Failure Logic Modelling: A Pragmatic Approach

Oleg Lisagor

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

March 2010

To the memory of my grandmother

Abstract

The research discipline of model-based system safety assessment, which has emerged in the last two decades, has attracted a significant amount of interest from academia, industry and government agencies. However, the discipline remains largely unorganised with various individual, often conceptually dissimilar, techniques being only categorised and related in an *ad hoc* fashion.

This Thesis identifies a coherent family of model-based safety assessment methods – *failure logic modelling* – and unifies existing techniques through a single well-defined Metamodel. This *Failure Logic Metamodel* (FLMM) identifies the key safety engineering concepts captured by failure logic modelling techniques, together with their inter-relationships. Whilst maintaining independence from any individual technique, notation or specification language, the abstract Metamodel has been shown to be *instantiable* in a third party-specification language (AltaRica Dataflow).

The Thesis demonstrates that existing failure logic modelling techniques cannot, without modification, adequately address key pragmatic challenges posed by extant characteristics of modern large-scale and complex safety-critical systems. To address such challenges two key contributions are made through extensions to the metamodel. Firstly, these extensions enable the *modelling of reconfigurable systems* (including those employing fault accommodation). Secondly, they enable the composition of *independently defined models* in a variety of settings, such as the composition of models of the same system defined from different viewpoints and composition of models of different systems with un-harmonised interfaces. In addition to these contributions, the general metamodel-based approach adopted by the thesis and proposed has helped *identify some significant 'emergent' characteristics and limitations of failure logic modelling* that, to date, have not been reported.

The overall contributions of the Thesis have been evaluated through case studies, peer reviews and direct metamodelling experiments. The findings of these evaluations are presented.

Table of Contents

ABSTRACT **3**

TABLE OF CONTENTS **4**

LIST OF FIGURES **11**

LIST OF TABLES **15**

ACKNOWLEDGEMENTS **17**

AUTHOR'S DECLARATION **18**

CHAPTER 1: INTRODUCTION **19**

1.1 FOREWORD: THE EVOLUTION OF SAFETY ANALYSIS METHODS **19**

1.1.1 TRADITIONAL SAFETY ASSESSMENT METHODS 19

1.1.2 FAILURE LOGIC MODELLING METHODS 22

1.1.2.1 Illustration 23

1.1.2.2 Claimed Benefits of Failure Logic Modelling 26

1.2 RESEARCH CHALLENGES **27**

1.2.1 COMPLEXITY OF BEHAVIOUR 27

1.2.2 COMPOSITION OF MULTIPLE FAILURE LOGIC MODELS 29

1.2.3 CONCEPTUAL INTEGRITY AND LANGUAGE INDEPENDENCE 31

1.3 MOTIVATION **32**

1.4 THESIS PROPOSITION **33**

1.5 THESIS STRUCTURE **33**

CHAPTER 2: LITERATURE SURVEY **35**

2.1 SAFETY ENGINEERING, ASSESSMENT AND TERMINOLOGY **36**

2.1.1 KEY TERMINOLOGY OF SYSTEM SAFETY 36

2.1.2 SYSTEM SAFETY ENGINEERING, ASSESSMENT AND LIFECYCLE 37

2.1.3 SCOPE OF THE PRESENT RESEARCH 41

2.2 CLASSICAL SAFETY ASSESSMENT METHODS **41**

2.2.1 INDUCTIVE METHODS 42

2.2.2 DEDUCTIVE METHODS 44

2.2.3 'BOWTIE' METHODS 49

2.2.4 DISCUSSION 52

2.3 Failure Logic Modelling Methods 52
2.3.1 FPTN 53
2.3.2 HiP-HOPS 55
2.3.3 Other Methods and Variants 57
2.4 Other Model-Based Safety Assessment Approaches 58
2.4.1 Failure Injection Approach 59
2.4.2 Failure Effects Modelling Approach 62
2.4.3 Hybrid Approaches 63
2.4.4 Model-Based Safety Assessment: Summary 66
2.5 Modelling Languages 68
2.5.1 AltaRica and Associated Dialects 69
2.5.2 The Architecture Analysis & Design Language 72
2.5.3 Language Selection 75
2.6 Conclusions 75

CHAPTER 3: UNIFYING FAILURE LOGIC METAMODEL 77

3.1 Introduction 77
3.1.1 Introduction to the Illustrative Example & Case Study 78
3.1.2 System Intent & Design 80
3.2. Unifying Metamodel for Existing Techniques 81
3.2.1 Common Key Concepts 83
3.2.2 Component Failure Logic 85
3.2.3 Model Structure and Hierarchical Organisation 86
3.2.4 Illustration 88
3.3 Extended FLMM: Dynamic and Normal Behaviour 90
3.3.1 Void Transition Triggers 91
3.3.2 Normal Events and States 93
3.3.3 Failure Handling States 95
3.4 Model Analysis 97
3.4.1 Fault Tree Synthesis and Sequence Generation 97
3.4.2 Simulation and FMEA/FMECA 100
3.5. Instantiation of the FLMM 100
3.5.1 Specification Language Requirements 101
3.5.2 Overview of Cecilia OCAS Tool 103
3.5.3 Representing FLM Concepts in AltraRica / OCAS 104

3.6 Case Study: Wheel Braking System 108
3.6.1 Predefined Types: Failure Modes and FM Classes 108
3.6.2 Model Architecture and Examples of Components 111
3.6.3 Virtual Components: Model-Level Input FMs and Observer 112
3.6.4 Model Simulation and Analysis 113
3.7 Role of the FLMM in the Safety Case 115
3.8 Limitations of the 'Baseline' Approach 116
3.9 Conclusions 118

