of locking of each wheel. The following modes are possible:

1) *Normal:* when there is no wheel locking occurring
2) *TDn_Critical_ X* : Temporary degraded mode when locking occurs in wheel(s) *X* with total *n* number of locking occur in the vehicle.
3) *PDn_ X*: Permanent degraded mode when locking occurs in wheel(s) *X* with total *n* number of locking occur in the vehicle

X here represents vehicle wheel(s): FL, FR, RL, RR. $X \subseteq \{FL, FR, RL, RR\}$. The states are mainly categorized based upon the *n* number of wheels locked (intentionally or not). Temporary degraded (*TD*) modes are marked as critical because they are only assigned to occurrence where either one or three wheels are locked, the occurrence of which has critical effects. These modes are temporary because the entry behaviour (which is executed immediately once the mode is entered) triggers event "/*X DiagonalLock"* which locks the corresponding diagonal wheel *X*, and therefore causes the system to move to a non-critical permanent degraded mode. Permanent degraded (*PD*) modes are not critical as they occur when two diagonal wheels or all four wheels are locked.

Figure 57 below describes this relationship and the transitions between modes. Here assumption is made that *DL* is designed in such way that it processes one locked wheel signal at a time. In a real-life scenario, it is possible for locking of multiple wheels to occur within a time period so close to one another it appears to be occurring simultaneously. To handle this, the *DL* is assumed to be able to register the time difference and sequence of occurrence. This allows appropriate action to be taken (i.e. intentionally activating the locking of diagonal wheel if necessary) before processing the next locked wheel(s).

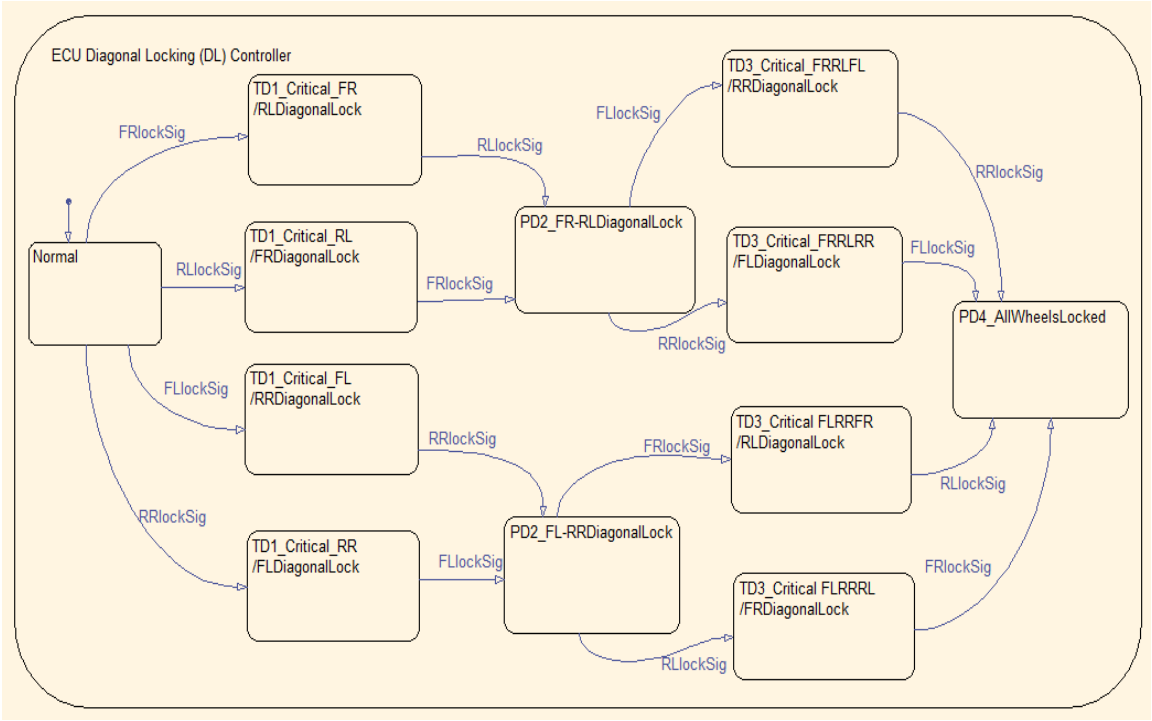
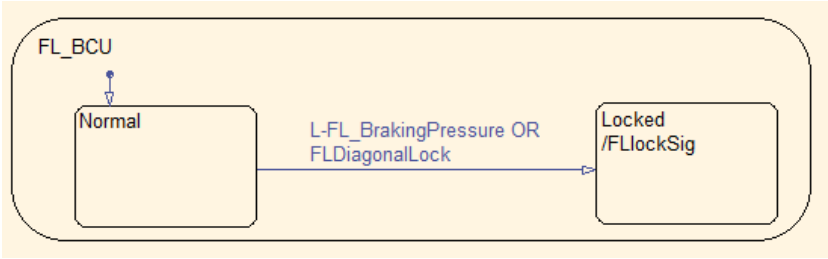


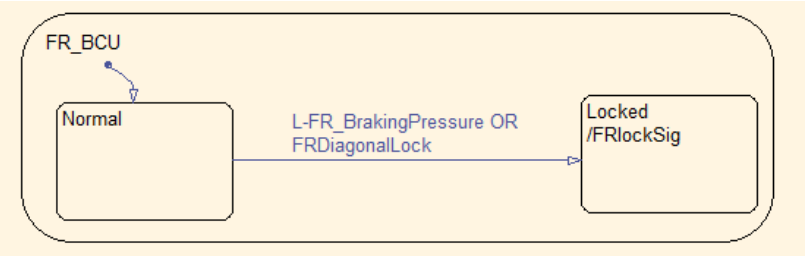
Figure 57: Mode chart for DL Controller

The events that trigger the mode transitions are signals from individual *BCUs* to indicate wheel locking. This locking can be caused by an intentional locking command from *DL* or unintentionally as a result of *L-BrakingPressure*. Figure 58 presents the mode chart for the wheel *BCUs*.

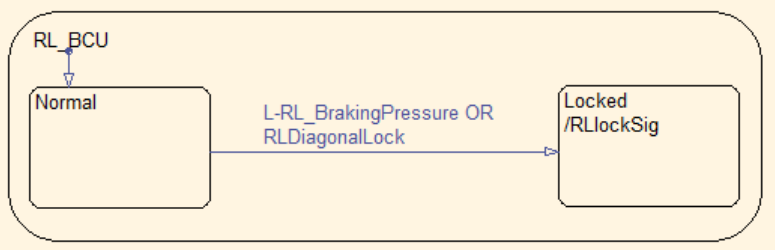
Front Left (FL) BCU



Front Right (FR) BCU



Rear Left (RL) BCU



Rear Right (RR) BCU

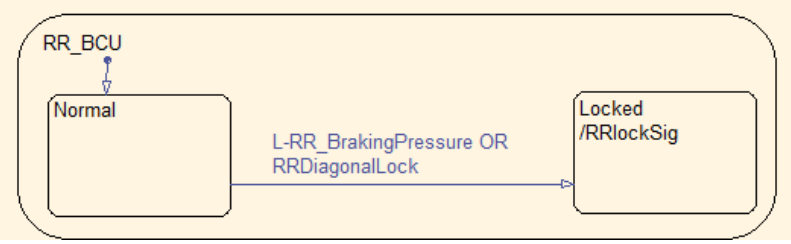


Figure 58: Mode chart for Wheel BCU

Similar to the previous process at this stage in section 4.2, we are able to extend the mode charts here by mapping the failure "*L-X_BrakingPressure*" to its minimal cut sets identified through FTA/FMEA (Figure 59 for FL wheel):

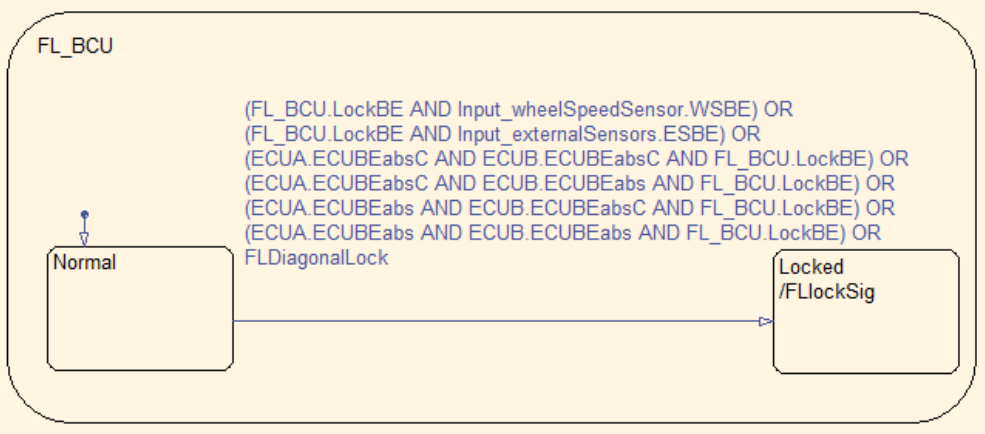
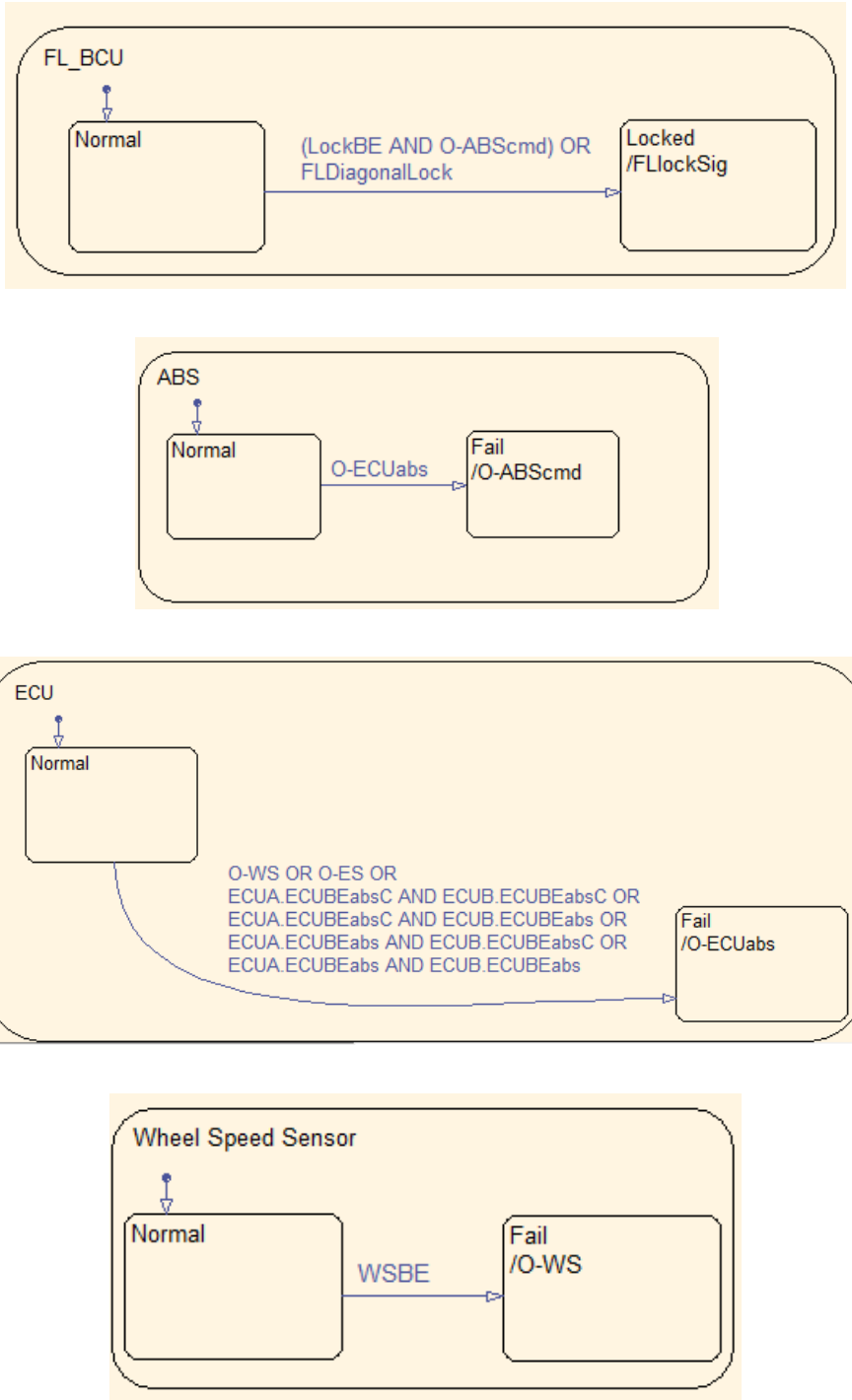


Figure 59: Expanded transition based on Minimal Cut Sets

At this stage we could also construct the mode chart to reflect the hierarchical structure, and enable generation of a NuSMV model which captures all the relevant modules that trigger corresponding transition events in *BCU*. The figure below depicts the mode charts for the *FL_BCU, ECU*, and two input sensors relating to the failure O-ABScmd*:*



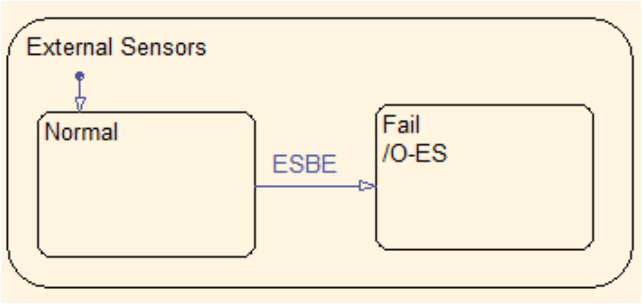


Figure 60: Mode chart for modules relating to Locking of Front Left (FL) wheel

The corresponding NuSMV model for the FL_BCU is presented below. As the FL_BCU receives its input from the ABS module (please see Figure 53), the propagated input deviation O-ABScmd gets passed as its input variable. It also receives the command to intentionally lock its wheel from the *DL*, *FLDiagonalLock* (Figure 55), which is also passed as an input variable. Its internal malfunction *LockBE* is also included as part a local variable. These allow the forming of its output deviation failure expression:

LockBE AND O-ABScmd OR FLDiagonalLock

The complete SMV model can be found in Appendix B.2.

```
MODULE FL(O-ABScmd, FLDiagonalLock)
VAR
States : {Normal, Locked} ;
LockBE : boolean;
counter : 0..1;
FLlockSig : boolean;
locked: boolean;

ASSIGN
init(States) := Normal;
init(LockBE) := 0;
init(counter) := 0;

locked := (LockBE & O-ABScmd)| FLDiagonalLock;
FLlockSig := case
States = Normal : 0;
1: 1;
esac;

next(States):=case
States = Normal & locked = 1 : Locked;
1: States;
esac;

next(LockBE) := case
LockBE = 1 : 1;
1: {0,1} ;
```

```
esac;

next(counter):=case
States = Locked : 1;
1: counter;
esac;
```

This case, again, demonstrates how CSA assists the identification of root causes (of locking failure in a wheel), and its role in the construction of the system state machines. The modelling of the intentional locking (nominal behaviour) of diagonal wheels by *DL Controller* in response to the locking failure of other wheels highlights the role of BSA. By enabling the modelling of these different aspects of the system, IACoB allows a better understanding of the system dynamic behaviour and enables verification of safety properties in this context.

For example, with this introduction of an intentional locking function, it is important to ensure that the system still adheres to the list of safety requirements (SR1 – SR3) defined in section 4.2.4. This modelling of modules and *DL* controller in behaviour in NuSMV subsequently allows verification of these safety properties for the extended model. Possible scenarios for safety and reachability of the control for intentional locking function can be examined. First we investigate SR1: "*Driving assistance function(s) must never hazardously interfere with basic critical function(s)".* This is adapted into SR1.3 to reflect the fact that the driving assistance function (anti-lock) to be investigated here refers to intentional diagonal locking, and in addition to investigating whether it affects the braking pressure, we aim to ensure that the activation of diagonal locking will not cause unintentional locking which lead to hazardous states.

SR1.3: "*Intentional locking of diagonal wheel should not lead to hazardous state"*

To do this, "hazardous state" is defined as the condition either where one wheel locks or where three wheels lock, i.e. the occurrence of either TD1_Critical_ *X* or TD3_Critical_ *X* respectively:

```
Hazardous := case
States = TD1_Critical_FR |States = TD1_Critical_RL | States =
TD1_Critical_FL | States = TD1_Critical_RR |States =
TD3_Critical_FRRLRR |States =  TD3_Critical_FLRRFR | States =
TD3_Critical_FLRRRL : 1;
1: 0;
esac;
```

The activation of *DL* (*DLActive*) is defined as the condition when any of the wheels has been diagonally locked intentionally:

```
DLActive := FLdiagonalLock | RLdiagonalLock | FRdiagonalLock | RRdiagonalLock;
```

We aim to verify that the diagonal locking function itself will not lead to or always be the cause of the system entering this hazardous state. This can be expressed in CTL as:

```
SPEC !AG(DLActive -> Hazardous);
```

As this property is verified to be true by the model checker, the next SR2 - "*The system shall be able to withstand the occurrence of n failures, without entering a hazardous state. "* - can be investigated. Instead of counting the number of failures, the *Counter* variable is assigned to keep record of the number of locked wheels (whether intentional or caused by a failure). For this scenario, the initial aim here is to ensure that when the number of locked wheels is not one or three, the system should not reach the Hazardous state. This can be specified in the following SR2.2 and the accompanying CTL expressions:

SR2.2: *"In situations where the number of wheels locked is not one or three, the system shall not enter the hazardous mode"*

```
SPEC AG((!(Counter = 1) & !(Counter = 3))->!Hazardous));
```

This property does not hold and the model checking traces demonstrate that the locking of two non-diagonal wheels at one point leads to locking of three wheels, which is Hazardous. With the current arrangement of *DL*, however, this means that the locking of two non-diagonal wheels should always eventually lead to the locking of all four wheels. Variable *TwoParallelWheelsLocked* is assigned to represent the locking of two non-diagonal wheels. This state of reachability can be verified through the following modified properties:

SR2.2: *"In situation where two non-diagonal (parallel) wheels are locked, all four wheels shall eventually be locked"*

```
TwoParallelWheelsLocked =(flw.States = Locked & rlw.States = Locked)|(frw.States = Locked & rrw.States = Locked) | (flw.States = Locked & frw.States = Locked) | (rlw.States = Locked & rrw.States = Locked)
```

```
AG(TwoParallelWheelsLocked -> AF(States =  PD4_AllWheelsLocked));
```

This property is verified to be true by the model checker, and therefore we are assured that non-diagonal locking of two wheels will also lead to locking of all four wheels (non-hazardous state).