CHAPTER 4: COMPOSITION OF MULTIPLE MODELS 119

4.1 Introduction 119
4.1.1 Illustration of the Problem Addressed by the Chapter 120
4.1.2 Objectives for the Composition of Failure Logic Models 122
4.2 Views and Domains of Safety-Critical Platforms 122
4.2.1 Views and Viewtypes 122
4.2.2 Domains and Domain-Specific Models 124
4.2.3 FLMM in Engineering Domain Framework 126
4.3 Allocation Viewtype and Composition of DSFMs 127
4.3.1 The Allocation Domain as the Unifying Concept 128
4.3.2 DSFM Interfaces and Composition 130
4.3.2.1 Peer Domains 131
4.3.2.2 Alternative Views 132
4.3.2.3 Different Semantic Spaces 134
4.4 Defining Composable DSFMs 136
4.4.1 The Consistency of FM Interfaces 136
4.4.2 Richness of 'Soft Interfaces' 138
4.4.3 The Granularity and Scope of Model Architectures 139
4.5. Instantiation in AltaRica 140
4.5.1 AltaRica OCAS Synchronisations 141
4.5.2 Emulating Dependent Weak Synchronisation 142
4.5.3 Instantaneous Events: Flow-to-Event Conversion 144
4.6 Case Study: Common Computational Platform 145
4.6.1 Overview of the Computation Infrastructure 147
4.6.1.1 Architecture of the Infrastructure System 147
4.6.1.2 DSFM of the Infrastructure System 148
4.6.2 Partition and VL Allocation 150

4.6.3 Integration Layer and DSFM Composition 151
4.6.3.1 Software Components: Partitions and Processes 152
4.6.3.2 Software Communications: Virtual Links 155
4.6.4 Model Analysis 158
4.7 Relationship to Common Cause Analysis 159
4.8 Model Complexity & Limitations of the Analysis Tool 162
4.9 Conclusions 164

CHAPTER 5: MULTI-MODE AND RECONFIGURABLE SYSTEMS 165

5.1 Introduction 165
5.1.1 Illustration of the Problem Addressed by the Chapter 165
5.2 System Modes 166
5.2.1 FLMM Extension 168
5.2.2 Modes as Component Context 173
5.2.2.1 Component Failures and Dynamic Exposure Intervals 173
5.2.2.2 Reusability of Component Characterisations 174
5.2.2.3 Context Boundaries and Dysfunctional Modes 176
5.3 Implementation of Modes in AltaRica/OCAS 180
5.3.1 Procedure for Encoding Modes in AltaRica 180
5.3.2 Illustration 183
5.4 Implications for the Modelling Process 186
5.4.1 Establishing Model Architecture 186
5.4.2 Basic Components Characterisation and Model Composition 189
5.5 Case Study: Electrical Power Distribution System 190
5.5.1 EPDS Overview 190
5.5.2 Model Architecture: Key Principles and Assumptions 192
5.5.3 Failure Logic Model Architecture: Modes 194
5.5.4 Components Failure Logic Characterisation 196
5.5.5 Refinement of Mode Models 199
5.5.6 Characterisation of Controllers 201
5.6 Key Findings and Limitations 203
5.6.1 Loops 203
5.6.2 Order of Transitions with Void Triggers 204
5.6.3 Complexity of Component Characterisations 205
5.6.4 Analysis Complexity 206

5.7 Related Work 207

5.8 Conclusions 210

CHAPTER 6: EVALUATION 213

6.1 Evaluation Strategy 213

6.1.1 Technical Challenges Identified by the Thesis Proposition 214

6.1.2 Pragmatic Approach Hypothesis 216

6.1.3 Well-Defined and Sound Approach Hypothesis 217

6.2 Evaluation through Case Studies 220

6.2.1 Wheel Braking System 221

6.2.2 Aircraft Fuel System 221

6.2.3 Integrated Modular Avionics 224

6.2.4 Aircraft Electrical Power Distribution System 226

6.2.5 Case Studies Summary 229

6.3 Metamodel Experiments 230

6.3.1 FLMM Validation in Eclipse 231

6.3.2 Mapping between FLMM and Existing Failure Logic Modelling Methods 233

6.3.2.1 HiP-HOPS 233

6.3.2.2 FPTN 235

6.3.2.3 Summary: HiP-HOPS as a Set of FLMM Constraints 236

6.3.3 Relationship Between the FLMM and FTA 237

6.3.4 Non-Coherent and Dynamic Behaviour 240

6.3.4.1 Negation 240

6.3.4.2 Priority AND Gate 240

6.3.4.3 Dynamic Fault Tree Gates 242

6.3.5 Evaluation by Metamodel Instantiation 243

6.4 Evaluation Through Peer Review 244

6.4.1 Airbus Dependability Network 244

6.4.2 The MISSA Project 245

6.4.3 Peer Review Summary 246

6.5 Identified Limitations and Mitigations 248
6.5.1 Volume of Results 249
6.5.2 Strong Circular Dependencies 250
6.5.3 Complex Modes and Reconfiguration Logic 251
6.5.4 Reuse and Composability of the Component Characterisations 252
6.5.5 Complexity of Model Construction 254
6.5.6 Complexity of Model Analysis 254
6.6 Summary 255

CHAPTER 7: CONCLUSIONS 257

7.1 Summary of Contributions 257
7.1.1 Unifying Failure Logic Metamodel 258
7.1.2 Composition of Multiple Failure Logic Models 258
7.1.3 Modelling Reconfigurable and Multimodal Systems 259
7.1.4 The Non-Automatable and Non-Decomposable Nature of Failure Logic Modelling 260
7.2 Further Work Areas 260
7.2.1 Transformation of Failure Logic Models 261
7.2.2 Model Analysis 261
7.2.3 Issues of Modelling Time 262
7.2.4 Improving Reusability 262
7.2.5 Alternative Modelling Paradigms 262
7.2.6 Other Model-Based Safety Assessment Approaches 263
7.3 Coda 263

APPENDIX A: FAILURE LOGIC METAMODEL 265

A1. Metamodel Specification 267
A2. Well-Formedness Constraints 271

APPENDIX B: WHEEL BRAKING SYSTEM CASE STUDY 277

B2. WBS Failure Logic Model: "Pseudo-code" 278
B2.1 Braking System Control Unit 278
B1.2 Hydro-Mechanical Components 284
B2. WBS Failure Logic Model: AltaRica Dataflow 291
B3. Revised BSCU Model: AltaRica Dataflow 302