Next we continue to check the requirement SR3: "*Dormant functions shall only be activated when needed*". In this scenario, intentional diagonal locking should only be instantiated when the *ABS* function is not working, as in its presence, the *ABS* would be expected to manage the prevention of wheel locking:

SR3.2: "*Intentional locking of diagonal wheel shall not be instantiated when ABS function is working*"

For this, we need to again define the condition *ALLOFF* where no diagonal locking is taking place. For every situation where the *ABS* is working (therefore omission O-ABScmd = 0), ALLOFF should be true. This can be expressed in CTL as:

```
SPEC AG((O-ABScmd = 0)-> ALLOFF) ;
```

This is also verified to be true by the model checker.

Additional properties to check system robustness and failure recoverability could include the verification of whether intentional diagonal locking will always eventually result in the system moving from the hazardous state to a non-hazardous state. This aims to ensure that the *DL* fulfils its function as a fail-safe mechanism. Non-hazardous states refer to the condition where either only two diagonal wheels are locked or all four wheels are locked, which brings us to the next requirement to verify, SR4:

SR 4: *"Intentional locking of wheels shall always eventually lead the system to non-hazardous states*"

This can be modelled in CTL as:

```
SPEC AF (DLActive ->! Hazardous);
```

This property aims to ensure that DLActive is performing its task to ensure system moves from a hazardous state to a non-hazardous state. This is also verified to be true by the model checker.

4.4 Chapter Summary

This chapter explored the application of the IACoB process in a vehicle brake-by-wire system. It investigated how the approach utilizes CSA and BSA to help perform safety assessment and influence system design in the early phase of the system development.

Two main models were presented to highlight different discussion elements. The first model described high level system functional design where FFA and FTA/FMEA (CSA) were used to effectively identify root causes of hazardous functional failures (i.e. absence of braking pressure). Appropriate design modification and improvements, including introduction of backup mechanisms for critical functions, were then made to reduce or avert risk of failure. This was followed by formal verification (BSA) via the NuSMV model checker to verify that the design adheres to safety requirement specifications. The second model provided more details about allocation of functions to architectural elements. It explored a further particular failure (i.e. wheel locking) and subsequently recommended an additional new function (diagonal locking) to help the system respond to this failure. The integration of this function into the design and whether or not the predefined safety requirements specifications still hold were then analyzed.

The proposed approach provides assistance in evaluating the design and allows both CSA and BSA to exploits analysis results from previous stages to help with the safety assessment. In particular, generation of the mode chart in order to enable BSA utilizes results from FTA/FMEA in the composition of its event transitions. The process also allows verification to be performed early on an abstract mode chart before more concrete details are available.

CHAPTER 5. Case Study on Aircraft Wheel Brake System

5.1 Introduction to Aircraft Wheel Brake System

This second case study aims to explore further the role of BSA in influencing the system design. It investigates the application of IACoB process to an aircraft wheel brake system. The model presented here is mainly an adaptation from the (ARP 4761) aircraft wheel brake system, which is also referenced in (Joshi *et al.*, 2006).

The main function of the wheel brake system is to provide safe braking function for aircraft during the taxiing and landing. This mainly involves supplying correct pressure and preventing skidding. Secondary functions of the wheel brake system also include preventing unintended aircraft motion when parked, and stopping main gear wheel rotation upon gear retraction.

The braking system consists of two primary hydraulic pumps: *GreenPump* and *BluePump*. On Normal braking mode, *GreenPump* provides the required hydraulic pressure and the Alternate mode, which is powered by *BluePump*, is held on standby. When failure occurs on normal system, the brake is driven by hydraulic power generated by *BluePump*.

In the original (ARP 4761) example, another backup mechanism was in place lest both of the pumps fail. Here, however it has been deliberately excluded in the beginning of this discussion to demonstrate how our process arrives to the conclusions for the need of the safety measures. Therefore in the initial system model of this example, it is assumed that one backup hydraulic pump (*Blue Pump*) is sufficient.

In normal mode, *BSCU* (Brake System Control Unit) receives brake pedal positions as input and processes this information to produce control signals to the brakes. *BSCU* also monitors various input signals that indicate certain critical aircraft and system states to provide correct brake functions and improve fault tolerance mechanism, generate warnings, indications, and maintenance information to other systems.

5.1.1 Nominal system model

The brake system used in this study is a modified version of the one used in ARP4761. Its architecture is illustrated in Figure 61.

The system consists of the following main components: *BSCU* (Brake System Control Unit), two hydraulic pressure lines, mechanical components, and an output component. *BSCU* (Brake System Control Unit) is the digital controller in the system which accepts inputs to compute braking and anti-skid commands. Aircraft speed and deceleration rate are used when auto brake is true. For brevity, the auto brake function has been excluded from discussion. The *BSCU* itself consists of two redundant Command and Monitor units. The Command units perform the computation to output the required braking command as well as the anti-skid command. The Monitor units supervise their corresponding Command units, and when deviation is detected in the first Command unit, the second unit is selected. When both Command units are detected to be invalid, *BSCU* is said to be invalid.

Two hydraulic pressure lines - Normal (green line powered by *GreenPump*) and Alternate (blue line powered by *BluePump*) - are used. The *GreenValve* and the *BlueValve* are used to control the pressure from the *GreenPump* and the *BluePump* respectively. In normal working condition, *GreenValve* and *BlueValve* are both open to provide constant stream of pressure to *SelectorValve*. The *SelectorValve* selects only one of the two redundant hydraulic systems to prevent a situation where both the green and blue system provide pressure to the brake, with the green line selected by default. This pressure is relayed to corresponding meter valves which adjust the valve position to output the required amount of pressure based on the command from *BSCU*. WBS is an output function which outputs the pressure.

The system switches to Alternate when one of these conditions occurs:

1) *GreenPump* produces pressure below threshold (or omitted)
2) Or when any other failures occur along the green line causing normal line output to fall below threshold (or omitted).

Once *BSCU* decides that Alternate line should be activated, it sends an OnAlternate signal which informs *SelectorValve* to inhibit any pressure from *GreenValve*. The

SelectorValve in turn engages the Alternate mode and relay pressure from *BlueValve*. Once the system switches to Alternate, it will not revert back to Normal. Figure 61 shows the basic system system structure. *NormalP*, *AlternateP* and *WBS* blocks are intermediate blocks which only propagate failures.

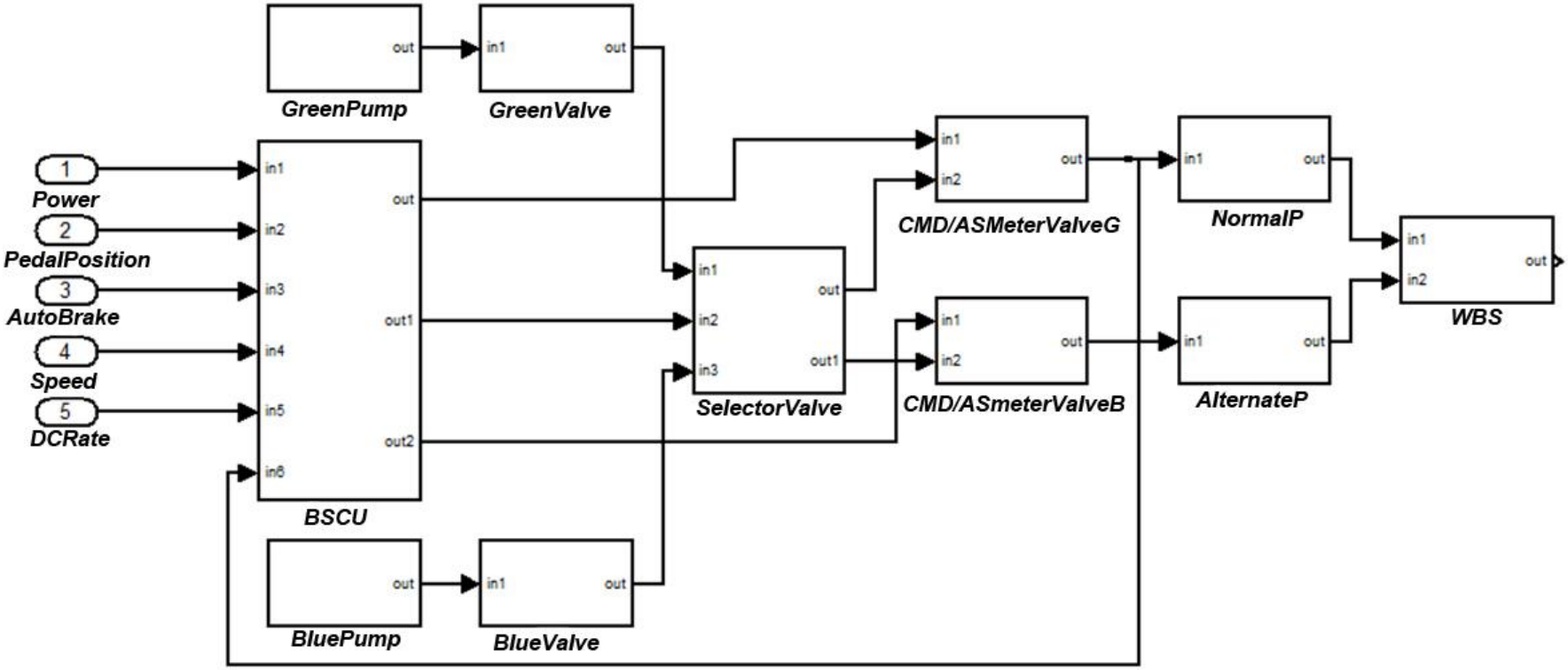


Figure 61: Simulink model of wheel brake system

5.2 FTA/FMEA

Once constructed, the system model is extended with failure information. To maintain simplicity, each component is assumed to carry one internal malfunction which directly causes omission of the component's output. *BSCU*, however, has two types of internal malfunction, which are related to the Monitor and Command units. Inputs to the *BSCU* are assumed to be supplied as intended. Table 15 summarizes the failure information for each component. Although the analysis has been largely focused on omission failures, other failure types like 'valve stuck at open' (causing commission failure) and 'valve stuck at value' can also be included.

Table 15: Internal failure for Wheel Brake System components

Component	Failure Mode	Description
GreenPump	GreenPumpBE	Internal malfunction in Green Pump which causes omission failure
GreenValve	GreenValveBE	Internal malfunction in Green Valve which causes omission failure
BluePump	BluePumpBE	Internal malfunction in Blue Pump which causes omission failure
BlueValve	BlueValveBE	Internal malfunction in Blue Valve which causes omission failure
CMD/Anti-SkidMeterValveG	GCMDASBE	Internal malfunction in the command/ anti-skid green line meter valve which causes omission failure
CMD/Anti-SkidMeterValveB	BCMDASBE	Internal malfunction in the command/ anti-skid blue line meter valve which causes omission failure
SelectorValve	selValveBE	Internal malfunction in the Selector valve which causes omission failure
BSCU	CMDBE	Internal malfunction in the BSCU which causes omission failure in both BSCU command units
	MonitorBE	Internal malfunction in the BSCU which causes omission failure in both BSCU both monitor units

Once failure extension of the model was completed, FTA and FMEA were automatically performed by HiP-HOPS. The derived FMEA shows how component failures contribute to the omission of pressure failure in WBS. As in the previous case study, the effects of component failures are distinguished between direct effects and further effects. The direct effect in Table 16 indicates that omission of the *BSCU* Command unit or the *SelectorValve* will directly contribute to the absence of WBS pressure. Additionally, the further effects table Table 17 shows that failures in hydraulic pumps, valves, and meter valves lead to omission of WBS pressure but only in combination with other failures.

Table 16: FMEA Direct Effects for Wheel Brake System

Components	**Failure Mode**	**Direct Effects**	**Severity**	**Comments/ Recommendation**
SelectorValve	selValveBE	O-WBS.pressure	Catastrophic	System should move to degraded mode. Failure should at most affect only anti-skid command. Introduce backup that read pedal positions separately.
BSCU	CMDBE	O-WBS.Pressure	Catastrophic	Backup mechanism should be introduced to provide pressure in the event of SelectorValve failure

Table 17: FMEA Further Effects for Wheel Brake System

Components	**Failure Mode**	**Effects**	**Severity**	**Contributing Failure Modes**
BluePump	BluePumpBE	O-WBS.pressure	Catastrophic	GCMDASBE
				GreenValveBE
				GreenPumpBE
BlueValve	BlueValveBE	O-WBS.pressure	Catastrophic	GCMDASBE
				GreenValveBE
				GreenPumpBE

CMD/AS MeterValveB	BCMDASBE	O-WBS.pressure	Catastrophic	GCMDASBE
				GreenValveBE
				GreenPumpBE
CMD/AS Meter ValveG	GCMDASBE	O-WBS.pressure	Catastrophic	BluePumpBE
				BlueValveBE
				BCMDASBE
GreenPump	GreenPumpBE	O-WBS.pressure	Catastrophic	BluePumpBE
				BlueValveBE
				BCMDASBE
GreenValve	GreenValveBE	O-WBS.pressure	Catastrophic	BluePumpBE
				BlueValveBE
				BCMDASBE

Direct reading of the FMEA shows that omission of either the *BSCU* Command unit or the *SelectorValve* directly lead to omission of pressure on the wheel-brake system. This absence of brake pressure is identified as a failure with catastrophic severity, and therefore single-points of failure CMDBE and selValveBE should be prevented. This can be achieved via introduction of backup mechanisms.

In this case, an *AccumulatorPump* and a *ManualMeterValve* are introduced to support the hydraulic system. An Accumulator is an energy storage device which contains built up pressure that can be released when both Green line and Blue line fail. The Accumulator supports the Alternate pressure line, and when activated the system is said to be in Emergency braking mode.

5.3 Revised Model

As shown in the revised model illustrated in Figure 62, the *AccumulatorValve* is introduced and placed between *SelectorValve* and *ManualMeterValve*. It receives and regulates pressure inputs from *SelectorValve* and *AccumulatorPump*. The *AccumulatorValve* also receives a signal from *BSCU* to indicate the activation of Alternate mode. When the system is running under Alternate mode and the *SelectorValve* is providing pressure, the *AccumulatorValve* does not produce any output. But in the case where Alternate mode is on and pressure from the *SelectorValve* is absent or falls under threshold, pressure from *AccumulatorPump* is released and supplied instead.

The second critical single-point of failure identified by the FMEA of the initial model is the combined failure of *BSCU* command units, denoted as CMDBE; this is a single failure representing the internal malfunction in both primary and secondary command units. CMDBE results in omission of the braking command signal being fed to the *CMD/ASMeterValve*s, which are designed to supply correct value of pressure according to braking command. This subsequently results in the omission of both normal and alternate pressure lines. One solution to avert this failure is by enabling the system to also obtain the braking commands directly from mechanical pedal position of brake pedals. This way, failure in *BSCU* braking command units will only result in the absence of skidding prevention instead of complete loss of pressure. *ManualMeterValve* obtains the basic braking command from *MechanicalPedal*, which reads the pedal position input directly. If pressure is provided from *AccumulatorValve*, *ManualMeterValve* supplies the braking pressure for the system in emergency mode.

The introduction of new mechanism means that FTA and FMEA analysis need to be updated. Iteration of the analyses is made efficient with the semi-automated nature of FTA and FMEA facilitated by HiP-HOPS. The results of HiP-HOPS analysis of the improved model which includes new components (*AccumulatorPump, AccumulatorValve, ManualMeterValve* and *MechanicalPedal*) and their failure annotations (partly adapted from (Johsi *et al.*, 2006) show that there is no longer any single-point of failure. We assume that elimination of single point failure is sufficient and the design is deemed to be acceptable for the next stage of the process.

The process continues with the construction of state machines that can be used for the purposes of a BSA. These should record normal modes where the system delivers its main function of delivering brake pressure and degraded modes where assistance function like anti-skid have been lost or sacrificed. Anti-skid feature is only provided when the system is operated under normal or alternate condition. This is done with the simplified assumption that the pedal position command does not propagate any failure. *NormalP, AlternateP* and *EmergencyP* are intermediate blocks which only propagates failures.