APPENDIX C: COMPUTATION AND COMMUNICATIONS PLATFORM DSFM.....309

C1. MODIFICATION OF THE BSCU MODEL (WBS DSFM).....309

C2. INTRODUCTION TO THE COMPUTATION INFRASTRUCTURE AND ARCHITECTURE DESCRIPTION.....311

C2.1 INTRODUCTION TO THE INTEGRATED MODULAR AVIONICS 311

C2.2 ARCHITECTURE OF THE INFRASTRUCTURE 314

C3. MODEL DESCRIPTION 316

C3.1 INFRASTRUCTURE FAILURE MODES: "EXTERNAL" OUTPUT FMS AND IMA FUNCTIONS.....317

C3.2 INFRASTRUCTURE FAILURE MODES: "INTERNAL" FMS.....320

C3.3 FAILURE LOGIC MODEL: NETWORK COMPONENTS.....321

C3.4 FAILURE LOGIC MODEL: CPIOMS.....325

APPENDIX D: SUMMARY OF THE AIRCRAFT FUEL SYSTEM REVIEW 327

D1. COMPLEX MODE LOGIC 327

D2. INTENTIONAL ARCHITECTURAL LIMITATIONS.....329

D3. COMPLEXITY OF SCALE AND DESIGN DECOMPOSITION 330

D4. CIRCULAR DEPENDENCIES AND LOOPS.....331

D5. TIME-DEPENDENCY AND RELIANCE ON CONSUMABLE RESOURCE 333

ABBREVIATIONS.....335

REFERENCES.....337

List of Figures

Figure 1 - Schematic of the Illustrative Tank System....23
Figure 2 - HiP-HOPS Characterisation of the Valve....24
Figure 3 - Example of a Fault Tree for "Tank Overflow" Condition....25
Figure 4 - Minimal Cut Sets for "Tank Unable to Provide Fluid" Condition....25
Figure 5 - ARP4761 Safety Assessment Diagram[139]....40
Figure 6 - Principal FTA Event Symbols....45
Figure 7 - Standard FTA Logic Gate Symbols....45
Figure 8 - Villemeur's 'Failure Classification as to Causes' [160]....46
Figure 9 - Dynamic Fault Tree Gates: Functional Dependency (a), Cold Spare (b)....47
Figure 10 - Specific Symbols of Cause-Consequence Diagrams [160]....51
Figure 11 - Example of FPTN Module: Hydraulic Pump....54
Figure 12 - Example of HiP-HOPS Component: Hydraulic Pump....55
Figure 13 - HiP-HOPS Model Hierarchy (schematic)....56
Figure 14 - Example of FPTC Equations....58
Figure 15 - Overview of the FI/ESM Approach....60
Figure 16 - Node Example in AltaRica: Switch....70
Figure 17 - Node Composition in AltaRica: Switch Pair....70
Figure 18 - Node Example in AltaRica Dataflow: Switch....71
Figure 19 - Example of a Component in the AADL Error Model Annex [56]....73
Figure 20 - Architecture of the Hypothetical Aircraft Wheel Braking System....78
Figure 21 - BSCU Architecture....79
Figure 22 - Partial FLMM: Elementary Concepts....85
Figure 23 - Permissible Flows in Hierarchical Failure Logic Models....87
Figure 24 - Basic FLMM: Model Structure and Component Behaviour....88
Figure 25 - Failure State Spaces Illustration: Green Meter Valve....90
Figure 26 - Void Transition Triggers (in the Immediate FLMM Context)....92
Figure 27 - Fault Tree Segment for Unavailability of Pressure from the Accumulator....93
Figure 28 - Normal States and Events (in the Immediate FLMM Context)....94
Figure 29 - Complete Baseline FLMM....96
Figure 30 - Possible Fault Tree for Pressure Omission Failure Mode of the Green Meter Valve....98
Figure 31 - Cecilia OCAS Interface: Behaviour Specification....103
Figure 32 - AltaRica Characterisation of the WBS Accumulator (FMs as Booleans)....106
Figure 33 - Failure Logic Characterisation of the Accumulator (Pseudo-code)....107
Figure 34 - OCAS Characterisation of Accumulator (FMs as Enumeration Literals)....108
Figure 35 - OCAS: HydraulicFMs Type Along with Field Types (PressureFMs & LeakFM)....109

Figure 36 - OCAS: PumpFMs Type 110
Figure 37 - AltaRica: Failure Logic Characterisation of the Isolation Valve 111
Figure 38 - OCAS: Top-Level Failure Logic Model Architecture of WBS 112
Figure 39 - Characterisation of the 'Virtual' Input Component for *PowerFMs* Type Input 113
Figure 40 - OCAS: WBS Model Architecture with 'Virtual' Input Components 113
Figure 41 - OCAS: Model Simulation 114
Figure 42 - SEI "Views and Beyond" Framework 124
Figure 43 - Engineering Domain: Internal Structure and Traces to Platform 125
Figure 44 - DSFMs and Engineering Domains 127
Figure 45 - Allocation Domain 129
Figure 46 - Peer Domains: Composition of Electrical and Wheel Braking Systems' DSFMs 132
Figure 47 - Alternative Views: Composition of Leaks and Pressure DSFMs (WBS) 134
Figure 48 - Transformation of the Target ('Effect') Component: Valve in "Pressure DSFM" 143
Figure 49 - Transformation of the Source ('Cause') Component: Valve in "Leaks DSFM" 143
Figure 50 - Synchronisations Between "Leaks" and "Pressure" DSFMs: Green Meter Valves 144
Figure 51 - Boolean Flow Detector Component 145
Figure 52 - Revised Model Architecture of BSCU 146
Figure 53 - Revised Model Architecture of a BSCU Channel 146
Figure 54 - Simple IMA Architecture 147
Figure 55 - Redundant AFDX Networks 148
Figure 56 - Architecture of IMA DSFM (Power FM Flows not Shown) 149
Figure 57 - Architecture of the Failure Logic Model's Translation Layer 152
Figure 58 - FM Flows Between the CPIOM and Allocated Partitions (Side 1 Only) 152
Figure 59 - AltaRica Characterisation of Homogeneous Partition Translation Component 153
Figure 60 - Synchronisations Between Translation and WBS Components 154
Figure 61 - Internal Structure of the COM Partition Translation Component 154
Figure 62 - Structure of a Simple VL Translation Component (CPIOM-Internal VLs) 156
Figure 63 - Structure of a General VL Translation Component (VLs Across Different CPIOMs) 156
Figure 64 - Composed Failure Logic Model (Automatically Generated Flows) 157
Figure 65 - CCA in DSFM Context 160
Figure 66- Revised FLMM: Modes of Complex Components 169
Figure 67 - Revised FLMM: Normal Events of Complex Components 170
Figure 68 - Complete Revised Failure Logic Metamodel (FLMM) 172
Figure 69 - Difference Between Failure Logic of Green and Blue Meter Valves 175
Figure 70 - BSCU Architecture 178
Figure 71 - BSCU Modes and Transitions 179
Figure 72 - Revised Failure Logic Model Architecture of BSCU (partial) 179
Figure 73 - Characterisation of the Virtual Translator Component 179