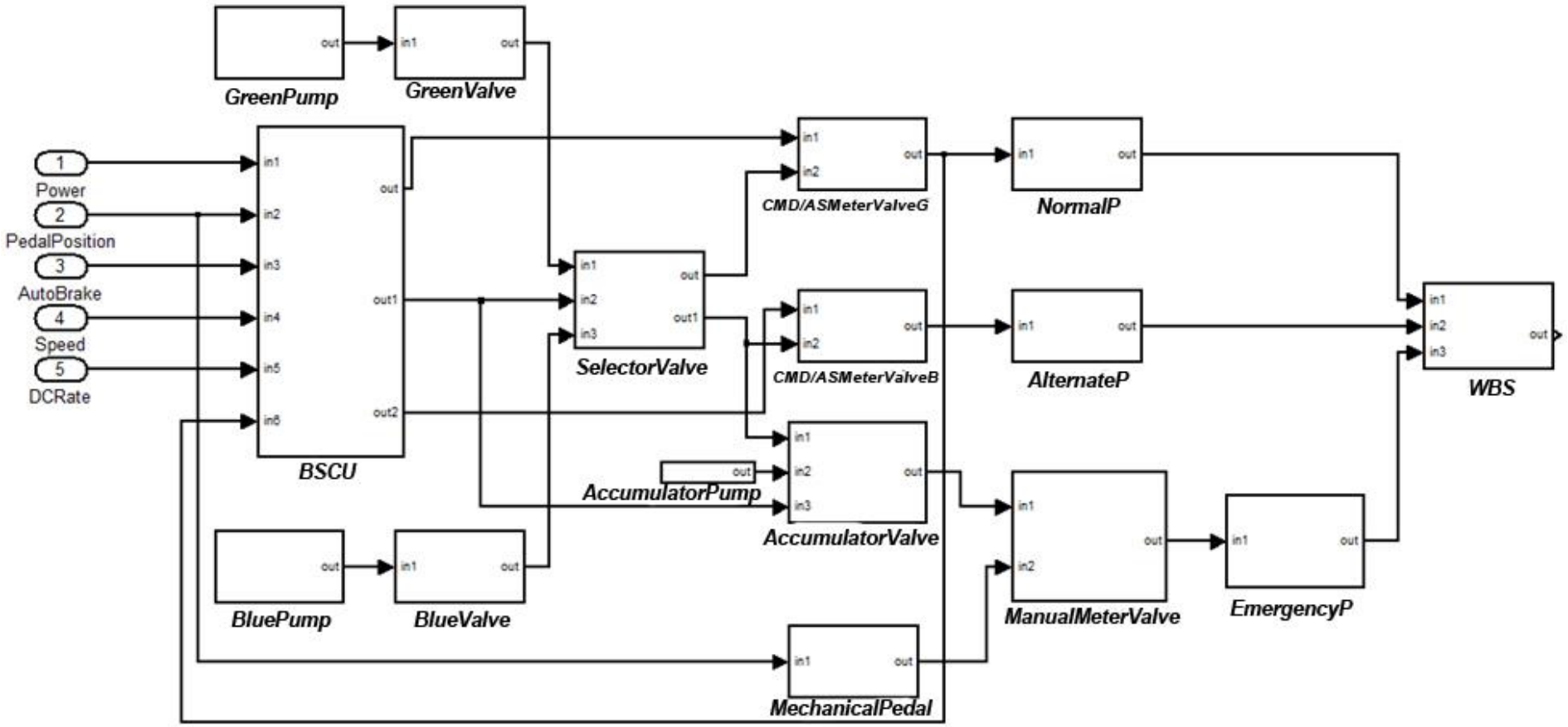


Figure 62: Revised model for wheel-brake system

5.4 Construction of Mode charts

The process proceeds to formulate system states based on delivery of functions. As mentioned earlier, WBS produces two different functions: provision of pressure and anti-skid. Although it is seemingly similar to the Brake-by-wire study presented in Chapter 4 (provision of brake pressure and ABS functions), we take slightly a different approach in grouping system modes according to the different way the model is presented.

Based on the delivery of these functions and the different hydraulic lines through which pressure can be supplied, one possible way to categorize system modes is as the following:

1) Normal (WBS_Normal) mode: where hydraulic pressure is provided by Green line, and anti-skid function is present.
2) Degraded1 (WBSD1_ALTERNATE): where hydraulic pressure is provided by Blue line, and anti-skid function is present.
3) Degraded2 (WBSD2_EMERGENCY): where hydraulic pressure is provided by Accumulator pump and anti-skid function is absent.
4) Fail (WBS_FAIL): where there is no hydraulic pressure provided.

Transitions between these modes can be formulated with the help of FMEA-ModeChart assistance table:

Mode	**Severity**	**Functions Delivered**	**Functional Failure Causing Transition**	**Target Mode**
WBS_Normal	-	Hydraulic pressure supplied through normal (green) line and anti-skid function is delivered	O-NormalP	WBSD1_ALTERNATE

WBSD1_ALTERNATE	Marginal	Hydraulic pressure supplied through alternate(blue) line and anti-skid function is delivered	O-AlternateP	WBSD2_EMERGENCY
WBSD2_EMERGENCY	Critical	Hydraulic pressure supplied through emergency line	O-EmergencyP	WBS_FAIL
WBS_FAIL	Catastrophic	No pressure supplied	-	-

Mode chart for WBS can be constructed based on the information from assistance table, which is shown in Figure 63 below:

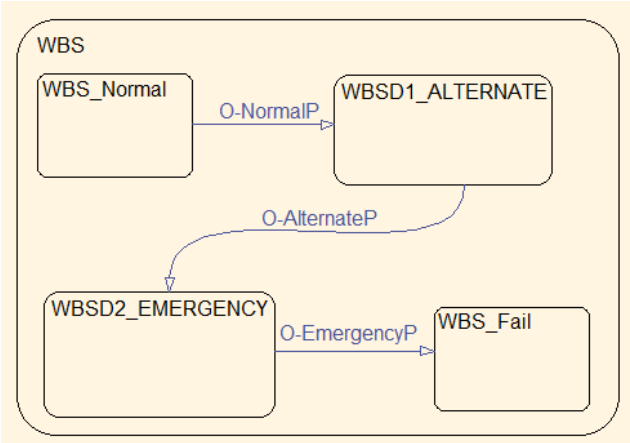


Figure 63: Abstract state machine for wheel brake system

5.5 Model Design Evolution from Requirement Verifications

Having constructed the abstract mode chart, we could then refine the state machines to engage the component behaviour. Translation from HiP-HOPS model to NuSMV automata model can be performed as described previously (please see section 3.8). This subsequently allows us to perform verifications on system model to ensure that it satisfies certain safety properties. As the design advances, both the formal model and safety properties are further refined to facilitate necessary lower level verification. It is then common to refine the state machine by including more specific parameter values (e.g. WBS pressure threshold in this case).

Verification can be performed with or without constraints on the maximum number of faults that can occur. A scenario for this example can be the verification of the property that: "*When there is omission of normal pressure, alternate will always replace it".* This property will not hold because after a certain number of component failures, alternate line will eventually fail. The specification can be revised by including specific assumptions, for example denoting the number of individual component failures deemed acceptable.

Another interesting aspect of model-checking of this system can be discovered during the verification of simple properties like:

SR5: *"When output is not supplied by Normal Line, and there is no failure accounted in Alternate line, pressure shall be supplied from Alternate line"*

This property does not hold, and NuSMV produces a counterexample trace that demonstrates how the condition is breached. The counter example describes a scenario where although omission in output from *CMD/ASMeterValveG* (Normal line) is 0, input deviation and internal malfunction in *CMD/ASMeterValveB* (Alternate line) is 0, output for Alternate line – which is expected to be 1 in this situation – is also 0.

Upon quick inspection, it is identified that the cause lies in the fact that the system employs dynamic cold standby. This information has been added manually to the NuSMV (state machine) models to reflect the different types of pressure line, and that only one can be active at one time. This means that the backup component is activated only when the primary component fails. To model this, we incorporate 'activation'

control as part of the dynamic behaviour modelling of the backup component. Activation signal is essentially used to indicate whether the component is expected to provide output. This is particularly useful to accurately describe an omission failure. In describing an omission failure, it is important to distinguish between absence of component output due to the component being not activated (not needed) or due to actual failures. Therefore, it is no longer sufficient to define the output of a component exclusively based on the negation of omission output, but to also take into consideration whether the component is activated. Manual intervention is required to include this additional information on as it was not contained in the CSA model.

To ensure that activation signal is taken into consideration within the modelling of CMD/ASMeterValveB for Alternate line, the following simple description exemplifies how output and omission of output can be described:

```
Omission-Output = Omission-Input OR internal_malfunction
Output = Active AND NOT(Omission-Output)
```

Activation properties can be introduced to describe a set of conditions related to the component(s) activation. This helps outline the assumptions needed for verification of a safety properties, e.g. to check whether a specification holds when a certain component, or a set of components, are activated. For example, we could define activation of BackupComponents as the activation of either meter valve in Alternate line or the activation of accumulator valve. This enables us to verify properties such as the following safety requirement:

SR6: *When Normal line is functioning, no backup mechanism shall be activated.*

Which can be expressed in the following CTL statements, where Backup_Active representing condition when either green or blue meter valves are active:

```
Backup_Active := CMD/ASMeterValveB.Active  OR AccumulatorValve.Active
AG((WBS.Status = Normal) -> NOT(Backup_Active))
```

Information to describe assumptions and activation control like this requires manual intervention, as they are not captured in the initial CSA annotated model.

Further examples of how component activation (or their deactivation) can affect the modelling and analysis assumptions in Altarica models can be found in (Bieber *et al.*, 2002).

In NuSMV automata model, an activation signal can be passed as an independent input parameter or can be assigned from the observation of other relevant input parameters (for example, the absence of specific component input). It is also possible to model the failure in the activation signal itself. However, in this example, the activation signal is assumed to be reliable and not associated with any failure.

Verification is also useful to uncover overlooked flaws in design. For example, another counterexample is produced when we are trying to verify the following properties:

SR7: *When both Normal line and Alternate line are not producing output, as long as there is no failure accounted along the emergency line, the system shall not fail*.

This property, again, does not hold and NuSMV returns a counterexample. The counterexample indicates that it is possible for a situation to occur such that: when internal malfunction CMDBE occurs in *BSCU* command units – resulting in omission failure in both Normal and Alternate lines due to the absence of braking command – the *AccumulatorValve* does not produce output. This subsequently leads to the omission of output in the Emergency line, causing the system to fail.

Upon closer inspection on the counterexample, it is revealed that this is because *AccumulatorValve* is assigned to monitor output from *SelectorValve*. It is designed only to produce output when *SelectorValve* fails to supply pressure when system is in Alternate mode. In this situation, however, *SelectorValve* is functioning correctly by supplying pressure, and therefore AccumulatorValve does not output any pressure. This result in the absence of pressure supplied to *ManualMeterValve*, and subsequently absence of emergency line which lead to system failure.

In comparison to an analysis performed based on CSA alone, this weakness would not have been detected. For example, we assume that condition 'system fails' refers to the top event Omission of WBS system, which could be modelled as:

O-WBS = O-NormalP AND O-AlternateP AND O-EmergencyP

For the analysis of requirement SR7, assumption is made that all the minimal cut sets for O-EmergencyP are false. The following presents list of minimal cut sets for O-EmergencyP:

MechanicalPedal.MPedalBE
ManualMeterValve.ManualMBE
BSCU.MonitorBE
AccumulatorValve.AccValveBE
AccumulatorPump.AccumPumpBE AND SelectorValve.selValveBE
AccumulatorPump.AccumPumpBE AND BlueValve.BlueValveBE
AccumulatorPump.AccumPumpBE AND BluePump.BluePumpBE

In the occurrence of BSCU.CMDBE failure which causes omission in both O-Normal and O-Alternate, if all these minimal cut sets for O-EmergencyP are false, O-*EmergencyP* is false. This implies that failure O-WBS will be false, indicating that the system will not fail. This could lead to a false belief that the design fulfilled SR7.

Model checker has demonstrated how weakness in logical connection like this can be uncovered.

One way to rectify this design weakness is by assigning *AccumulatorValve* to monitor output directly from *CMD/ASMeterValveB*. If output is not produced when system is on Alternate mode, *AccumulatorValve* should be activated and supply the required pressure. The following Figure 64 illustrates the revised model, which is assumed to have fulfilled the hypothetical list of safety requirements.

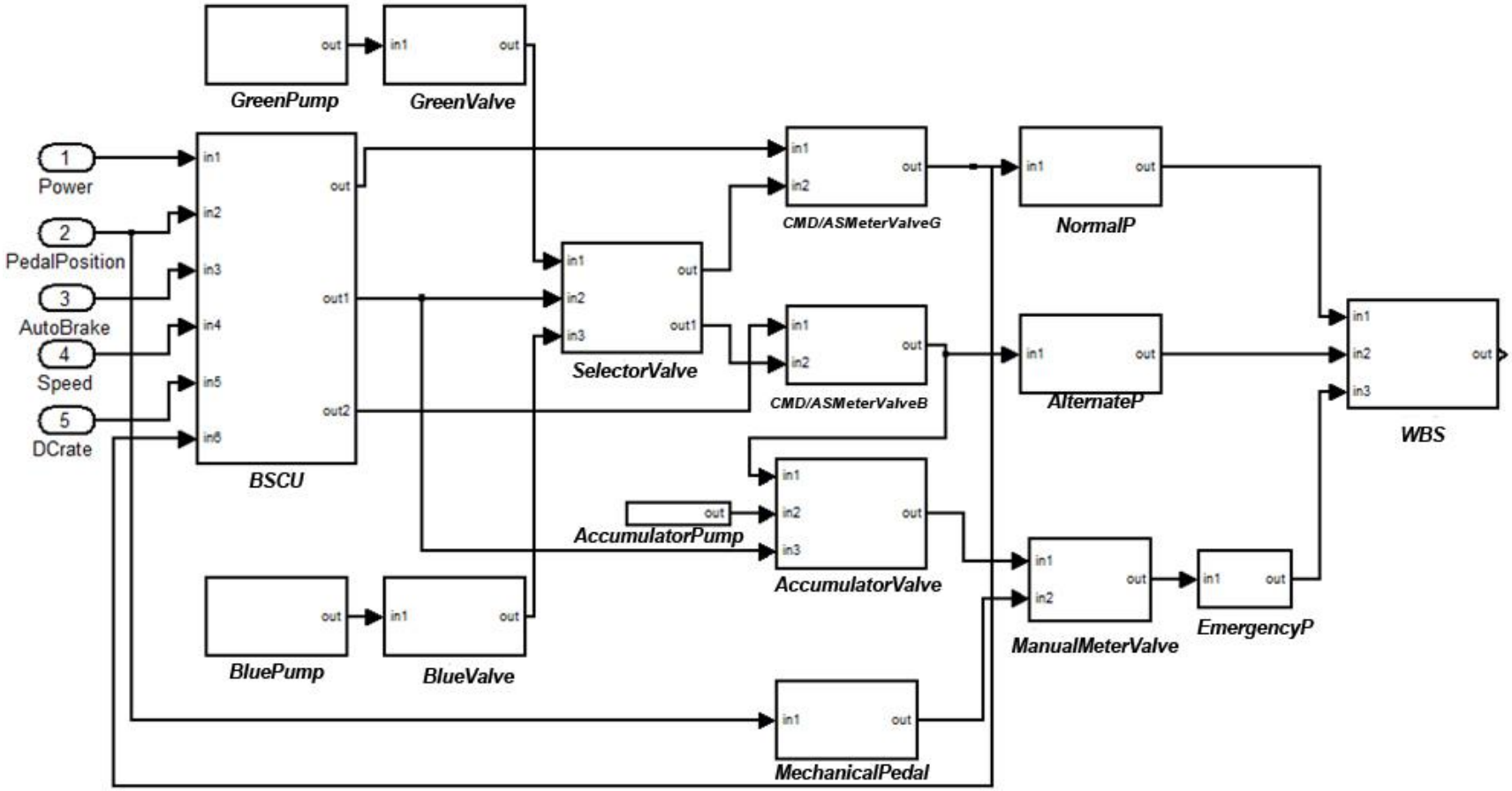


Figure 64: Revised model developed with assistance of model-checker

In summary, the application of IACoB and corresponding analysis results could help to either increase confidence about the safety of the design or identify design weaknesses that stimulate design iterations.

As it has been demonstrated in this case study, CSA and BSA techniques described can be iterated as the design evolves and undergoes changes and refinements. Overall, the process can contribute towards a more controlled approach towards safety, which does not allow safety properties simply to emerge at the end, but attempts to guide the design using the result of a continuous safety assessment.

5.6 Chapter Summary

This chapter presented a second case study on an aircraft wheel-brake system. To demonstrate the application of IACoB at a later stage, an architectural model of the system was presented. This case study focused mainly on the value of the model checking and how it influences the evolution of the design. IACoB starts with the FTA and FMEA performed on the initial model. This provided an assessment of the fault tolerant level of the system, and the identification of the system critical points. The model was revised based on these analysis results, before model checking was performed to verify safety properties or functional correctness of the components. Through the model checking, we discovered several behavioural aspects of the model which can be improved, which otherwise, could not be detected through the use of FTA/FMEA alone. These mainly involved control logic (for example, the activation control of a component).

CHAPTER 6. Detectability

Chapter 3 and Chapter 4 discussed the potential contribution of IACoB in fault tolerant design. The discussion so far has been focused on identifying critical points in the design, e.g. single points of hazardous failure, and on producing recommendations e.g. on location of functional and architectural redundancies. This chapter explores another aspect of fault tolerant design, that of "fault detection".

The term fault detection typically describes the process of identifying disturbances in processes and deviations from intended behaviour typically caused by component failures. Successful early fault detection means that measures can be taken to prevent the propagation of such disturbances. Fault detection has become increasingly important in many technical processes.

Components often incorporate mechanisms for detecting errors propagated through a system. These mechanisms, in practice, can also fail to detect the faults. The notion of *detectability* used in this thesis precisely refers to the probability of fault detection to be performed correctly.

6.1 Detectability in FMEA

In the current industrial practice, it is possible to extend an FMEA table with an additional column to allow description of the 'Detectability' of each failure. This can be done by identifying the means of detection, typically a monitoring mechanism that relies on observation of system parameters or an internal testing mechanism that constantly checks the health of a component. By studying this information, it is subsequently possible to establish how likely it is that the corresponding failure is detected. A detectability number can be assigned to rank the ability of these inspection techniques to detect failure modes. Detectability table, for example one that is presented in the Table 18 can be used to associate detection likelihood and the detection number. The assigned detectability number measures the probability of the failure goes undetected, which means a higher detection number signifies a higher probability that the failure goes undetected and therefore a lower probability of detection.

Table 18: Detection Evaluation Criteria (Quality Associates, 1997)

Detection	Criteria	Rank
Absolute Uncertainty	Design control will not and/or detect failure model; or there is no design control	10
Very Remote	Design control has very remote chance to detect failure mode	9
Remote	Design control has remote chance to detect failure mode	8
Very Low	Design control has very low chance to detect failure mode	7
Low	Design control has low chance to detect failure mode	6
Moderate	Design control has moderate chance to detect failure mode	5
Moderately High	Design control has moderately high chance to detect failure mode	4
High	Design control has high chance to detect failure mode	3
Very High	Design control has very high chance to detect failure mode	2
Almost Certain	Design control will almost certainly detect failure mode	1

An example fragment from the FMEA table of a vehicle braking system is given in Table 19. Failure of function to provide Pressure (Primary and Backup) leads to omission of braking. This failure can be detected with a pressure sensor, with a high likelihood of correct detection.

Table 19: Example of FMEA Table Extended with Detectability information

Function	Failure Mode	Effects	Contributing Failure	Severity	Detection Method	Detectability Number
Primary Pressure	Internal Failure	Omission of Braking	Backup Pressure	Catastrophic	Can be detected locally using feedback from pressure sensor	2
...	...	...	...	...	...	...