Figure 74 - BSCU MonitoringModeObserver in AltaRica OCAS 183
Figure 75 - Revised AltaRica OCAS Specification of the Switch Component 184
Figure 76 - AltaRica Characterisation of the Virtual Translator Component 184
Figure 77 - BSCU Synchronisations 185
Figure 78 - Revised Architecture of BSCU's Failure Logic Model 185
Figure 79 - EPDS Architecture 191
Figure 80 - Architecture of EPDS Failure Logic Model 193
Figure 81 - EPDS Mode Model (partial) 196
Figure 82 - Partial Characterisation of J2 Junction (AltaRica OCAS) 199
Figure 83 - GEN Controller in EPDS 202
Figure 84 - Characterisation of GEN Controller (AltaRica OCAS) 203
Figure 85 - Some Loops in the AC Section of EPDS 204
Figure 86 - Static and Dynamic Model Hierarchies in HiP-HOPS 208
Figure 87 - Hierarchical Representation of EPDS Modes 209
Figure 88 - Evaluation Strategy 214
Figure 89 - Technical Challenges Identified by the Proposition (Evaluation Argument) 215
Figure 90 - Pragmatic Approach Hypothesis (Evaluation Argument) 218
Figure 91 - Well-defined Approach Hypothesis (Evaluation Argument) 219
Figure 92 - Strong Circular Dependency in EPDS Models 228
Figure 93 - Outline of the Overall EPDS Mode Model 229
Figure 94 - FLMM Extract (Emfatic) 231
Figure 95 - FLMM Extract (Ecore Diagram) 232
Figure 96 - FLMM Constraints Extract (EVL) 232
Figure 97 - HiP-HOPS Metamodel ('Core' Method Only) 234
Figure 98 - Relationship between HiP-HOPS and Failure Logic Metamodels 235
Figure 99 - Basic Failure State Space Model (FLM) for a HiP-HOPS Malfunction "Malf" 235
Figure 100 - Mapping between FTA and Failure Logic Metamodels 238
Figure 101 - Typical Interpretation of a PAND gate 241
Figure 102 - Improved Fault Tree Representation of a State-Guard-Trigger relationship 241
Figure 103 - Dynamic Fault Tree Gates: Functional Dependency (a), Cold Spare (b) 242
Figure 104 - Failure Logic Metamodel (Ecore Diagram) 266
Figure 105 - Architecture of the Hypothetical Aircraft Wheel Braking System 277
Figure 106 - Revised Architecture of the BSCU Model in AltaRica OCAS 302
Figure 107 - Revised Model Architecture of BSCU 310
Figure 108 - Revised Model Architecture of a BSCU Channel 310
Figure 109 - Failure Logic Characterisation of CMD Dataflow Component (AltaRica) 310
Figure 110 - Revised BSCU Monitor Component with "External" Failure Causes 311
Figure 111 - Layers of the IMA 312
Figure 112 - Simple IMA Architecture 314

Figure 113 - Internal Structure of an End Node 315
Figure 114 - Redundant AFDX Networks 316
Figure 115 - Architecture of the IMA DSFM 317
Figure 116 - Failure Logic Model of an End Node (Data Propagation "Pipelines" Only) 321
Figure 117 - AltaRica Characterisation of *PortInterface* Component 322
Figure 118 - AltaRica Characterisation of the *MemoryUnit* Component 323
Figure 119 - Structure of the *EndNode* Complex Component 324
Figure 120 - Structure of the *AFDXswitch* Complex Component Model 324
Figure 121 - Structure of the CPIOM Complex Component Model 325
Figure 122 - AltaRica Characterisation of the *Scheduler* Component 326
Figure 123 - Schematic of Wing Tanks Architecture 328
Figure 124 - Simplified Phases of Operation 328
Figure 125 - Simplified Mode Model of the Fuel System 329
Figure 126 - Schematic of a Trivial Aircraft Fuel System 332

List of Tables

Table 1 - HAZOP Guidewords [90]....50
Table 2 - Classification of the Model-Based Safety Assessment Methods....67
Table 3 - Mapping Between FLM Concepts and Appropriate AltaRica/OCAS Constructs....104
Table 4 - Minimal Cut Sets for Inadvertent Braking....114
Table 5 - Stereotypes of Allocation Domains (depending on the scope and viewtype)....129
Table 6 - Output Flows of CPIOM Components....150
Table 7 - EPDS Reconfiguration Rules....192
Table 8 - Basic Components' Sensitivity to- and Intent in- Modes....197
Table 9 - EPDS Mode Transitions Specification....200
Table 10 - Intended Behaviour of GEN Controller in Various Modes....201
Table 11 - Case Studies Coverage (Outline)....220
Table 12 - Coverage Achieved by the Case Studies....230
Table 13 - Summary of Peer Review Instances....247
Table 14 - Classes of the “External” IMA Output Failure Modes (IMA DSFM)....319

Acknowledgements

I am forever indebted to my supervisor, *Tim Kelly*, for his encouragement and invaluable advice. Without his help I wouldn't be able to finish this thesis.

I am also grateful to my colleagues at the High Integrity Systems Engineering Group and, particularly, *Katrina Attwood*, *David Pumfrey* and *John McDermid* for their continuous support.

Also, I would like to thank my external examiners, *Professor John Andrews* (Loughborough University, UK) and *Professor Mats Heimdahl* (University of Minnesota, USA) for their time, insightful comments and robust discussions as well as exceptionally kind and supportive examination style.

Having greatly benefited from the involvement in three European collaborative projects – ISAAC, Airbus DepNet and MISSA – I would like to thank colleagues from partner institutions for a number of challenging discussions and their contributions to my research: *Ove Åkerlund*, *Pierre Bieber*, *Marco Bozzano*, *Matthias Bretschneider*, *Antonella Cavallo*, *Marion Morel* and *Christel Seguin*. I am especially grateful to *Chris Papadopoulos* (Airbus Operations, UK) for the opportunity to work in these projects, his help and friendship.

Jean Gauthier, *Valerie Sartor* and *Xavier Leduc* of Dassault Aviation have provided me with licenses to Cecilia OCAS suite and continuous technical support which I gratefully acknowledge.

Finally, I am grateful to my friends who have become my family in the UK and helped me to maintain the pretence of sanity – *Bernadette Martinez-Hernandez*, *Jennifer Winter*, *David Efird*, *Barry Miller* and *Alvaro Miyazawa* – and, most importantly, to *my parents* for their love and support.

Author's Declaration

All of the work contained within this thesis, unless explicitly stated otherwise, represents original research and contribution of the author.

Some of the research presented in this thesis has been conducted under the auspices of two projects: Dependability Network (DepNet) and, ongoing at the time of writing, More Integrated Systems Safety Assessment (MISSA) – funded by Airbus Operations and European Commission respectively. Consequently, some of the material presented in Chapters Three and Four has been previously included in the DepNet 'White Paper' Report [74]. The material presented in Chapters Three and Five has been incorporated into the 'Failure Logic Modelling Handbook' written by the author and included (in a draft form) in a MISSA project deliverable [94].

Further, some of the material of this thesis has been previously published in conference papers [95-97].

Chapter 1: Introduction

1.1 Foreword: The Evolution of Safety Analysis Methods

Having emerged in the 1940s as a 'grass roots' movement, System Safety Engineering evolved into an organised engineering discipline by the 1960s [151]. System safety assessment has been identified as one of the key aspects of the safety engineering discipline. System safety assessment is concerned with understanding how unsafe conditions (hazards) may arise as a result of interactions between system components (including in the presence of failures).

1.1.1 Traditional Safety Assessment Methods

One of the first system safety assessment methods to emerge was Failure Modes Effects and Criticality Analysis (FMECA). Still widely used today, it was originally specified in November 1949 in US Military Procedure MIL-P-1629 (later superseded by MIL-STD-1629 [152]). The declared purpose of FMECA is "*to study the results or effects of item failure on system operation*" and rank the identified groups of failures (failure modes) "*according to the combined influence of severity classification and its probability of occurrence based upon the best available data*" [152]. The wider objective of FMECA is to facilitate the rational risk-based prioritisation of system design (or re-design) activities.

FMECA is an inductive (or a 'forward search') analysis approach that considers all possible effects of a single (or a small number of) component failure(s) (generalised into failure modes, which are defined as "*the manner by which a failure is observed. Generally describes the way the failure occurs and its impact on equipment operation*" [152]). The analysis is repeated for every failure mode of every component of the system. MIL-STD-1629 provides some guidance on how the analysis should be conducted; for instance, it stresses the importance of clear design definition, suggests minimal (broad) 'kinds' of failure modes that must be considered and requires the documentation of "local", "next-higher-level" and "end" effects of such equipment failure modes. The standard also provides guidance on the classification of probabilistic and severity characteristics of failure modes. However, the focus of most normative FMECA descriptions is not on the system analysis per se, but rather on overarching process management considerations such as the systematic documentation of analysis outcomes and recommendations. Possibly because of the lack of comprehensive system safety programs at the time of the original FMECA definition, the method essentially defines its own context.

In the 1960s, the US Minuteman Inter-Continental Ballistic Missile (ICBM) program provided a 'break through' in system safety engineering. The program was launched against a dual backdrop. Firstly, the Cold War and the Cuban Missile Crisis led to a sudden and dramatic increase (at least in the perception of the general public, government and military) in the potential severity of any possible incident or accident involving strategic weapons (far beyond the immediate catastrophic damage caused by the accident itself[1]). The historical approach to safety engineering, which partially relied on a fly-fix-fly approach and partially on decentralised and uncoordinated safety engineering responsibilities, evidently became *socially unacceptable* in this context.

Second, pre-Minuteman ICBM programs – such as Atlas and Titan – had been marred by serious and widespread safety deficiencies [91]. This demonstrated both that unstructured and ad hoc approaches to safety assessment are *no longer technically feasible* given the complexity of military systems and the fact that the *costs of retrospective design modifications are prohibitively high*[2].

As a result, the Minuteman ICBM project is widely accredited as pioneering a "contractual, formal, disciplined system safety program" [91]. In addition to (and, possibly, because of) this, the program also specified the first comprehensive and structured system safety assessment method – Fault Tree Analysis (FTA).

Arguably the most widely used safety analysis method today, FTA was originally developed in 1961 at Bell Telephone Laboratories, extended by the Boeing Company [48, 90] and, eventually, codified by the US Nuclear Regulatory Commission in 1981 [157]. In contrast to FMECA, FTA adopts a deductive – or backward search – approach. Starting from a specified undesired system-level effect (a "Top Level Event"), and guided by explicit construction principles and rules, the analysis method systematically and iteratively identifies the immediate causes of conditions (faults) until some elementary level is reached. The result of the system analysis is a hierarchical structure of such conditions connected by logical gates (typically representing logical disjunction and conjunction).

[1] For instance, Sagan attributes the following quote to the US Assistant Secretary of Defense: "*The explosion of a nuclear device by accident – mechanical or human – could be a disaster for the United States, for its allies, and for its enemies. If one of those devices accidentally exploded, I would hope that both sides had sufficient means of verification and control to prevent the accident from triggering a nuclear exchange. But we cannot be certain that this would be the case*" [130]

[2] Leveson reports (attributing to Rogers's 1971 "Introduction to System Safety Engineering") that the cost of necessary modifications to Atlas F missiles "*would have been so high that a decision was made to retire the entire weapon system and accelerate deployment of the Minuteman missile system*" [91]

FTA was revolutionary in that it provided a *systematic analysis method* rather than merely stating objectives of the activity and expected outcomes from it. Further, it has (implicitly) separated three key elements of the method:

a) a systematic process for the assessment of system design
b) a clear notation for capturing a model of causal relationships
c) methods for analysing this model (such as minimal cut set identification and various quantitative analyses)

Finally, FTA defined the concept of "failure space" – a view of the system that exemplifies causal dependencies between abnormal and undesired conditions (faults).

It is important to note that inductive and deductive analysis methods are often complementary and effective in different contexts. Inductive methods are typically incapable of considering complex and 'wide' failure scenarios but may be used to identify system hazards, or to verify that the set of hazards which have been elicited previously is complete. By contrast, deductive methods (such as FTA) must start with a specified system-level condition and thus cannot facilitate the identification or the verification of hazards; however, these methods can systematically identify failure scenarios which contain a large number of apparently independent failures.