This calculation of detectability numbers in the FMEA is often used as part of the calculation for a Risk Priority Number (RPN). The RPN in a FMEA serves as a threshold value for evaluation of an action (or recommendation) against failure modes. An RPN is determined by calculating the product of severity, occurrence and detectability rankings. Recommended evaluation criteria for severity and occurrence can be found in (Quality Associates, 1997). Similar to severity ranking (discussed in Chapter 3), RPNs can be used to assist prioritisation of failure management.

6.2 Detection and Response to Failures

At the level of system architecture, fault detection commonly involves monitoring functions which check variables against anticipated behaviour and generate alarms when necessary. Related functions often include automatic protection functions which initiate counteractions in response to detected hazardous failure; and fault diagnostic functions that locate the root causes of detected faults. For simplicity, we use the term *detection module* to represent the collection of these fault detection and response mechanisms, and the term *target module* to represent the systems or components it supervises.

In practice, 'detection modules' can be refined into several different types. (Adachi *et al.*, 2010) and (Torres-Pomales, 2000) discuss four different types of common (particularly in software) fault detection and fault tolerance techniques: self-protection, self-checking, checkpoint-restart, and process-pair. Each technique uses a different approach in detecting and handling failure, e.g. by blocking or mitigating input failures to prevent them from reaching the target module, or by preventing a failure in its target module from propagating.

Self-protection aims to protect the target module by ensuring that it is protected from external disturbances. This is done by detecting failures propagated from other (input) modules. Self-protection is able to detect all failure modes, but does not possess any mechanism to recover from detected failures. Therefore self-protection is often assumed to fail-silent when it detects failure.

Self-checking enables detection of an internal error within the target module itself, and aims to block or mitigate the propagation of this failure. It requires internal information of the target module, and ports are established to enable this communication. When an internal failure occurs in the target module, the information will be sent to the self-checking module. If the self-checking module successfully detects the failure, it blocks or reduces the failure. For instance, by, replacing the missing value with default parameter value and feeding it back to target module. This allows the target module to continue to work appropriately, unless when self-checking experiences failure itself.

The Checkpoint-restart technique detects failures and enables recovery by restarting the target module to a predefined restore-point. The Process-pair technique, on the other hand, employs redundancy which includes two identical modules. Its detection and recovery mechanisms are similar to those of checkpoint-restart but when failure is detected, a process-pair can complete execution without returning to stored check-points. Instead, it uses redundant secondary module and when failure is detected, it switches from primary module to the secondary module.

Logically, these fault tolerant techniques could perform their different mitigation strategies only after they successfully detect anomalies in target modules. Traditional limit-value based supervision methods of monitoring and automatic protection is often done by checking the measurable output variables against allowed limits. Although this is a simple and reliable technique, Isermann (Isermann, 2004) highlights its main limitation in that they often rely on relatively large change in the measurements, either after a large sudden failure or longer-lasting gradually increasing failure. It further discusses model-based fault detection techniques (for example, methods which are based on parameter estimation, parity equations, and state observers). Although these techniques improve classical fault detection methods, in practice there are cases where the detection module experience subtle failure and these failures affect the effectiveness of detection. Consequently, there is a need to represent and take into consideration the failure of the "detection module" itself during the modelling of the system failure behaviour. This is precisely an area where this thesis has hoped to make a contribution.

6.3 General Modelling of Detectability

The inability of a detection module to correctly detect failure of the target module also means inability to take corrective action. Further analysis shows a number of common scenarios:

1) Case 1: The detection module fails to detect, and the failure of the target module is simply propagated to other parts of the system.
2) Case 2: The detection module wrongly signals detected failure and inadvertently acts in the absence of failure in the target component.

3) Case 3: The detection module correctly detects failure, but then malfunctions and corrective measures fail to correct the failure of the target component.

These scenarios highlight the different situations where a failure in detection module does not only cause the inability to prevent failure (through provision of counteraction), but also potentially affect the transformation between different failure types, or even the occurrence of new failures stemming from the detection module itself.

Transformation between different failure types can also sometimes be part of the nominal behaviour of the detection module. In some scenarios, it is regarded as acceptable to transform one hazardous failure into different type of failure which has less hazardous consequences. For example, in a fail-silent scheme, the detection module is responsible for transforming value or commission failures into omission failures. But this failure transformation can only occur after the detection module has successfully detected the value or commission failure. Note that to model this, it is no longer sufficient to represent failure behaviour because the "success" of the detection module can also contribute to a different more benign failure effect. In this case, failure of the detection module means propagation of hazardous commission and value failures, while its success means transformation of these hazardous failures to more benign omissions.

In order to describe such situations in the context of CSA, and more specifically in HiP-HOPS analysis, we introduce the following general representation to describe the behaviour of a detection module:

1) Event *Failure* (representing internal malfunction): An internal malfunction of a detection module can affect failure behaviour just like any other component malfunction, and can be treated and analyzed as such. Figure 65 below illustrates an example of how internal malfunction *Failure* in detection module can play a part in the modelling of system failure. Detection_Module supervises the output of Target_Module. In the occurrence of internal malfunction BE, which causes omission failure, Detection_Module performs counteraction and provides correct output. However, Failure in Detection_Module alone is enough to cause omission failure. The omission failure expression of Detection_Module can be summarized as:

O-Detection_Module.Out = *Failure*

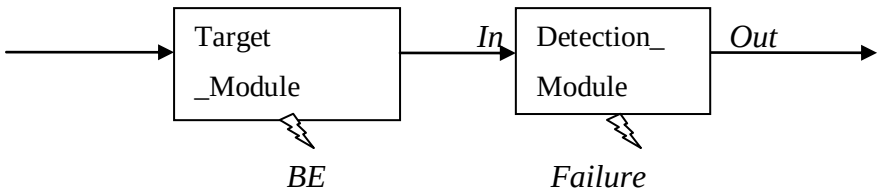


Figure 65: Internal malfunction in Detection Module

2) Event *Miss:* We assign a separate event to represent the occurrence of situations where the detection module fails to detect target module failure. Event *Miss* causes the failure of the target module failure to go undetected. For example, Figure 66 illustrates a situation where omission in Detection_Module can also be caused by it failing to detect omission in input deviation from Target_Module. This can be expressed as:

O-Detection_Module.Out = *Failure* OR (*Miss* AND O-In1)

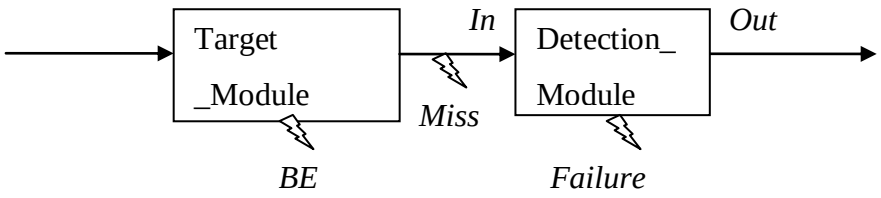


Figure 66: Event *Miss* in Detection_Module

3) Event *NOT Miss (¬Miss):* To describe situations where detection module in its working condition causes a failure, the complement event *Not Miss is* introduced. One common use of *Not Miss* is in the representation of failure transformation. For example, in Figure 66, a timing failure in Detection_Module can be caused by it detecting an omission failure in Target_Module. This happens when it tries to recover

(e.g. provide) the omitted parameter, and in doing so, become late in the timing. This can be expressed as:

$$\text{T-Detection\_Module.Out} = \neg \textit{Miss} \ \& \ (\text{O-In})$$

Classical HiP-HOPS modelling only uses coherent failure logic that uses AND and OR gates. The proposed type of modelling which also incorporates NOT gates enables to distinguish between success and failure and represent different effects in these two circumstances. With this modelling, it is now possible to represent all scenarios of failure presented in Cases 1 to 3 above.

6.4 General Analysis of Detectability

In HiP-HOPS analysis, events *Failure* and event *Miss* are treated like any other component internal malfunctions. In fault tree synthesis these are regarded as basic events. Consequently, the HiP-HOPS analysis techniques presented in Chapter 2, more specifically the MICSUP algorithm, can be used in fault tree analysis.

The inclusion of the complement ¬*Miss*, however, creates some complications. ¬*Miss* is implemented with the use of the NOT operator in fault trees. Traditionally, the use of negation of a failure event in failure modelling has generally been discouraged for several reasons. Firstly, it is often assumed that a component in its working condition should not contribute to system failure. In cases where it does, traditional solutions often suggest design modification to prevent it. It is also a common notion that the probability of the negation of failure event is almost always close to 1 which means it can be safely ignored in quantitative analysis. The inclusion of 'NOT' also results in the introduction of non-coherent structure which increases the complexity of analysis. Despite these arguments, (Johnston & Mathews, 1983) and (Sharvia & Papadopoulos, 2008) reviewed several scenarios where inclusion of NOT benefits failure modelling. These include conditions where the failure probability of a component becomes significant enough for the working probability to be included in quantitative analysis. This is often true in conditions that exceed the operation specification for a component. The NOT operator is also important in the failure modelling of some multitasking and

phased-mission systems. In other cases, the use of negation operator also assists the development of repair schedule for components.

In addition to these, we also argue that the use of negated event is required to allow more accurate representation of detectability in a model.

One alternative to using negated events is by treating the absence of failure *Miss* as a separate independent event. For example, by using new event *Catch* to represent this case instead of ¬*Miss*, which subsequently allows the model to maintain coherent structure. This, however, has been known to cause inaccuracy in the quantitative analysis (for example, calculation of failure probability), as demonstrated later in the example section. This is very likely due to the way quantitative analysis is performed in non-coherent structure, where 'hidden' cut sets (termed prime implicants in non-coherent structure) can potentially be produced.

To facilitate this type of modelling for detectability, HiP-HOPS has been extended with the ability to perform non-coherent analysis (Sharvia, 2008). The analysis of non-coherent fault trees in HiP-HOPS is implemented through an extension to MICSUP algorithm to allow iterated Consensus. The Consensus Law states that:

$$A.B + \neg A.C = A.B + \neg A.C + B.C$$

Which describes that if event B causes system failure when event A fails, and event C causes system failure when event A works, then the combination of event B and event C will inevitably causes system failure regardless the state of A. In such circumstances, then B.C is known as a 'hidden' prime implicant set that can be identified by the application of consensus.

To enable quantitative analysis, quantitative information can be assigned to each of these detectability parameters. For example, for the calculation of probability of events in the model, failure rate λ can be assigned for internal malfunctions of type *Failure*, and a fixed probability value can be assigned for events *Miss and* ¬*Miss*. In practice, the assignment of this detectability probability ν often relies on the degree of dependence. For a module with higher dependence on other modules (for example, because it requires information processed by other modules), the probability of *Miss* is likely to be high.

In an NuSMV model, detectability information can be translated to and treated as part of the module internal variable.

6.5 Example

6.5.1 Cruise Control System

This section presents a simplified adaptive cruise control system to demonstrate the application of detectability analysis. The system assists the driver by automatically adjusting vehicle speed to maintain safe following distance. It typically uses a radar sensor to monitor the vehicle in front and adjust vehicle speed to keep it at a pre-set following distance. The system is also extended with a brake support function.

Figure 67 illustrates the functional structure of the basic adaptive cruise system. It is an adaptation of a related driving assistance system, pre-collision detection system, presented in (Adachi *et al.*, 2010). The system gets its input from a set of sensors including *Radar_Sensor, Speed_Sensor, Pedal_Sensor, Switch_Sensor*. *Radar_Sensor* provides reading from the wave radar. *Speed_Sensor* provides information about the current speed of the vehicle. *Pedal_Sensor* provides information on the driver's operation (for example, input in accelerator or brake pedals). *Switch_Sensor* provides information on the selection of modes (for example whether cruise control is activated). And Memory provides information on pre-stored data.

Monitoring_Module gathers information from several sensors and provides signals regarding distance and relative speed of the vehicle ahead. This pre-processed information is then passed to *Logic_Module* which computes the distance between the vehicles and determines how fast the vehicle is approaching the vehicle ahead. Based on the pre-set desired following distance, it determines the appropriate time to start deceleration (or acceleration when the traffic is cleared). *Brake_Support* system aims to provide effective braking and assistance in cases when collision is imminent. For example, in design discussed in (Ford, 2010), if pressure on accelerator pedal is released quickly, indicating driver's desire to slow down, the system can apply brake pads against the brake disk even before the driver presses the brake pedal. This decelerates

the vehicle faster than the driver can move their foot to the brake pedal. In doing so, it shortens braking reaction time and braking distance. This information is supplied to *Cruise_Control*, which coordinates the information and decides on appropriate actions. It sends signals to corresponding Actuator_Module (i.e. brake or throttle) to perform appropriate actions.

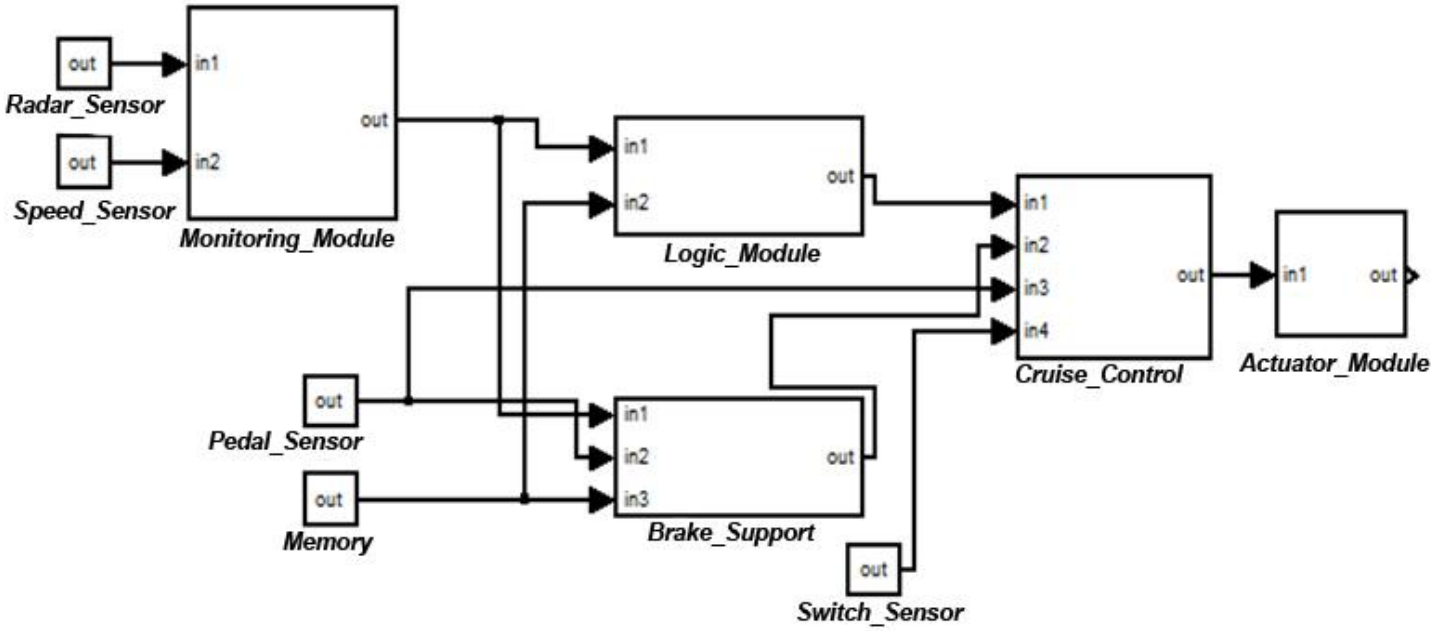


Figure 67: Cruise Control System

One way to demonstrate the contribution of detectability modelling is through quantitative analysis of the effect of detection of component failures to the failure probability (or unreliability) of the system. Before quantitative data is assigned, we annotate the cruise control system with failure information. To maintain simplicity, each input sensor is assigned with a uniform internal malfunction *BE* and each of the other main modules is assigned with two internal malfunctions, *BE1* and *BE2*. BE1 is a failure that is generically related to omission and value failure (for example, failure in physical or hardware), while BE2 is a failure that is generically related to timing and commission failure (for example, failure in software algorithm).

Hypothetical failure rates are assigned for each of the internal malfunction to illustrate the validity of approach. The failure rate for internal malfunction in sensors and memory (BE) is assigned at $\lambda = 1.15\times10^{-7}$ and failure rates of internal malfunction in other modules (BE1 and BE2) are assigned at λ 4.6×10^{-7} and $\lambda = 1.12\times10^{-6}$ respectively.

The following table presents the summary of failure expressions for the output deviations in the cruise control system modules. Omission, commission and value failures at module outputs are discussed:

Table 20: Failure information for Cruise Control functions

Function	**Output Deviation**	**Failure Expression**
(All) Sensors	O-out	BE
	C-out	BE
	V-out	BE
Monitoring_Module	O-out	BE1 OR O-in1 OR O-in2
	C-out	BE2 OR C-in1 OR C-in2
	V-out	BE1 OR V-in1 OR V-in2
Logic_Module	O-out	BE1 OR O-in1 OR O-in2
	C-out	BE2 OR C-in1 OR V-in1 OR V-in2
	V-out	BE1 OR V-in1 OR V-in2
Brake_Support	O-out	BE1 OR (O-in1 AND O-in2) OR O-in3
	C-out	BE2 OR C-in1 OR C-in2 OR V-in1 OR V-in2
	V-out	BE1 OR V-in1 OR V-in2 OR V-in3
Cruise_Control	O-out	BE1 OR O-in1 OR O-in2 OR O-in3

		OR O-in4
	C-out	BE2 OR C-in1 OR C-in2 OR C-in3 OR C-in4
	V-out	BE1 OR V-in1 OR V-in2 OR V-in3 OR V-in4
Actuator_Module	O-out	BE1 or O-in1
	C-out	BE2 or C-in1
	V-out	BE1 or V-in1

6.5.2 Detectability in Cruise Control

Once the model is annotated, we perform analysis on the cruise control system without the inclusion of any fault tolerance technique. For the purposes of this discussion, the quantitative analysis is performed examining omission, commission and value failures of Actuator_Module (which can be expressed as O-Actuator_Module.out, C-Actuator_Module.out, and V-Actuator_Module.out respectively) which form the top event if fault trees and effects in FMEAs produced by HiP-HOPS. With the failure rates provided, the probability of these events can be calculated.