Partly because of these complementary strengths of deductive and inductive approaches, methods – such as Hazard and Operability Analysis (HAZOP) [85, 143] – combining both search strategies have emerged. HAZOP is a 'bow-tie' technique whereby analysis starts with an internal system condition and proceeds both deductively and inductively to identify possible causes and consequences of the condition. Originating from the chemical process industry, HAZOP is a structured and systematic brainstorming technique which utilises a description (typically – a schematic) of the plant under consideration.

To identify conditions of interest, the team of experts considers possible interpretations of deviation "guidewords" (such as "more", "less", "as well as") in the context of the physical characteristics of each flow between plant elements identified in the design drawing. The resultant viable deviations are used to prompt a systematic walk-through of the design drawing to identify all possible causes of deviation and, subsequently, all possible plant level effects. Throughout these walkthroughs the analysis team establishes whether the deviation is hazardous, whether the operator is likely to be aware of the deviation from the intent and what protective measures are feasible. In addition to being a powerful and still popular analysis method, HAZOP is relevant to the model-based assessment approach developed in this thesis, since it establishes notions of "intent" and "deviation".

All traditional techniques, regardless of their logical orientation (backward, forward or 'bow-tie'), share one important characteristic – they investigate, sometimes iteratively, the *causal projection of <u>one</u> (or a very small number of) event(s)* of interest. For example, FTA investigates a backward projection of a single top-level event, and a single row in an FMECA table typically captures a forward projection of a single initiating event. In consequence, each method not only produces voluminous results, but also produces a large *number* of (sets of) different results (e.g. numerous fault trees or FMECA tables) that, whilst clearly related to the same system, cannot be easily and systematically correlated. Furthermore, the increasing complexity and scale of modern safety critical systems means that different analysis methods may be employed to investigate different aspects or parts of the system. A need to integrate these different analysis methods and to correlate their artefacts in a conceptually consistent and coherent way has motivated novel safety assessment methods such as Failure Propagation and Transformation Notation (FPTN) [57, 58] and Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [115, 117].

1.1.2 Failure Logic Modelling Methods

Whilst numerous analysis methods, including HAZOP (described above) and cause-consequence diagrams [90, 123], have *combined* the principles of inductive and deductive analysis, FPTN and HiP-HOPS have *unified* them by considering a system at the lowest level of design decomposition – the level of elementary components. The key idea behind both methods is the specification of causal relationships between deviations of component outputs on the one hand and internal failures of the components and input deviations on the other.

Within this setting and at the level of 'basic components', inductive and deductive methods converge: it is possible to perform a forward search exhaustively for all *combinations* of input deviations and internal failures as well as to perform a backward search for all output deviations. The resultant component specification holds information that can be seen as equivalent to *multiple* fault trees or FMEA tables with consistency further ensured (by construction) by means of reference to the same sets of failures, input and output deviations. Whilst the complexity of such collections of fault trees and FMEA tables is clearly greater than that of a single artefact, it is nevertheless manageable because of the relative simplicity of the component (i.e. the relatively small number of inputs, outputs and viable failures). Combined with a model of interdependencies between components (or component interactions in the "failure world"), the above characterisations hold all of the information necessary for the synthesis of system-wide FMEA tables or fault trees – in HiP-HOPS author's words: "*If we know the 'structure' of a system (model) and the 'local failure behaviour of its components' (IF-FMEAs) then we can mechanically derive the 'failure behaviour of the system' (fault trees)*" [115].

1.1.2.1 Illustration

The basic principles of HiP-HOPS and FPTN can be demonstrated on a simple system used extensively at the University of York for teaching Fault Tree Analysis. The system (Figure 1) is concerned with maintaining a certain level of potentially hazardous fluid in a tank. It primarily consists of a tank with an outflow pipe and a valve-controlled inflow pipe. The inflow valve can be either open or closed with the state being determined by a controller, implemented in software, based on inputs of two redundant level sensors located in the tank. When the level of fluid in the tank reaches a certain level (designated as "full") the controller closes the valve; as soon as the level is less than full – the controller opens the valve.

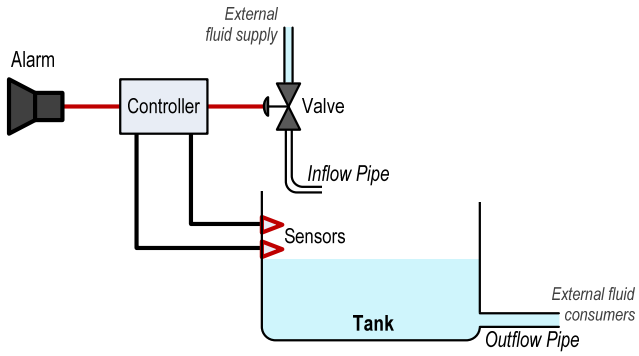


Figure 1 - Schematic of the Illustrative Tank System

Since the key hazard associated with this system is tank overflow, a "full" reading from either sensor is considered sufficient to stop the supply. Furthermore, the controller is capable of detecting certain classes of sensor failures (open and short circuits – which are most likely to occur). When failure of a single sensor is detected the controller raises the alarm and ignores the value of that sensor; if failures of both sensors are detected the controller shuts down the fluid flow into the tank (and, again, raises the alarm).

The system is also associated with another hazard: inability of a tank to supply fluid on the outflow pipe. This has lesser severity than the overflow condition.

Assuming that it is permissible to disregard failure of the pipework and electrical connectors in safety assessment, HiP-HOPS and FPTN models of the system would consist of five modules – corresponding to the components described above: tank, two sensors, valve and the alarm. Each module describes causal relationship between output deviations of the component on the one hand and its internal failures and input deviations on the other. For example, the tank module will have two possible output deviations of the fluid level: *Value Too High (overflow)* and *Value Too Low (inability to provide fluid downstream)*. The former can only be caused by a *commission* (provision when not required) of the inflow whereas the latter can be caused by either *omission (non-provision when required) of the inflow* or *Rupture* of the tank itself. In contrast, the two (identical)

sensor modules each have four possible output deviations: *Open Circuit*, *Short Circuit*, *Reading Too Low* and *Reading Too High*. However, these output deviations can only be caused by the *internal malfunctions* of the sensors. Finally the output deviations of the valve and controller modules can be caused by input deviations as well as internal failures. To illustrate, Figure 2 shows a possible characterisation of the valve module (in a *HiP-HOPS*-like format).