We assume that in the earlier stage of analysis, C-Actuator_Module.out has been identified as being more critical than the other failure types. The process to identify critical points in the system contributing to this failure can be performed (as discussed previously in Chapter 3 and Chapter 4), and detection modules can be assigned accordingly to address this. To maintain the simplicity of this example, although there are a number of contributing internal malfunctions that can contribute to C-Actuator_Module (for example, BE in input sensors or BE2 in *Logic_Module* among others), we place focus on *Monitoring_Module* and *Brake_Support*.

The detection modules to be added in this architecture can be realized in various implementations, adopting different structures and characteristics. In practice, multi-objective optimisation techniques can be employed to help determine the optimal solutions. HiP-HOPS itself incorporates multi-objective optimisation capabilities (Parker, 2010), but their use was deemed out of scope in this work.

Here we arbitrarily select one possible implementation of detection modules. It is presented as a basis for evaluating detectability and therefore by no means represents an optimal design solution. Although in practice each detection module might possess

different configuration and fault tolerant characteristic, all detection modules in this example are identical in their function, they all enforce fail-silence in response to detected commission failures. Figure 68 illustrates the design of the system with detection modules incorporated. The detection module placed between *Monitoring_Module* and *Logic_Module*, *DM_LM*, aims to prevent further failure propagation from *Monitoring_Module*. To achieve this, detection module *DM_LM* transforms commission and value failure into omission failure. The detection module placed between *Brake_Support* and *Cruise_Control* modules, *DM_CC*, has the same objective and causes failure propagated from *Brake_Support* to fail silent.

With the rationale that fault tolerant components are reasonably more reliable than its target modules, the failure rates for internal malfunctions *Failure* in *DM_LM* and *DM_CC* are both assigned a lower failure rate of $\lambda = 1.12 \times 10^{-7}$. The probability of event *Miss* in detection modules is assigned a fixed probability of $\nu = 0.12$.

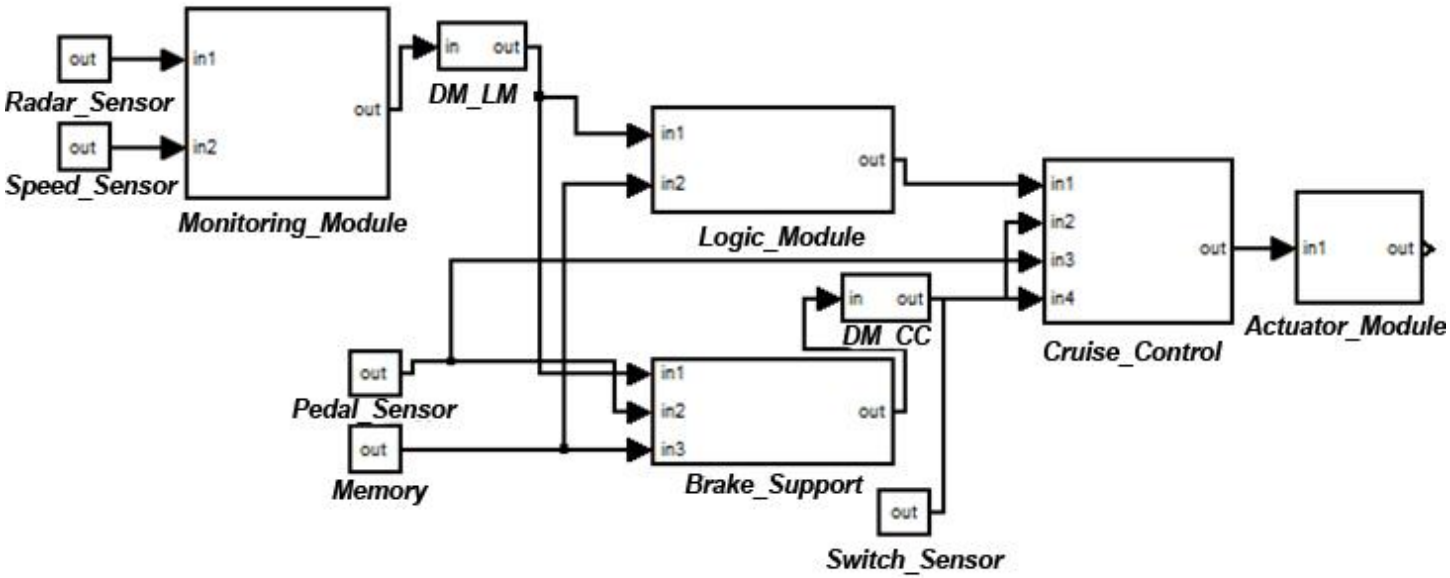


Figure 68: Cruise Control with Detection Module

Table 21 summarizes the failure expression of detection modules *DM_LM* and *DM_CC*.

Table 21: Failure information for Detection Modules

Function	Output Deviation	Failure Expression
DM_LM	O-out	Failure OR O-in OR (¬Miss AND (C-in OR V-in))
	C-out	Miss AND C-in
	V-out	Miss AND V-in
DM_CC	O-out	Failure OR O-in OR (¬Miss AND (C-in OR V-in))
	C-out	Miss AND C-in
	V-out	Miss AND V-in

One obvious effect of the introduction of detection modules is the improvement in system reliability. For example, for top event C-Actuator_Module.out, probability declines from 0.064 to 0.039 with the use of detection modules. Probability for top event V-Actuator_Module.out also decreases from 0.028 to 0.018. This is compensated by the increase in probability for top event O-Actuator_Module from 0.028 to 0.058 as other failure types are transformed into omission failure. But since omission is deemed more benign than inadvertent application of function, this is acceptable.

To show the significance of detectability modelling, we also compare the analysis between situations where detection modules are assumed to model only internal malfunction that represents its own failure behaviour (*Failure*) and situations where in addition to this, they also models scenarios where they fail to detect failures of the target module (*Miss*). As expected, the probability for all top events of fault trees increase as *Miss* is introduced. A Summary of tabulated analysis results is presented in APPENDIX C.

The significance of the inclusion of detectability modelling in enabling a more accurate qualitative analysis should also be highlighted. For example, the transformation of commission and value failures to omission failure requires the *Miss* event not to occur. To accurately model omission failure DM_LM which expresses this condition, the *Not Miss* event is employed. This allows the failure expression to be written as:

$$O\text{-}DM\_LM.out = failure\ OR\ O\text{-}in\ OR\ (\neg miss\ AND\ (C\text{-}in\ OR\ V\text{-}in))$$

As mentioned earlier, the inclusion of the NOT operator results in a non-coherent analysis. To maintain a coherent fault tree structure, ¬*Miss* event can be replaced with an independent *catch* event.

The incorporation of the NOT operator in the failure expression of the detection module can have a significant quantitative effect on the probabilities of system-level output deviation (*Actuator_Module*). It can either increase or decrease the probability of system failure according to the (different) sets of prime implicants produced compared to the use *Catch*.

To examine this option of treating ¬*Miss* as an independent new event *Catch*, we will use a revised version of the model. We now introduce a new processing module and an additional detection module into the cruise control system as shown in Figure 69. The new processing module, *Fading_Brake* (Autopressnews, 2006) aims to gradually build up the braking pressure in conjunction with constantly hard braking to help reduce the risk of wear and retained pedal feeling. It supplies information to achieve this to *Brake_Support* module. *Fading_Brake* possesses identical internal malfunctions (BE1 and BE2) and failure rates to the other main processing modules.

The detection module *DM_FB* is attached to *Fading_Brake* and operates in a similar way to a backup structure or a check-point-restart technique described in (Adachi *et al.*, 2010). When *Fading_Brake* experiences failure, *DM_FB* restarts the module to a pre-stored reset checkpoint. This subsequently causes transformation between different failure types. For example, when *DM_FB* detects omission or failure in *Fading_Brake* and resets the module, a value failure will inevitably occur as parameters reset into (and execution continue from) their pre-stored point. Similar to previous detection modules, the failure of *DM_FB* is assigned at $\lambda = 1.12\times10^{-7}$ with probability of event *Miss* $\nu = 0.12$.

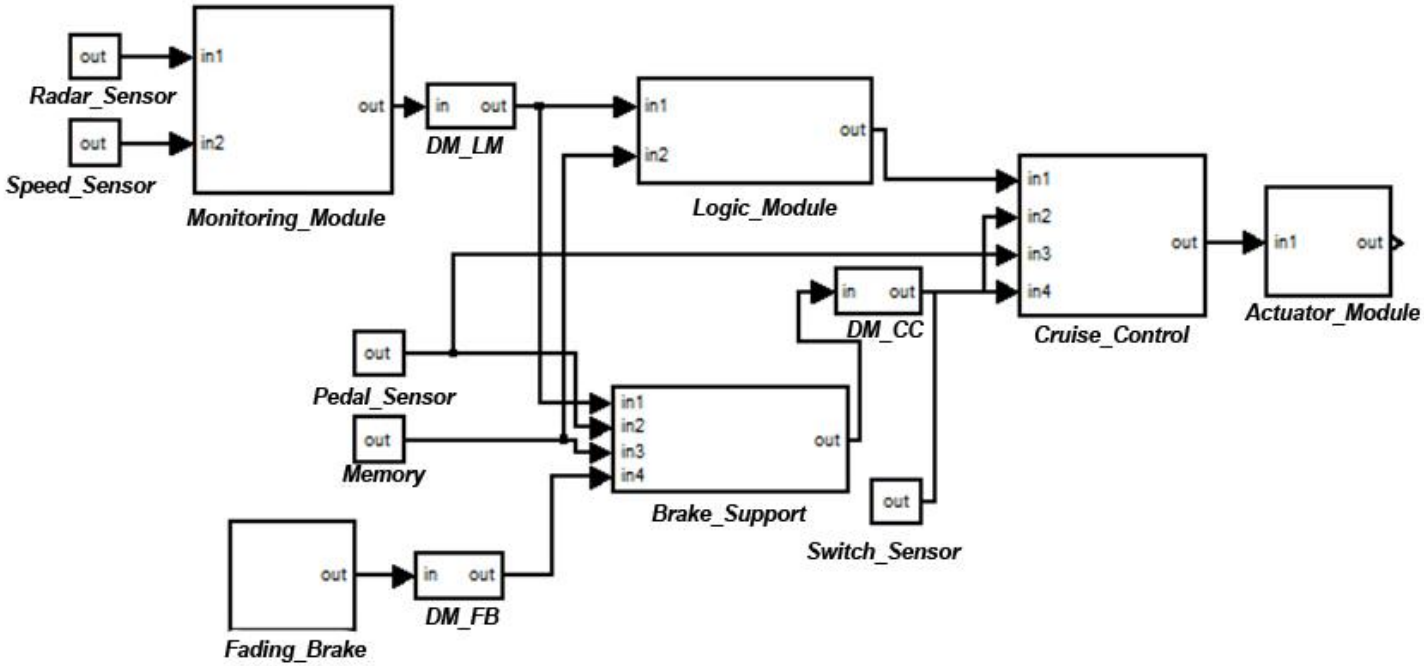


Figure 69: Cruise Control System with Fading Brake

Table 22 summarizes the failure information for *Fading_Brake* module, detection module *DM_FB* and updated *Brake_Support*. Based on this failure information, the system model is updated and analyzed.

Table 22: Failure Information for Cruise Control with Fading Brake

Function	Output Deviation	Failure Expression
Fading_Brake	O-Out	BE1
	C-Out	BE2
	V-Out	BE1
DM_FB	O-Out	Failure OR (Miss AND O-in)
	C-Out	Miss AND C-in
	V-Out	(NOT Miss AND (C-in OR O-in)) OR V-in
Brake_Support	O-out	BE1 OR (O-in1 AND O-in2) OR O-in3 OR O-in4
	C-out	BE2 OR C-in1 OR C-in2 OR V-in1 OR V-in2
	V-out	BE1 OR V-in1 OR V-in2 OR V-in3 OR V-in4

To demonstrate the role of non-coherent structure, we replace ¬*Miss* event with an independent *Catch* event in the expression. This time the analysis produces more interesting results. The probability for top event O-Actuator_Module.Out with the use of ¬*Miss* in all detection modules is 0.072, and when *Catch* is used as replacement the probability changes into 0.064. This is a substantial disparity in probability which can mislead designers into accepting models which do not meet reliability requirements.

This disparity in probability calculation can be attributed (as explained previously) to hidden prime implicants generated through Consensus algorithm. This is demonstrated by further studying the resultant prime implicant sets. Although the analysis of both fault trees produces the same number (19) of prime implicant sets, the sets are not completely identical.

The full prime implicant sets for both fault trees are included in the Appendix D. We compare the prime implicant sets produced by both fault trees (*NOT Miss* and *Catch* regarded as interchangeable accordingly) and the following differences are highlighted:

1) The following prime implicant is produced by analysis of non-coherent fault tree (uses *NOT Miss*), but is not contained within the coherent fault tree (uses *Catch*):
   Monitoring_Module.BE2 AND DM_CC.NOT_Miss (exp.1)

2) The following prime implicant is produced by analysis of coherent fault tree but contained within the non-coherent fault tree:

Monitoring_Module.BE2 AND DM_CC.Catch AND DM_LM.Miss (exp.2)

To understand these differences, we studied the rest of the prime implicant sets which are identical between the fault trees. Upon closer inspection, it is identified that prime implicant *MonitoringModule.BE2 AND DM_LM.NOT_Miss* and *Monitoring_Module.BE2 AND DM_CC.NOT_Miss AND DM_LM.Miss* (or *Monitoring_Module.BE2 AND DM_CC.Catch AND DM_LM.Miss*) are originally produced from the analysis of both fault trees. The difference is that in the analysis of the non-coherent fault tree, these prime implicants produce a new hidden prime implicant through Consensus (exp.1): *Monitoring_Module.BE2 AND DM_CC.NOT_Miss*. Occurence of the latter is sufficient to cause system-level failure regardless of the presence of *DM_LM.NOT_Miss* or *DM_LM. Miss.* This in turns eliminates *Monitoring_Module.BE2 AND DM_CC.Catch AND DM_LM.Miss (*exp.2) which becomes redundant. On the other hand, the coherent fault tree is not able to establish the link between event *Miss* and *Catch*, and is therefore unable to produce the hidden prime implicant in (exp.1). It subsequently retains (exp.2).

This demonstrates that the use of NOT operator (and non-coherent analysis) helps produce a more accurate result in detectability analysis. In the translation process of a model to NuSMV, the parameters for detectability can be included in the module internal variables like other basic events.

6.6 Chapter Summary

This chapter introduced the concept of detectability in the context of CSA, which describes the ability of a module to correctly detect errors. Its role within fault tolerant design is explored. The method introduces failure-relevant parameters, which model the events where errors are correctly detected (or not) in addition to the internal malfunction of the detection module. These parameters can be modelled as part of the failure information in HiP-HOPS.

The general analysis of these parameters can be performed as a part of the CSA analysis. Commonly-used fault tree analysis algorithms can be used, except for the events where errors are detected (i.e. it is NOT missed by detection module). This notion of detecting error correctly leads to the inclusion of NOT operator, and subsequently a non-coherent fault tree structure. To enable the analysis of non-coherent fault tree, HiP-HOPS synthesis and analysis algorithms were extended, and Consensus algorithm was implemented.

The use of NOT operator in a fault tree has been long debated. Here in the context of detectability, we showed how it affects the accuracy of quantitative analysis. A small example of cruise control system is presented to show the application of detectability. In an effort to maintain a coherent structure of the fault tree, an alternative was explored. The use of NOT operator has been shown to contribute to a more accurate top-event probability calculation.

CHAPTER 7. Conclusions

7.1 Contributions

Compositional Safety Analysis and behavioural safety analysis techniques have emerged as two separate and competing paradigms for performing model-based safety analysis.

This thesis argued that the traditional gap between the two approaches can be overcome, as CSA and BSA are effectively combined in a novel model-based design and safety analysis process which therefore benefits from the advantages of both approaches, namely the flexibility, early applicability and scalability of CSA and the precision, behavioural analysis capabilities and detailed insights offered by BSA (see also statement of "hypothesis" in Introduction).

To assess and support this research hypothesis, several objectives were defined. In the following discussion, we revisit these objectives and summarize how, and to what extent, they have been achieved:

Objective 1. To examine CSA and BSA techniques and investigate their strengths, limitations, and application in different stages of design development. This thesis determines complementary aspects of these techniques that can be exploited via synergistic combined application.

This thesis investigated CSA and BSA characteristics and identified potential for integration. The review of several prominent CSA and BSA techniques presented in Chapter 2 provides insight into the different characteristics, working mechanisms and applicability of each technique.

The strength of CSA lies in the simplicity of its Boolean-based analysis approach. This makes it possible for safety analysis to be performed in a quick and iterative manner. Fault tree synthesis can be performed in linear time and overall the analysis scales up to large and complex models. CSA also facilitates a 'divide-and-conquer' approach which becomes the basis of its compositional nature. The failure analysis of a complex system

can be constructed based on the composition of failure analyses of its components. This subsequently makes the process easily manageable. CSA also produces safety artefacts, namely fault trees and FMEA, which are familiar to safety analysts and therefore eases their engagement. CSA is generally used for reliability engineering and it is possible to extend this with advanced capabilities for design optimization. Fundamental limitations for CSA include the fact that there is no support for formal verification of safety properties and that CSA facilitates mostly analysis of static models.

BSA, on the other hand, uses brute-force exploration to assess system behaviour. This exhaustive exploration provides explicit assurance of model correctness with respect to safety specifications. BSA often employs model-checking to perform this verification. Despite these strengths, BSA can only be applied at a later stage of the design, where the design model is relatively mature. This is unfortunate because changes at later stages are often costly, and the technique misses opportunities to effectively influence design process earlier.

From the study of these characteristics, we identified the complementary aspects which lend themselves to the foundation of integration. First, we looked into the different stages of the system development where each technique can be employed. CSA is generally applicable from the early PSSA stage up until the end of the design. It is applicable to early, experimental models and can be iterated as the design becomes more detailed. BSA, in contrast, is generally applicable towards the end of the PSSA, and requires formal and more detailed mature models.