Output Failure Mode	Description	Input Deviation Logic	Component Malfunction Logic	Failure Rate (λ)
Hyd_Output.Omission	Valve does not provide fluid when intended Caused by internal failure of the valve (*Stuck_Closed*) or omission of fluid flow into the valve (*Hyd_Input.Omission*) or lack of a command from the controller to open (*Ctrl_Input.Omission*)	*Hyd_Input.Omission* | *Ctrl_Input.Omission*	*Stuck_Closed*	2e-4
Hyd_Output.Commission	Valve provides fluid when not intended Caused by internal failure of the valve (*Stuck_Open*) or inadvertent command from the controller to open (*Ctrl_Input.Omission*)	*Ctrl_Input.Commission*	*Stuck_Open*	8e-4

Figure 2 - HiP-HOPS Characterisation of the Valve

The five modules are interconnected in terms of "flows of deviations" (i.e. ability of an output deviation of one component to affect the correct operation of the other) allowing for automated traversal of the model to synthesise fault trees or FMEA tables. For instance, Figure 3 shows a fault tree for the overflow condition. Alternatively, the model can be analysed directly – to generate Minimal Cut Sets (MCSs) for any condition (e.g. Figure 4 for the inability of the tank to supply fluid on demand). It is important to stress that all such analysis artefacts (FTs, FMEA tables and MCSs) generated from the same model for either the same or different conditions will be consistent by construction.

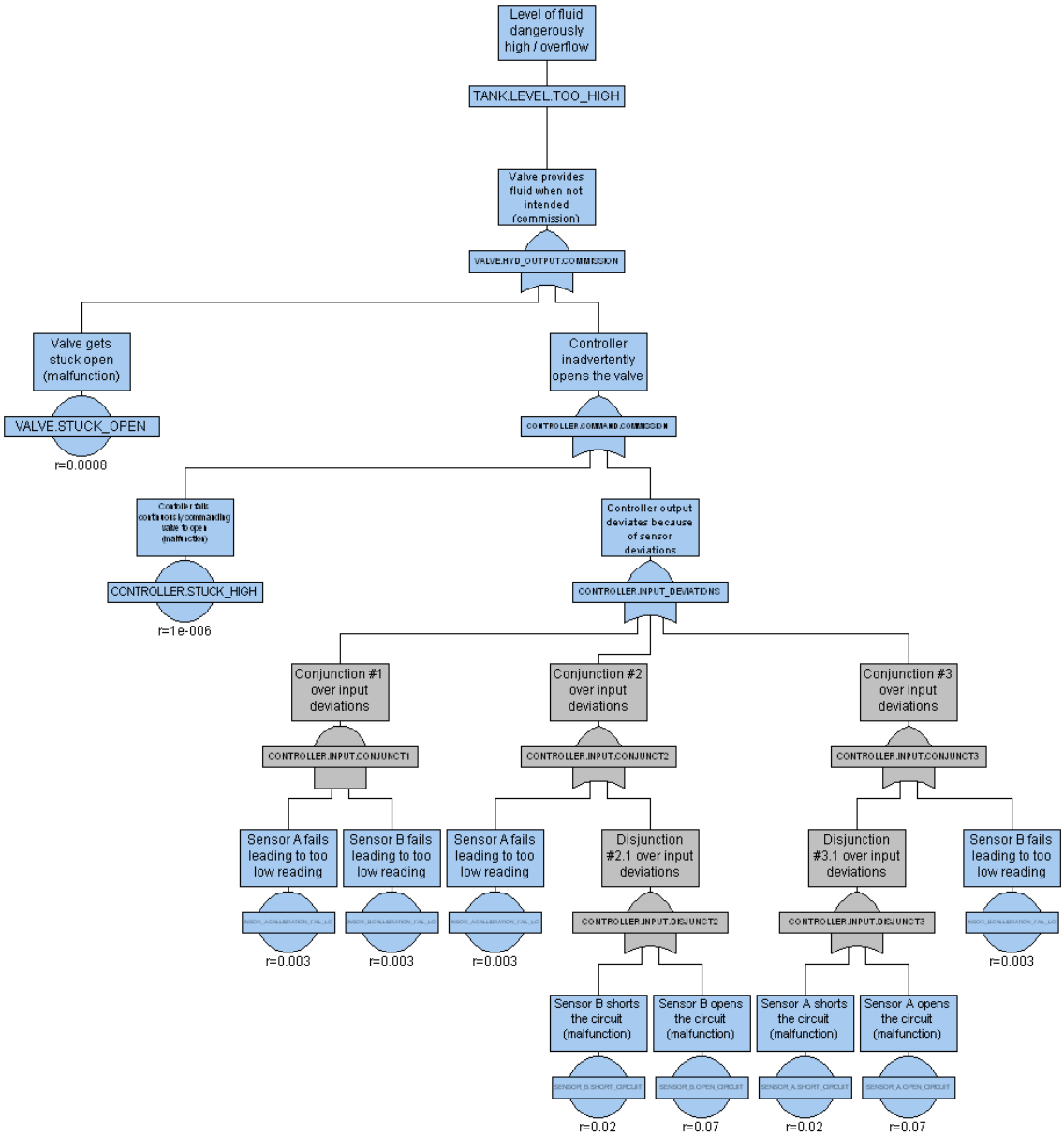


Figure 3 - Example of a Fault Tree for "Tank Overflow" Condition

Minimal Cut Sets: *Omission-Level* (*Tank* Module)	
Tank.Rupture	
Controller.Stuck_Low	
Valve.Stuck_Closed	
SystemInput.Fluid_Supply.Omission	
Sensor_A.Callibration_Failure_High	
Sensor_B.Callibration_Failure_High	
Sensor_A.Fail_Open	Sensor_B.Fail_Open
Sensor_A.Fail_Open	Sensor_B.Fail_Short
Sensor_A.Fail_Short	Sensor_B.Fail_Open
Sensor_A.Fail_Short	Sensor_B.Fail_Short

Figure 4 - Minimal Cut Sets for "Tank Unable to Provide Fluid" Condition

1.1.2.2 Claimed Benefits of Failure Logic Modelling

Although HiP-HOPS and FPTN unify the key classical system safety analysis methods, they are most closely related to FTA. Both approaches share the notion of "failure space" and advocate the explicit modelling of causal dependencies between various events and conditions within this 'world'. Most importantly, both approaches also share a key strength (which is often overlooked in literature) – they facilitate a review of the system from an analytical perspective which is *conceptually dissimilar* to that adopted by the design process. One of the key goals of such a review is to uncover dependencies between system components which were previously overlooked by the design engineers or whose effect on safety was unappreciated. As a result, the mere act of constructing fault trees or failure logic models can generate value-adding feedback to the development process before the constructed models are fully analysed.