CSA and BSA also aim to achieve different assessment objectives. Although both can be used to analyze possible causes for failures, CSA aims to identify safety problems early in the design by showing the causes of system failure; while BSA provides verification of formal models with regard to safety properties. A combined application allows us to achieve wider analysis coverage and a more robust assessment. The ability to introduce BSA verification capabilities early in the development stage is particularly valuable.

In this thesis, we have shown that CSA and BSA have different objectives and different, complementary strengths and weaknesses. We have therefore made a case for their synergistic combined application. “Synergistic combined application” refers to the

process of harnessing analysis results (or safety artefacts) from CSA and BSA for the benefit of both techniques in an incremental and continuous manner. The process and the activities involved are explained further in the next objectives.

Objective 2. To propose a systematic method to utilize analysis results from CSA and BSA in the course of design. This involves investigating how input to each technique can be systematically constructed, in particular, how results of CSA can assist the construction of behavioural model for BSA's formal verification. It is also important to understand how these results can provide constructive feedback to designers towards an iterative system modelling process.

This thesis has developed, IACoB, a novel method for combined synergistic application of CSA and BSA. Following the review of prominent CSA-based and BSA-based techniques, we have decided to select HiP-HOPS to facilitate CSA and NuSMV model checker to facilitate BSA, based on the arguments considered in Chapter 2. In Chapter 3, the IACoB safety analysis process has been introduced and developed to describe the integrated application. IACoB was developed as a process method which allows BSA to be performed following CSA by building upon its analysis results and safety artefacts.

The process starts with a system model, which can be an early functional model or a more detailed architectural model. Model construction is followed by an analysis of effects of failure or a severity assessment phase, in which the severity level of failures of output functions or components is determined. To enable CSA, the elements of the system model are then annotated with local failure behaviour. This allows the HiP-HOPS tool to automatically perform fault tree and FMEA synthesis and analysis. The results of CSA offer constructive feedback for designers by providing them with information on failure causes and assisting the quick identification of weak points in the design. This ultimately helps contribute to a better revised design.

Once designers are assured by the CSA results, FTA and FMEA results are used to assist the construction of a behavioural model of the system for BSA. Behavioural models for BSA can be classified in two generic groups according to the stage of design where these models are developed. In early functional design where information on dynamic behaviour is not widely available, an abstract mode chart can be constructed from FMEA results. An FMEA-ModeChart Assistance Table is used to help organize

the core elements of the mode chart. The mode chart captures transitions of the system from normal to degraded and failed modes in response to the failures predicted in the FMEA.

Note that in this approach FMEA results become directly useful in the construction of behavioural models that can be used in the design and BSA of the system. This is a novel contribution of this thesis. In a typical industrial practice, FMEAs are employed at the end of design for certification purposes, while in IACoB FMEA becomes a **design tool** for assessing, and refining the behaviour of the system. In the later stages of development where more design information is available, abstract mode charts of IACoB can be refined to make reference to components and their behaviour. These mode charts can be enhanced by analysts to show detailed nominal and failure behaviour. Model checking performed on these models can be used to verify whether the specified system behaviour conform to safety requirements.

In Chapter 3, we have shown how close ties can be derived and maintained between abstract and refined mode charts. This subsequently improves traceability of relationships between failures, which is made possible via exploitation of the hierarchical mechanism of FTA/FMEA generation in HiP-HOPS.

Objective 3. To illustrate how a chosen CSA and a chosen BSA technique can in practice be harmonised in the context of a method for combined application. Different MBSA techniques assume different representations of failure information and system modelling. In the context of combined application, it is important to explore ways for translation of information (in particular, failure information) between relevant models. The thesis shows the integration of HiP-HOPS with NuSMV and defines a process for useful semi-automatic translation of information between the two models.

This thesis harmonised the representation of failure information between two different techniques that presently define the state-of-the-art in their respective areas. In the later part of Chapter 3, we described ways of translating information from HiP-HOPS into NuSMV. Failure behaviour in HiP-HOPS is captured within the failure annotation of components. We have shown that it is possible to preserve this information and incorporate it as part of the failure-relevant behaviour in a NuSMV model. HiP-HOPS

failure annotations typically contain information on the failure modes (output deviations) and their 'failure expression' which explains the failure causes (described in terms of internal malfunctions and input deviations). The translation process involves mapping this information into NuSMV variables and defining state transitions relating to these failures. In contrast to techniques like FSAP/NuSMV (Bozzano *et al.*, 2003b) which is largely based on success-logic, the relationship between failures here can be defined and managed as failure-logic. This means that the description of component output is determined by the condition of output deviations, and corresponding input deviations are assigned and passed accordingly.

In addition to harmonising and passing failure logic, the system hierarchies and propagation of failure effects can also be neatly captured and transferred from HiP-HOPS into the NuSMV model. We have also shown that this allows connections between mode charts to be systematically established, and this eventually enables a more-manageable refinement of transitions, and helps to guarantee consistency in the model as it evolves.

Chapter 4 and Chapter 5 presented case studies on automotive brake-by wire and aircraft wheel-brake system in which we demonstrated the value of combined iterative application of CSA and BSA to the design, how the IACoB process can be applied, and how analysis results from one technique can be exploited for the benefit of the other technique. These case studies ultimately show how the IACoB offers significant benefits over using only a single analysis approach.

Objective 4. The final research objective is to study the potential use of this approach in the design of mechanisms for detection and recovery from failures. More specifically, we propose a generic mechanism for modelling the *Detectability* (or NOT) of errors propagated among components of an architecture within a typical CSA. We show that the inclusion of this mechanism makes it possible to use the results of CSA as a basis for rational decisions about the inclusion of fault tolerant mechanisms in a design.

This thesis developed a novel concept for modelling the detectability of failures in CSA. The study of detectability in the context of CSA was presented in Chapter 6. We started the discussion with the concept of detectability in FMEA, where the detection

likelihood of failure modes is evaluated. We then explored further the use of detectability in system architecture and its role within fault tolerant design. Detectability in this context refers to the ability to correctly detect errors, and this ability is generally assigned to a component as part of its fault-detection and fault-tolerance mechanisms. A generic method to model detectability was subsequently proposed.

The method introduced failure-relevant parameters that model not only the internal malfunction of the component which performs detection, but also probabilities that errors are either correctly detected or go undetected by detection modules. The implication of these occurrences is modelled as part of the failure information in HiP-HOPS.

Because this concept is introduced as a part of CSA, we also studied how the general analysis on detectability can be performed. These parameters can be analysed using common fault tree analysis algorithms, with the exception of events where errors are detected (i.e. NOT missed by the detection module). The notion of events where errors are detected (and handled) leads to inclusion of NOT gates and a non-coherent fault tree structure. To enable the analysis of non-coherent fault trees, HiP-HOPS synthesis and analysis algorithms were extended, and a Consensus algorithm was implemented.

The use of non-coherent fault trees (and the inclusion of NOT gate for that matter) has been long debated. In this thesis, we presented a case in support of this argument and showed how the inclusion of NOT gate enables a more accurate modelling of detectability. For a practical demonstration of this, we presented a small case study on an automotive cruise control system and showed how detectability can be applied. Alternatives were also explored in an effort to maintain coherency of the fault trees and the analysis results were compared. From this, it was demonstrated that the inclusion of NOT gates have a significant role to play in system analysis, particularly in the correct quantification of system reliability.

7.2 Limitation of concepts

IACoB inherits the limitations of CSA and BSA. One challenge lies in the limited information on dynamic behaviour it initially captures. This is due to the fact that the initial failure information is directly obtained from a CSA-based technique, where focus

is placed on effectively capturing the failure propagation and hierarchy, rather than the dynamic behaviour. In certain circumstances, for example in the analysis of a phased mission system, more information on dynamic behaviour might be required than what is captured in CSA models. This can be addressed by independently extending the BSA model produced.

The extent of formal verification that can be performed largely depends on the level of information contained within the model. For a large complex system, model checking faces the challenge of state-space explosion, and this is a challenge that IACoB inherits from BSA. Abstraction techniques in model checking (Bérard, *et al.*, 2001) can be further investigated in the future to address this issue.

Another issue with IACoB is the fact that, currently, the integration process is mainly manual. This can be potentially labour-intensive and error-prone when performed repeatedly on a larger system. However, there is automation in both CSA and BSA, and it is also possible, to a large extent, automate the integration. Another related problem is the manual process of assigning failure expressions, which brings about the new kinds of manual errors compared to failure-injection methods. Recent development towards a language for describing failure patterns (Wolforth, 2010) is one way to improve the process. There is also the lack of support for specifying requirement properties. Errors are common during the conversion of safety properties from natural language to CTL. We believe it can be beneficial to develop tool support that can assist this process.

Challenges which are related to the nature of manual processing can be potentially resolved with automation. We hope supporting tools can be developed in the future to ease the task of conversion between models, and improve the process of storing and retrieving failure information and safety specifications.

IACoB is also mainly performed to assess and verify the design, interaction of functions or components, and control logic of an early design model. Therefore, another limitation of IACoB is that it does not address errors that arise later on in the development lifecycle (for example, coding and implementation errors).

One main problem of the detectability concept is the additional computational expense introduced by the analysis of computationally-extensive non-coherent fault tree analysis algorithm.

7.3 Future Work

The aim of establishing a framework to allow combined application of CSA and BSA has been achieved to a certain extent. The results of this work on the IACoB approach provide the foundation for potential future work in the following directions:

1. Improvement on modelling experience

In addition to the guidelines on translation of failure information between HiP-HOPS and NuSMV models outlined in Chapter 3, we believe it is beneficial to develop an automated translation support tool. Although the manual construction of the NuSMV model is manageable for smaller systems explored in this thesis, an automated translator would ease the process and increase its scalability. Behavioural information (for example, description of model states) which is not included within HiP-HOPS annotations can be obtained by extending the failure editor.

Support for graphical representation of sate machines can also be introduced to assist behavioural modelling. Various translator tools like sf2smv (Banphawatthanarak *et al.*, 1999), stm2smv (Loer, 2003), or mdl2smv (Juarez-Dominguez *et al.*, 2008) have been developed to convert commercial graphical behavioural tool like Stateflow or Statemate into SMV model. We believe that similar capabilities for the HiP-HOPS failure editor might be beneficial in making the behavioural modelling process more intuitive. Alternatively, future work that looks into the integration of these established graphical tools with HiP-HOPS failure editor could be investigated.

2. Improvement on integration with nominal behavioural model

The NuSMV models produced from the HiP-HOPS annotated models are essentially a formal functional ‘error-model’. Although the extension to include description of nominal behaviour can be relatively straight forward, we believe a degree of automation in this process will be helpful. This is particularly useful if the formal nominal behavioural model is developed in parallel with the HiP-HOPS model. One possible way to achieve this is by enforcing common references to states and events in the two state machines that describe nominal and failure behaviour, and then by automatically parsing and combination of both models into a single combined representation. Further research can be done in this aspect as it is currently a manual process.

3. Support for formal requirement properties and visualization of traces

The translation of requirements from natural language to temporal-logic formulae is not a trivial process. With the reference of general classification of properties from (Bérard, *et al.*, 2001), it is beneficial to have a set of generic templates to define the property specifications. This template should allow frequently-used property patterns to be saved into pattern library, and instantiated whenever needed. Comprehensive review of property specification patterns and their hierarchy was presented in (Dwyer *et al.*, 1999), and example of this specification pattern support is presented in IFADIS (Loer, 2003).

Traces are usually produced to show counterexamples. A trace is a sequence of execution steps that leads from system initial state to the state that violates safety properties. Each step in between describes value changes in the variables. These traces produced from NuSMV counterexamples are in textual form. A graphical viewer for these traces, for example traces chart illustrated in (Peikenkamp, 2006), can provide a more intuitive outlet for display and analysis.

4. Failure modes completeness and harmonisation

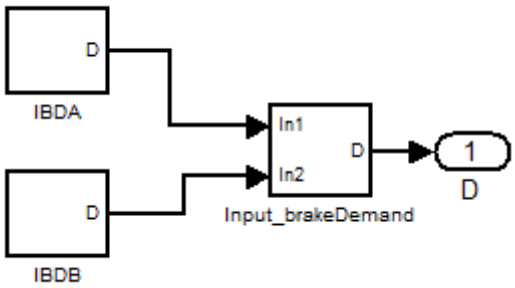
The concept of failure modes is central to both CSA and BSA approaches. In IACoB, we employ generalized failure modes which belong to four categories: omission, commission, value or timing failures. These are reflected in the structural as well as behavioural models. In occasions where we need to derive an explicit list of failure modes and validate its completeness, (Ortmeier, 2004a) describes a method that uses failure-sensitive specifications. This method defines an initial chaotic model which describes all possible combinations between inputs and outputs. It then extracts the combinations which violate specification rules, which are made into a list of failure modes, and eliminates ones that are not relevant. The remaining 'good' combinations (those that conform to the specification rules) are validated against the nominal model, which is constructed separately. Lastly this model is integrated again with the failure modes to form the 'error model'. This technique yields the benefits of being able to generate a more complete specification of failure modes and validation of the nominal formal model. However it does suffer from the exponential size of the sets used. Therefore, one area that we could look into in the future is the potential of applying this

approach as part of IACoB for selected critical components, as opposed to the whole system.

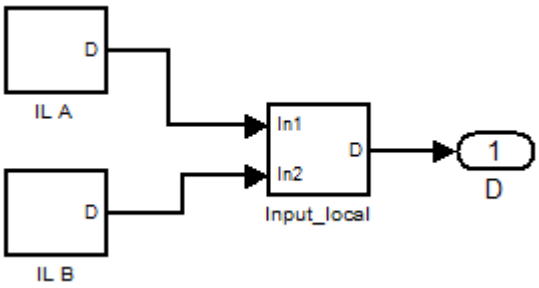
APPENDIX A: Backup structure for brake-by-wire system

The following figures show the backup structure scheme for brake-by-wire system presented in Chapter 4.

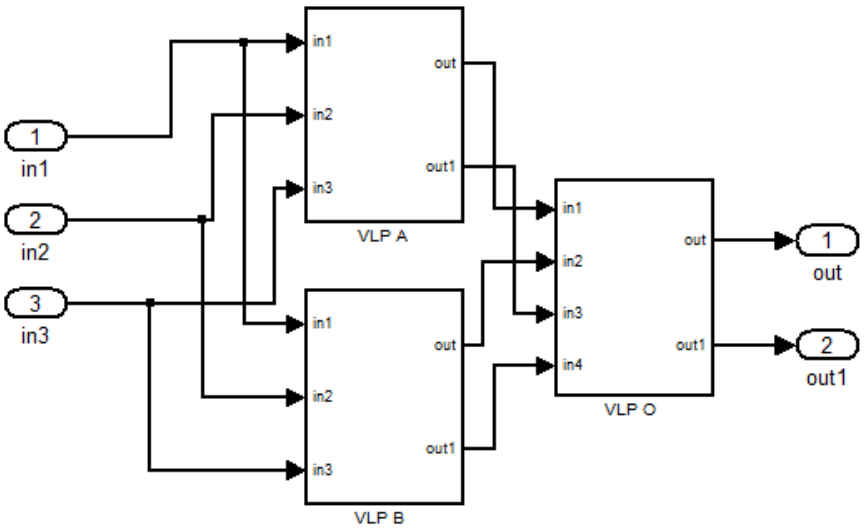
A.1. Brake Demand Input Function



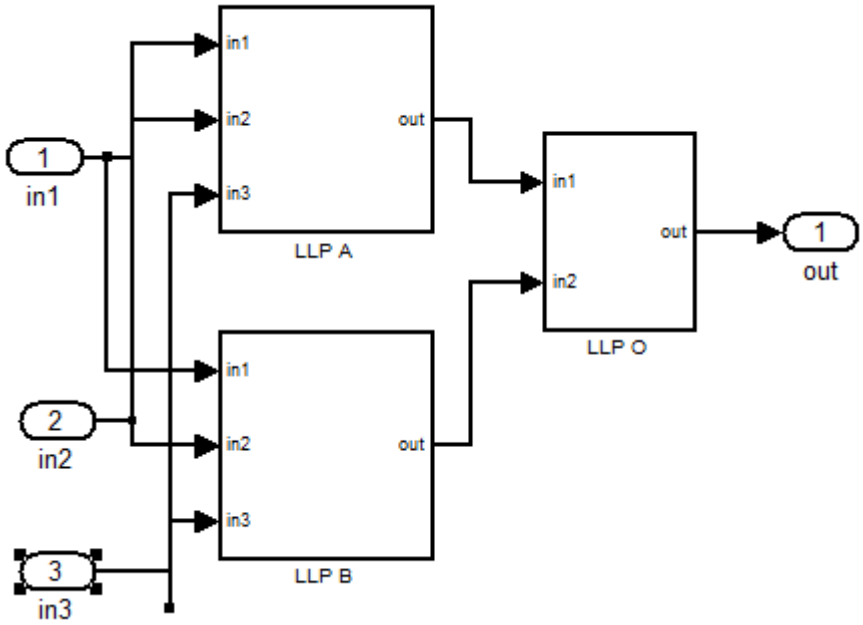
A.2. Local Parameters Input Function



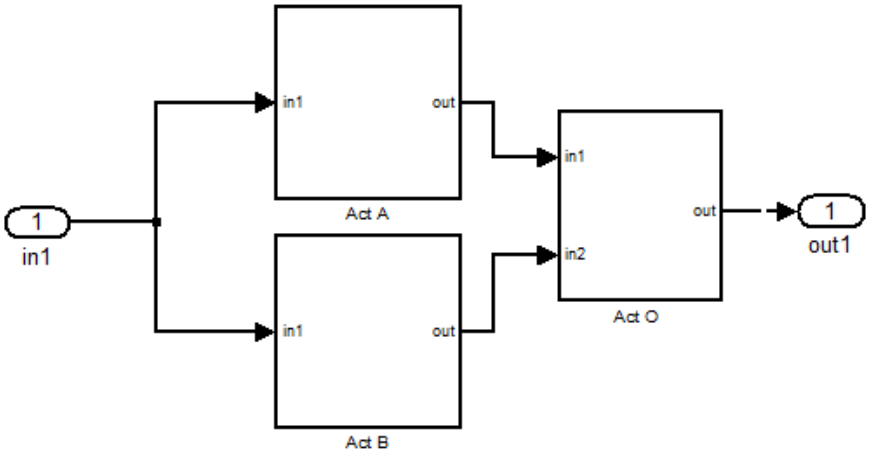
A.3. Vehicle Level Processing Function



A.4. Local Level Processing Function



A.5. Braking Energy Function



APPENDIX B: NuSMV model for brake-by-wire

B.1. The following shows an example of how the abstract mode chart for brake-by-wire system can be represented in NuSMV model:

```
MODULE ABS
VAR
O-Output: boolean;
Output: boolean;
counter: 0..1;

ASSIGN
init(O-Output) := 0;
Output := !O-Output;

counter :=case
O-Output : 1;
1: 0;
esac;

next(O-Output):=
case
O-Output = 1: 1;
1: {1,0};
esac;

-----------------------------------------------

MODULE ELEC

VAR
O-Output: boolean;
Output: boolean;
counter: 0..1;

ASSIGN
init(O-Output) := 0;
Output := !O-Output;

counter :=case
O-Output : 1;
1: 0;
esac;

next(O-Output):=
case
O-Output = 1: 1;
1: {1,0};
esac;

-----------------------------------------------

MODULE HYDRAULIC

VAR
O-Output: boolean;
```

```
Output: boolean;
counter: 0..1;

ASSIGN
init(O-Output) := 0;
Output := !O-Output;

counter :=case
O-Output : 1;
1: 0;
esac;

next(O-Output):=
case
O-Output = 1: 1;
1: {1,0};
esac;

-----------------------------------------------

MODULE main

VAR
SystemMode: {BBW_Normal, BBW_PD1, BBW_PD2, BBW_Fail};
counter: 0..3;

absB: ABS;
elec: ELEC;
hydraulic : HYDRAULIC;

ASSIGN

init(SystemMode) := BBW_Normal;
next(SystemMode):=case
SystemMode = BBW_Normal & elec.O-Output = 1 : BBW_PD2;
SystemMode = BBW_Normal & absB.O-Output = 1 : BBW_PD1;
SystemMode = BBW_Normal & elec.O-Output = 1 & hydraulic.O-Output = 1 :
BBW_Fail;
SystemMode = BBW_PD1 & elec.O-Output = 1 & hydraulic.O-Output = 1 :
BBW_Fail;
SystemMode = BBW_PD1 & elec.O-Output = 1 : BBW_PD2;
SystemMode = BBW_PD2 & hydraulic.O-Output = 1 : BBW_Fail;
1: SystemMode;
esac;

counter := absB.counter + elec.counter + hydraulic.counter;
```

B.2.The following shows an example of how the refined mode chart can be constructed for brake-by-wire system:

```
MODULE FL(O-ABScmd, FLDiagonalLock)
VAR
States : {Normal, Locked} ;
LockBE : boolean;
counter : 0..1;
FLlockSig : boolean;
locked: boolean;

ASSIGN
init(States) := Normal;
init(LockBE) := 0;

locked := (LockBE & O-ABScmd)| FLDiagonalLock;
FLlockSig := case
States = Normal: 0;
1: 1;
esac;

next(States):=case
States = Normal & locked = 1: Locked;
1: States;
esac;

next(LockBE) := case
LockBE = 1 : 1;
1: {0,1} ;
esac;

counter:=case
States = Locked : 1;
1: 0;
esac;

-------------------------------------
MODULE FR(O-ABScmd, FRDiagonalLock)
VAR
States : {Normal, Locked} ;
LockBE : boolean;
counter : 0..1;
FRlockSig : boolean;
locked: boolean;

ASSIGN
init(States) := Normal;
init(LockBE) := 0;

locked := (LockBE & O-ABScmd)| FRDiagonalLock;
FRlockSig := case
States = Normal : 0;
1: 1;
esac;

next(States):=case
States = Normal & locked = 1 : Locked;
1: States;
```

```
esac;

next(LockBE) := case
LockBE = 1 : 1;
1: {0,1} ;
esac;

counter:=case
States = Locked : 1;
1: 0;
esac;

-----------------------------------
MODULE RL(O-ABScmd, RLDiagonalLock)
VAR
States : {Normal, Locked} ;
LockBE : boolean;
counter : 0..1;
RLlockSig : boolean;
locked: boolean;

ASSIGN
init(States) := Normal;
init(LockBE) := 0;

locked := (LockBE & O-ABScmd)| RLDiagonalLock;
RLlockSig := case
States = Normal : 0;
1: 1;
esac;

next(States):=case
States = Normal & locked = 1 : Locked;
1: States;
esac;

next(LockBE) := case
LockBE = 1 : 1;
1: {0,1} ;
esac;

counter:=case
States = Locked : 1;
1: 0;
esac;
-----------------------------------

MODULE RR(O-ABScmd, RRDiagonalLock)
VAR
States : {Normal, Locked} ;
LockBE : boolean;
counter : 0..1;
RRlockSig : boolean;
locked: boolean;


ASSIGN
init(States) := Normal;
init(LockBE) := 0;
```

```
locked := (LockBE & O-ABScmd)| RRDiagonalLock;

RRlockSig := case
States = Normal : 0;
1: 1;
esac;

next(States):=case
States = Normal & locked = 1 : Locked;
1: States;
esac;

next(LockBE) := case
LockBE = 1 : 1;
1: {0,1} ;
esac;

counter:=case
States = Locked : 1;
1: 0;
esac;
-----------------------------------

MODULE ABS (O-ECUabs)
VAR
States : {Normal, Fail} ;
O-ABScmd: boolean;

ASSIGN
init(States) := Normal;

next(States):=case
O-ECUabs = 1 : Fail;
1: Normal;
esac;

O-ABScmd := case
States = Normal : 0;
1: 1;
esac;

-----------------------------------

MODULE ECU (O-WS, O-ES)
VAR
States : {Normal, Fail} ;
O-ECUabs: boolean;
ECUABEabs: boolean;
ECUABEabsC: boolean;
ECUBBEabs: boolean;
ECUBBEabsC: boolean;

ASSIGN
init(States) := Normal;

next(States):=case
O-WS | O-ES | (ECUABEabsC & ECUBBEabsC) | (ECUABEabsC & ECUBBEabs) |
(ECUABEabs & ECUBBEabsC) | (ECUABEabs & ECUBBEabs)  : Fail;
1: Normal;
esac;
```

```
O-ECUabs := case
States = Normal : 0;
1: 1;
esac;

next(ECUABEabs):=case
ECUABEabs = 1 : 1;
1: {1,0};
esac;

next(ECUABEabsC):=case
ECUABEabsC = 1 : 1;
1: {1,0};
esac;

next(ECUBBEabs):=case
ECUBBEabs = 1 : 1;
1: {1,0};
esac;

next(ECUBBEabsC):=case
ECUBBEabsC = 1 : 1;
1: {1,0};
esac;

-------------------------------------


MODULE WS
VAR
States : {Normal, Fail} ;
WSBE : boolean;
O-WS : boolean;

ASSIGN
init(States) := Normal;
init(WSBE) := 0;

next(WSBE) :=case
WSBE = 1 : 1;
1: {1,0};
esac;

next(States):=case
WSBE  : Fail;
1: Normal;
esac;

O-WS:= case
States = Normal : 0;
1: 1;
esac;


-------------------------------------

MODULE ES
VAR
States : {Normal, Fail} ;
ESBE : boolean;
O-ES : boolean;
```

```
ASSIGN
init(States) := Normal;
init (ESBE) := 0;

next(ESBE) :=case
ESBE = 1 : 1;
1: {1,0};
esac;

next(States):=case
ESBE  : Fail;
1: Normal;
esac;

O-ES:= case
States = Normal : 0;
1: 1;
esac;


-------------------------------------
MODULE main

VAR
States: {Normal,TD1_Critical_FR,TD1_Critical_RL,
TD1_Critical_FL, TD1_Critical_RR, PD2_FR-RLDiagonalLock,
PD2_FL-RRDiagonalLock, TD3_Critical_FRRLFL,TD3_Critical_FRRLRR,
TD3_Critical_FLRRFR,TD3_Critical_FLRRRL,PD4_AllWheelsLocked};
counter: 0..4;
FLdiagonalLock: boolean;
FRdiagonalLock: boolean;
RLdiagonalLock: boolean;
RRdiagonalLock: boolean;
O-ABScmd : boolean;
FRlockSig : boolean;
RLlockSig : boolean;
RRlockSig : boolean;
FLlockSig : boolean;

ALLOFF: boolean;
DLActive: boolean;
Hazardous : boolean;
TwoParallelWheelsLocked : boolean;


flw : FL(O-ABScmd, FLdiagonalLock);
frw : FR(O-ABScmd, FRdiagonalLock);
rlw : RL(O-ABScmd, RLdiagonalLock);
rrw : RR(O-ABScmd, RRdiagonalLock);
ws: WS;
es: ES;
ecu : ECU (ws.O-WS, es.O-ES);
abs : ABS(ecu.O-ECUabs);

ASSIGN
init(States) := Normal;
O-ABScmd := abs.O-ABScmd;

counter := flw.counter + frw.counter + rlw.counter + rrw.counter ;
FRlockSig := frw.FRlockSig;
```

```
RLlockSig := rlw.RLlockSig;
RRlockSig := rrw.RRlockSig;
FLlockSig := flw.FLlockSig;
init (TwoParallelWheelsLocked) := 0;


FLdiagonalLock:= case
States = TD1_Critical_RR | States = TD3_Critical_FRRLRR : 1;
1: 0;
esac;

FRdiagonalLock:= case
States = TD1_Critical_RL | States = TD3_Critical_FLRRRL: 1;
1: 0;
esac;

RRdiagonalLock:= case
States = TD1_Critical_FL | States = TD3_Critical_FRRLFL : 1;
1: 0;
esac;

RLdiagonalLock:= case
States = TD1_Critical_FR | States = TD3_Critical_FLRRFR : 1;
1: 0;
esac;

ALLOFF := !FLdiagonalLock &  !RLdiagonalLock &  !FRdiagonalLock  &
!RRdiagonalLock;
DLActive := FLdiagonalLock |  RLdiagonalLock |  FRdiagonalLock  |
RRdiagonalLock;
Hazardous := case
States  =  TD1_Critical_FR  |States  =  TD1_Critical_RL  |  States  =
TD1_Critical_FL    |    States    =    TD1_Critical_RR    |States    =
TD3_Critical_FRRLRR|States   =     TD3_Critical_FLRRFR   |   States   =
TD3_Critical_FLRRRL : 1;
1: 0;
esac;

next(TwoParallelWheelsLocked):=case
TwoParallelWheelsLocked = 1: 1;
1 : ((flw.States = Locked & rlw.States = Locked)|( frw.States = Locked
& rrw.States = Locked) | (flw.States = Locked & frw.States = Locked) |
(rlw.States = Locked & rrw.States = Locked)) & counter = 2;
esac;

next(States) := case
States = Normal & FRlockSig : TD1_Critical_FR;
States = Normal & RLlockSig : TD1_Critical_RL;
States = Normal & FLlockSig : TD1_Critical_FL;
States = Normal & RRlockSig : TD1_Critical_RR;
States = TD1_Critical_FR & RLlockSig : PD2_FR-RLDiagonalLock;
States = TD1_Critical_RL & FRlockSig : PD2_FR-RLDiagonalLock;
States = TD1_Critical_FL & RRlockSig : PD2_FL-RRDiagonalLock;
States = TD1_Critical_RR & FLlockSig : PD2_FL-RRDiagonalLock;
States = PD2_FR-RLDiagonalLock & FLlockSig : TD3_Critical_FRRLFL;
States = PD2_FR-RLDiagonalLock & RRlockSig : TD3_Critical_FRRLRR;
States = PD2_FL-RRDiagonalLock & FRlockSig : TD3_Critical_FLRRFR;
States = PD2_FL-RRDiagonalLock & RLlockSig : TD3_Critical_FLRRRL;
States = TD3_Critical_FRRLFL | States = TD3_Critical_FRRLRR | States =
TD3_Critical_FLRRFR    |    States    =    TD3_Critical_FLRRRL    :
PD4_AllWheelsLocked;
```

```
1: States;
esac;
```

APPENDIX C: Summary of Quantitative Analysis

The effects of detectability parameters on system probability for top event C-Actuator_Module.Out:

Cruise Control System without activation of any detection module:

Minimal Cut sets Produced	**Probability**
11	0.064

Cruise Control System with activated detection module:

Detectability Parameters	**Minimal Cut sets Produced**	**Probability**
Internal Malfunction *Failure*	6	0.036
Internal Malfunction Failure, Event *Miss*, ¬ *Miss*	11	0.039

Detectability Parameters	**Prime Implicants Produced**	**Probability**
Event ¬ *Miss*	11	0.039
Event *Catch*	11	0.039

Cruise Control System with Fading Brake for top event C-Actuator_Module.Out:

Detectability Parameters	**Prime Implicants Produced**	**Probability**
Event ¬ *Miss*	11	0.039
Event *Catch*	11	0.039

Cruise Control System with Fading Brake for top event O-Actuator_Module.Out:

Detectability Parameters	**Prime Implicants Produced**	**Probability**
Event ¬ *Miss*	19	0.072
Event *Catch*	19	0.064

Appendix D: Prime Implicants for Cruise Control System

Prime implicant results for O-Actuator_Module (Cruise Control System with Fading Brake) with the use of ¬ *Miss:*

13 x Cut Sets of Order	**Probability**
Monitoring_Module.BE1	0.00458944
Logic_Module.BE1	0.00458944
Cruise_Control.BE1	0.00458944
Brake_Support.BE1	0.00458944
Actuator_Module.BE1	0.00458944
Switch_Sensor.BE	0.00114934
Speed_Sensor.BE	0.00114934
Radar_Sensor.BE	0.00114934
Pedal_Sensor.BE	0.00114934
Memory.BE	0.00114934
DM_LM.failure	0.00111937
DM_FB.failure	0.00111937
DM_CC.failure	0.00111937
5 x Cut Sets of Order	**Probability**
Monitoring_Module.BE2 DM_CC.NOT_miss	0.00980101
Monitoring_Module.BE2 DM_LM.NOT_miss	0.00980101

Brake_Support.BE2 DM_CC.NOT_miss	0.00980101
Fading_Brake.BE1 DM_CC.NOT_miss	0.0040387
DM_FB.miss Fading_Brake.BE1	0.000550732
1 x Cut Sets of Order	**Probability**
Fading_Brake.BE2 DM_CC.NOT_miss DM_FB.NOT_miss	0.00862489

Prime implicant results for O-Actuator_Module (Cruise Control System with Fading Brake) with the use of *Catch:*

13 x Cut Sets of Order	**Probability**
Monitoring_Module.BE1	0.00458944
Logic_Module.BE1	0.00458944
Cruise_Control.BE1	0.00458944
Brake_Support.BE1	0.00458944
Actuator_Module.BE1	0.00458944
Switch_Sensor.BE	0.00114934
Speed_Sensor.BE	0.00114934
Radar_Sensor.BE	0.00114934
Pedal_Sensor.BE	0.00114934
Memory.BE	0.00114934
DM_LM.failure	0.00111937

DM_FB.failure	0.00111937
DM_CC.failure	0.00111937
4 x Cut Sets of Order	**Probability**
DM_LM.catch Monitoring_Module.BE2	0.00980101
Brake_Support.BE2 DM_CC.catch	0.00980101
DM_CC.catch Fading_Brake.BE1	0.0040387
DM_FB.miss Fading_Brake.BE1	0.000550732
2 x Cut Sets of Order	**Probability**
DM_CC.catch DM_FB.catch Fading_Brake.BE2	0.00862489
DM_CC.catch DM_LM.miss Monitoring_Module.BE2	0.00117612

APPENDIX E: List of Abbreviation

ABS	Anti-lock Brake System
BBW	Brake-by-wire
BDD	Binary Decision Diagram
BSA	Behavioural Safety Analysis
BSCU	Brake System Control Unit
CEG	Cause Effect Graphs
CFT	Component Fault Trees
CSA	Compositional Safety Analysis
CTL	Computational Tree Logic
DSPN	Deterministic and Stochastic Petri Nets
EHB	Electrical Hydraulic Brake
EMB	Electrical Mechanical Brake
ESP	Electronic Stability Program
ESSaReL	Embedded Systems Safety and Reliability Analyser
FFA	Functional Failure Analysis
FFBD	Functional Flow Block Diagram
FHA	Functional Hazard Assessment
FMEA	Failure Modes and Effects Analysis
ForMoSA	Formal Methods and Safety Analysis
FSAP/NuSMV	Formal Safety Analysis Platform/NuSMV

FSM	Finite State Machine
FTA	Fault Tree Analysis
HAZOP	Hazard and Operability Study
HiP-HOPS	Hierarchically Performed Hazards Origin and Propagation Studies
IACoB	Integrated Application of Compositional and Behavioural
IF-FMEA	Interface Focussed FMEA
LTL	Linear Temporal Logic
MBSA	Model-based Safety Analysis
MICSUP	Minimal Cut Set UPward
NuSMV	New Symbolic Model Verifier
PLTL	Propositional LTL
PSSA	Preliminary Safety Assessment
RPN	Risk Priority Number
SEFT	State-Event Fault Trees
SMV	Symbolic Model Verifier
TTP	Time-triggered Communication Protocol
WBS	Wheel-brake system

REFERENCES

Adachi, M., Papadopoulos, Y., Sharvia, S., Parker, D., Tohdo, T., 2010. An approach to optimization of fault tolerant architectures using HiP-HOPS. *Software Practice and Experience*. Wiley Interscience. DOI: 10.1002/spe.1044.

Arnold, A., Begay, D., Crubille, P., 1994. *Construction and analysis of transition systems with MEC. World Scientific*: Singapore. ISBN: 981-02-1922-9.

Arnold, A., Gerald, P., Griffault, A., Rauzy.A., 2000. The Altarica formalism for describing concurrent systems . *Fundamenta Informaticae*, 34, pp 109-124.

ARP 4754, 1994. *Aerospace recommended practice: Certification considerations for highly-integrated or complex aircraft system*s . Society of Automotive Engineering. Warrendale, PA: SAE.

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: SAE.

Autopressnews, 2006. *All new Volvo S80 safety systems*. Available through: Autopressnews < http://www.autopressnews.com/2006/2006csm/m03/volvo/volvo_s80_safety_systems.shtml> [Accessed at: 11/12/2010].

Banphawatthanarak, C., Krogh, B.H., 1999. *Verification of Stateflow diagrams using SMV: sf2smv 2.0*.Pittsburgh: Carnegie Mellon University.

Barfield, L., 2004. *The user interface concept and design*. Addison Wesley, pp 43- 72. ISBN: 0954723902, 9780954723903.

Berard, B. Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci., L., Schnoebelen, McKenzie, P.,1999. *System and software verification: model-checking techniques and tools*. Germany: Springer-Verlarg. ISBN: 3540415238, 9783540415237.

Bieber, P., Bougnol, C., Castel, C., Heckmann, J., Kehren, C., Metge, S., Seguin, C., 2004. Safety assessment with Altarica – Lessons learnt based on two aircraft case studies, in *Proceedings of 18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification,* pp 505-510.

Bieber, P., Castel, C., 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, Lecture Notes in Computer Science, 2485,* pp 624-628.

Blanchard, B.S., 1998. *System engineering management*. New York: John Willey&Son. ISBN: 0471291765, 9780471291763.

Bobbie, O., Buggineni, V., JI, Y.M., 2001. Model checking with sf2smv/SMV and simulation of parallel systems, in *Proceedings of the Hunstville Simulation Conference (Society of Computer Simulation – SCS),* pp 159-166. Hunstville, USA.

Bondavalli, A., Simoncini, L., 1990. *Failure classification with respect to detection. Esprit Project Nr 3092 (PDCS: Predictably Dependable Computing Systems*).

Bozzano, M. *et al*, 2003b. ESACS: an integrated methodology for design and safety analysis of complex systems, in *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pp 237-245. Springer.

Bozzano, M., Cavallo, A., Cifaldi, M., 2003a. Improving safety assessment of complex systems: an industrial case study. *Lecture Notes in Computer Science*, 2805, pp 208-222.

Bozzano, M., Villafiorita, A., 2006. The FSAP/NuSMV-SA safety analysis platform. *International Journal on Software Tools for Technology Transfer (STTT)*, 9, pp 5-24.

Carley, L., 2004. Brake-by-wire. *Brake&Front End Magazine*. Available through: http://www.brakeandfrontend.com/.

Cavada, R. *et al.*, 2005. *NuSMV 2.3 Manual*. Italy: IRST. Available through: http://nusmv.fbk.eu/NuSMV/userman/v23/nusmv.pdf. Accessed at : [10/12/2010].

Ciardo, G., Lindermann, C., 1993. Analysis of deterministic and stochastic Petri nets, in *Proceedings of the 5th International Workshop on Petri nets and Performance models* (*PNPM 1993*), pp 160 – 169.

Cimatti, A. *et al*., 1998. Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing*, 10, pp 361-380.

Cimatti, A., Clarke, E., Giuchiglia, F., Roveri, M., 1999. NuSMV: a new symbolic model verifier, in *Proceedings of Computer-aided Verification (CAV'99). Lecture Notes in Computer Science*, 1633, pp 495-499. Springer-Verlag, London.

Clarke, E.M., Emerson, A., 1980. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, pp 169-181. Springer-Verlag, Lodon.

Clarke, E.M., Wing, J.M., 1996. Formal Methods: state of the art and future directions.*ACM Computer Surveys*, 28(4), pp 626-643.

Colombo, D., 2007. *Brake-by-wire system development: Technology and development process.* FIAT Group, Torino. Available through: <http://staff.polito.it/enrico.canuto/Home_page/pdf/Incontro18gen2008/DColombo.pdf > . Accessed at: [16/02/2011].

Davey, C., Friedman, J., 2007. Software systems engineering with model-based design, in *Proceedings of 4th International Workshop on Software Engineering for Automotive Systems (SEAS'07)*, pp 7. IEEE. DOI: 10.1109/SEAS.2007.9.

Drusinksy, D., 2006. *Modeling and verification using UML Statecharts*. Oxford: Newnes. ISBN: 0750679492, 9780750679497.

Dwyer, M.B., Avrunin, G.S. Corbett, J.C., 1999. Patterns in property specifications for finite-state verification. In GARLAN, D., and KRMAER, J., eds., *21st International Conference on Software Engineering*, Lost Angeles, California, pp 411-420.

FAA, 2006. *NAS System Engineering Manual*. Available through: Federal Aviation Administration <

http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/operations/sysengsaf/seman/SEM3.1/Section%204.4.pdf> [Accessed at: 10/12/2010].

Faller, R., 2009. *Importance of Functional Safety and IEC 61508*. Exida. Available through: < http://www2.dke.de/de/Wirueberuns/MitteilungenderDKEGeschaeftstelle/documents/vde%20dke%20-%20tagung%20zur%20iec%2061508%20_%20pr%C3%A4sentationen/importance%20of%20functional%20safety%20and%20iec%2061508.pdf> [Accessed at: 14/02/2011].

Ford, 2010. *Adaptive cruise control and collision warning with brake support*. Available through : Ford Media < http://media.ford.com/images/10031/Adaptive_Cruise.pdf> [Accessed at: 09/12/2010].

German, R., Mitzlaff, J., 1995. Transient analysis of deterministic and stochastic Petri nets with TimeNET, *Proceedings of the 8th International Conference on Computer Performance Evaluation, Modelling Techniques, and Tools and MMB* (*Lecture Notes in Computer Science*, vol. 977), pp 209-223.

Grunske, L., Kaiser, B., Papadopoulos, Y., 2005. Model-driven safety evaluation with state-event-based component failure annotation. *Lecture notes in computer science*, 3489, pp 33-48.

Hamann, R., Uhlig, A., Papadopoulos, Y., Rüde,E., Grätzu., Lien, R., 2008. Derivation of Ship System Safety Criteria by means of Risk-Based Ship System Safety Analysis, *ASME 27th International Conference on Offshore Mechanics and Arctic Engineering* (OMAE'08), Estoril, Portugal. Proceedings on CD.

Harel, D., 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8, pp 231-274.

Harel, D., Namad, A., 1996. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), pp 293-333.

Hedenetz, B., Belschner, R., 1998. *Brake-by-wire without mechanical backup using a TTP-communication network*. Society of automotive engineering International

Congress, SAE 981109. Available through: < http://www.vmars.tuwien.ac.at/projects/xbywire/projects/new-BBW.html>. Accessed at: [21/02/2011].

Heimdahl, M.P.E., 2007. *Formal model-based development in aerospace systems: challenges to adoption, Lecture notes*. University of Minnesota.

Holzmann, G., Joshi, R., Groce, A., *2005. New challenges in model checking*. *Lecture Notes in Computer Science*, 5000. DOI: 10.1007/978-3-540-69850-0_4

Huth, M., Ryan, M., 2000. *Logic in computer science – modelling and reasoning about systems*. Cambridge University Press. ISBN: 052154310.

IEC 61508, 1998. *Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission. Geneva.

Isermann, I., 2004. Fault-tolerant ddrive-by-wire systems. Available through: < http://www.apca.pt/~apca_docs/CONTROLO2004/controlo2004/papers/pdf0098.pdf>. Accessed at: [10/12/2010].

Isermann, R., 2002. Fault-tolerant drive-by-wire systems. *Control Systems Magazine.* IEEE, 5(2), pp 64-81.

Isermann, R., 2004. Model-based fault detection and diagnosis: status and application, in *Proceedings of the 16th IFAC Symposium on Automatic Control in Aerospace*, pp 71-85. DOI:10.1016/j.arcontrol.2004.12.002.

Johannessen, P., Grante, C., Alminger, C., Eklund, U., 2001. Hazard analysis in object oriented design of dependable systems. *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01),* pp 507-512. DOI: 10.1109/DSN.2001.941436.

Johnston, B.D., Matthews, R.H., 1993. *Non-coherent structure theory: a review and its role in fault tree analysis*. United Kingdom: UK Atomic Energy Authority.

Jones, P.M., Mitchell, C.M., 1987. Operator modelling: conceptual and methodological distinctions, in *Proceeddings of the 31st Annual Meeting of the Human Factors Society*, 1, pp 31-35.

Joshi, A. Heimdahl, M., Miller, S,. Whalen, M., 2006. *Model-Based safety analysis*. University of Minnesota. Advanced Technology Center.

Joshi, A. Heimdahl, M.P.E, 2005. Model-based safety analysis of Simulink models using SCADE design verifier, in *Proceedings of the 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pp 122-135. Springer.

Juarez-Domiguez, A.L., Nancy, A.D., Joyce, J.J., 2008. Modelling feature interactions in the automotive domain, in *Proceedings of the 2008 International Workshop on Models in Software Engineering (MiSE'08)*, pp 45-50. DOI: 10.1145/1370731.1370743.

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

Kaiser, B., Gramlich, C., 2004. State-Event-Fault-Trees A safety Analysis Model for Software Controlled Systems. *Computer Safety, Reliability, and security*, 3219, pp 195-209.

Kaiser, B., Liggesmeyer, P., Mackel, O., 2003. *A New Component Concept for Fault Trees*. SCS'03 Australia.

Katoen, J.P., 2002. Principles of model checking. *Lecture Notes in Computer Science*, pp 16-35. University of Twente.

Kehren, C., Sequin, C., Bieber, P., Castel, C., Bougnol,C., Heckmann. J-P., Metge, S., 2000. Safety Assessment with AltaRica. *IFIPInternational Federation of Information Processing*, 156, pp 505-510.

Kletz, T. A., 1997., HAZOP – Past and Future, *Reliability Engineering and System Safety*, **55**(3), pp 263-266.

Knight, J.C., 2002. Safety critical systems: challenges and directions (summary of state-of-the-art presentation), *in Proceedings of International Conference on Software Engineering*, pp 547-550. Orlando, Florida.

Kripke, S.A., 1963. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16, pp 83-94.

Langenwalter, J., Erkkinen, T., 2004. Embedded steer-by-wire system development. *Embedded World*. Germany: MathWorks.

Lisagor, O. McDermid, A.J., Pumfrey, D.J., 2006. Towards a practicable process of automated safety analysis, *in Proceedings of the 24th International System Safety Conference (ISSC)*. Albuquerque, New Mexico, USA.

Lisagor, O., McDermid, J.A., Pumrey, D., 2003. *Safety analysis of software architecture – lightweight PSSA*. University of York.

Loer, K., 2003. *Model-based automated analysis for dependable interactive systems*. Computer Science Department, University of York. UK.

McMillan, K.L., 1993. *Symbolic model checking*. Kluwer. ISBN:0792393805.

Miller, S.P., Tribble, A., Whalen, M., Heimdahl, M., 2003A. Proving the shalls: early validation of requirements through formal models. *International Journal on Software Tools for Technology Transfer (STTT), 8(4),* pp 303-319. Springer-Verlag.

NASA, 1995. *NASA systems engineering handbook*. SP-610S. Available through: MIT < http://snebulos.mit.edu/projects/reference/NASA-Generic/NASA-STD-8739-8.pdf> [Accessed: 10/12/2010].

NASA, 2007. *NASA system engineering handbook*. Washington DC. Available through: NASA STI < http://education.ksc.nasa.gov/esmdspacegrant/Documents/NASA%20SP-2007-6105%20Rev%201%20Final%2031Dec2007.pdf> [Accessed: 10/12/2010].

Nossal, R., Lang, R., 2002. Model-based system development: An approach to building x-by-wire application. *IEEE Micro*, 22(4), pp 56-63.

Ortmeier, F , Thums, A., Schellhorn, G., Reif, W., 2004b. Combining formal methods and safety analysis – the ForMoSA approach, in *Proceedings of Softspez Final Report*, pp 474-493.

Ortmeier, F., Reif, W., 2004a. *Failure-sensitive specification: a formal method for finding failure modes.* Institut fur Informatik, Augsburg.

Pagetti, C., Cassezf., Roux, O., 2003. Hierarchical modelling and verification of timed systems in timed AltaRica, *in Proceedings of FACS the First Workshop on Formal Aspects of Component Software*, pp 63-80.

Palshikar, G.V., 2004. *An Introduction to Model Checking*. Available through : < www.embedded.com>. Accessed at: [12/11/2010].

Papadopoulos, Y., 2003. Model-Based System Monitoring and Diagnosis of Failures using Statecharts and Fault Trees. *Reliability Engineering and System Safety*, 81, pp 325-341.

Papadopoulos, Y., 1998. Safety analysis of a distributed brake by wire system for cars. *ESPRIT 23396 (TTA) Deliverable*. University of York.

Papadopoulos, Y., 2000. *Safety-directed system monitoring using safety cases*. PhD Thesis. University of York.

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

Papadopoulos, Y., Maruhn, M., 2001. Model-Based automated synthesis of fault trees from Matlab-Simulink models. *International Conference on Dependable Systems and Networks*, pp 77-82.

Papadopoulos, Y., McDermid, J.A., 1999. The potential for a generic approach to certification of safety-critical systems in the transportation sector. *Reliability Engineering and System Safety*, 63(1), pp 47-66. Elsevier Science.

Papadopoulos, Y., McDermid, J.A., SASSE, R., HEINER, G., 2001. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering and System Safety*, 71, pp 229-247.

Papadopoulos, Y., Parker, D., Walker, M., Petersen, U., Hamann, R., Wu, Q., Uhlig, A., 2005. Automated failure modes and effects analysis of systems on-board ship. *International Conference on Marine Research and Transportation*, Ischia, Italy. Proceedings on CD.

Parker, D. 2010. *Multi-objective optimisation of safety-critical hierarchical system*. PhD Thesis. University of Hull.

Parker, D., Papadopoulos, Y., Walker, M., 2006. Component-based, automated FMEA of advanced active safety systems, *31st World Automotive Congress*. Published by JSAE, Yokohama, Japan. ISBN: 4-915219-83-6, 2006.

Parker, D., Papadopoulos, Y., 2007. Optimization of networked controlled systems using model-based safety analysis techniques. *in Proceedings of IEEE International Conference on Networking, Sensing and Control,* pp 425-430. London.

Peikenkamp, T., 2006. *Application of traditional and new safety analysis techniques in model-oriented design*. Germany: OFFIS .

Point, G., Rauzy.A., 1999. Constraint automata as a description language. *European Journal on Automation*, 33(8-9), pp 1033-1052.

Pumfrey, D., 1999. *The principled design of computer system safety analyses*. PhD Thesis. University of York, UK.

Quality associates, 1997. *Severity, occurrence and detection criteria for design FMEA*. Available through: < http://www.fmeainfocentre.com/guides/DesignPktNewRating.pdf> . Accessed at: [10/12/2010].

Rauzy,A., 2002. Mode automata and their compilation into fault trees. *Reliabilty Eingeering and System Safety*, 78(1), pp 1-12. Elsevier Science.

SafeProd, 2005. *Safety requirements specification guideline*. Process Industry IEC 61511. Available through: SafeProd< http://www.sp.se/sv/index/services/functionalsafety/Documents/Safety%20requirements%20specification%20guideline.pdf> [Accessed: 10/12/2010].

Schatz, B., Pretschener, A., Huber, F., Phillips, J., 2002. Model-based development of embedded systems, in *Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pp 298-312.

SEF, 2001. *Sytems engineering fundamentals*. Department of Defense. Virginia: Defense Acquisition University Press. Available through: <http://spacese.spacegrant.org/SEModules/Reference%20Docs/DAU_SE_Fundamentals.pdf>. Accessed at: [11/12/2010].

Seguin, C., Bieber, P., Castel, C., Kehren, C., 2006. *Formal assessment techniques for embedded safety critical system*.

Sharvia, S., 2007. *Extending fault tree synthesis with negative logic operator*. MSc Thesis. University of Hull, UK.

Sharvia, S., Papadopoulos, Y., 2008. Non-coherent modelling in compositional safety analysis, *in Proceedings of the 17th World Congress, International Federation of Automatic Control (IFAC WC'08)*. Seoul, Korea. Paper available from ifac-papersonline.net

Sharvia, S., Papadopoulos, Y., 2009. Model-based safety analysis using compositional analysis and formal verification, *5th International Conference on Computer Science and Information Systems (ICCSIS'09).* Athens, Greece.

Sharvia, S., Papadopoulos, Y., 2010. Integrating compositional safety analysis and formal verification. *Strategic Advantage of Computing Information Systems in Enterprise Management*, pp 181-201.

Sommerville, I., 2004. *Software engineering*. London: Addison Wesley. ISBN: 978-0321210265.

Suri, N., Walter, C., Hugue, M.M., 1995. *Advances in ultra-dependable distribute systems*. IEEE Computer Society Press. ISBN: 0818662875.

Torres-Pomales, W., 2000. *Software Fault Tolerance: A Tutorial*. Virginia: NASA Langley Research Center. Technical report: NASA-2000-tm210616.

Tribble, A.C., Miller, S.P., 2003. Software safety analysis of a flight management – a status report, *22nd Digital Avionics Systems Conference (DASC'03*).

Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F., 1981. *Fault tree handbook*. Washington D.C., USA: US Nuclear Regulatory Commission, III1-V-3.

Visser, W., 2002. *Software model checking*. Research Institue for Advanced Computer Science. NASA Ames Research Center.

Von Der Beek, M., 1994. A comparison of statecharts variants. In Langmaack, H., de Roever, W.P., Vytopil, J., eds, Formal techniques in real-time and fault-tolerant systems. Lecture Notes in Computer Science, 863, pp 128-148. Springer-Verlag.

Walker, M. and Papadopoulos, Y., 2006. PANDORA: *The time of Priority-AND gates*. INCOM 2006, France, pp 237-242.

Walker, M. *et al.*, 2008. Review of relevant safety analysis techniques. Traffic Efficiency and Safety through Software Technology Phase 2 (ATESST2).

Weilkiens, T., 2007. *Systems engineering with SysML/UML – modelling , analysis, design*. London: Elsevier, Morgan Kaufmann. ISBN: 0123742749.

Wilkinson, P.J., Kelly, T.P., 1997. Functional hazard analysis for highly integrated aerospace system, *in Proceedings of Certification of Ground/Air Systems Seminar*, pp 41-46. DOI: 10.1049/ic:19980312

Wolforth , I.P., 2010. *Specification and use of component failure patterns*. PhD Thesis, University of Hull.