In addition to this shared strategic strength, HiP-HOPS, FPTN and similar methods which were subsequently developed have claimed a number of (related) relative advantages over the classical methods:

- the ability to partially *automate safety analysis* of the system
- the establishment of much clearer and more detailed *traceability* between system design and safety assessment artefacts through an explicit (and shared) notion of the "component"
- the ability to perform safety analysis in a *compositional* fashion
- the use (and construction) of *composable safety analysis* artefacts
- the ability to *reuse safety analysis* artefacts (components' characterisations) whenever the corresponding components are reused across different design models.

Overall, the artefacts produced by HiP-HOPS, FPTN and some of the later methods developed using similar concepts (e.g. [163, 165]) became collectively known as failure propagation models and, lately, *failure logic models* [96]. This thesis identifies how these methods can be applied to modern complex safety-critical systems and how their underlying conceptual framework can be systematically extended to tackle some of the key challenges posed by such systems in industrial context. It is important to stress that the objective of this thesis is *not* to present "yet another" failure logic modelling method but rather to consider the entire family of existing methods, identify their shared practical weaknesses and propose pragmatic extensions.

The following sections present the motivation for, and the challenges addressed by, this thesis. The chapter concludes with definition of the Thesis Proposition and the chapter-by-chapter outline of the thesis.

1.2 Research Challenges

Despite its apparent strategic advantages (discussed above), failure logic modelling has not hitherto been widely accepted by industry. The main obstacle to the acceptance is the lack of application to systems of realistic complexity and scale. The two key examples of properties of industrial safety-critical systems which are not addressed by failure logic modelling techniques are:

- *Operation over different modes*, whereby the system dynamically reconfigures upon either detection of failure or transition between phases of operation. Failure logic modelling has been perceived as being fundamentally inapplicable to systems exhibiting such behaviour because of its typical reliance on both the combinatorial framework for describing component logic and on a purely compositional view of the systems (whereby behaviour of the system is attributed to the behaviour of elementary components).
- *Complexity of scale*, whereby it is impractical to design large-scale industrial safety-critical systems as a single, unified whole. Instead, such systems are typically designed in a federated fashion by a number of different stakeholders. Consequently, some of the interactions between such independently-designed systems can only be identified at integration time. Failure logic modelling is perceived as inapplicable in this context since, under current methods, composition relies on the pre-existence of well-identified and matching interfaces between models.

Furthermore, failure logic modelling techniques typically introduce idiosyncratic specification notations which are often perceived as being unsuited to an industrial setting. Reliance on such notations also makes it difficult to relate individual techniques and to establish whether particular limitations are specific to an individual technique or the failure logic modelling approach as a whole.

The following sections describe these challenges to the failure logic modelling approach, addressed in this thesis, in more detail.

1.2.1 Complexity of Behaviour

Model-based safety assessment approaches have historically been motivated by the increasing complexity of modern safety-critical systems. A key aspect of this complexity is reliance on *dynamic reconfiguration*, whereby a system may perform different functions at different stages of its operation or may deliver functions in different ways depending on its failure history. Traditional safety analysis methods are perceived as being inefficient (if capable at all) in the context of highly reconfigurable systems.

Whilst the failure logic modelling approach shares this motivation, the issues related to reconfigurable systems have largely remained unaddressed in publications on the proposed methods. The challenges reconfigurable systems pose to failure logic modelling methods are two-fold.

Firstly, the failure logic of the components of reconfigurable systems cannot be adequately modelled in combinatorial terms (combinatorial techniques were inherited by the most prominent failure logic modelling methods from Fault Tree Analysis). The purpose of reconfiguration is often to remove detected threats from the system. However, some failures (e.g. short circuits or hydraulic leaks) may have effects on some components (e.g. power generators or pumps respectively) which persist even after the initiating failure is 'isolated' by reconfiguration. The effects of other failures may, of course, be transient and 'cleared' by a reconfiguration action. Combinatorial failure logic modelling methods, by definition, cannot differentiate between persistent and transient effects of failure. Furthermore, these methods do not allow for the capture of sufficient information – such as conditional exposure intervals of failures – as is necessary for accurate quantitative analysis of complex and reconfigurable systems.

Secondly, the reconfiguration logic cannot be attributed to single basic components of the failure logic models. This is a unique characteristic of the failure logic modelling approach which is not shared by some other approaches that fully integrate design and safety models. However, interactions between components in failure logic models are captured in terms of *deviations from intent.* In a reconfigurable system, different modes of operation will allocate different intents to components. For model behaviour to be consistent, all component characterisations must have access to a consistent and coordinated indication of the 'current intent'. System modes and their switchover logic must be captured at higher levels of model decomposition, thus forming the context of characterisations of basic components. Whilst the original definitions of failure logic modelling methods were careful not to make strong assertions about the feasibility of complete decomposition of the failure logic of the system to characterisations of individual elementary components, some of the later work (e.g. [60] and [116]) implied that such completeness was feasible. To a specialist reader, this assertion equates to a limitation of any 'pure' failure logic modelling technique.

It is important to note that the above discussion highlights a significant general characteristic of failure logic models: the dependency of component failure logic on the wider system configuration (and the associated system intent). This context-dependency challenges the compositionality and reuse claims often made for failure logic modelling. Challenges posed by modes and reconfiguration are just one aspect of this general problem.

Overall, these limitations of the current failure logic modelling methods, have resulted in a *perception* that the approach is *fundamentally inapplicable* to dynamically reconfigurable systems and, consequently, in a shift of focus (especially of the industrially-sponsored research) to different model-based safety assessment approaches. Therefore, the challenge posed by dynamic systems to the failure logic modelling framework must be addressed, in author's view, as a matter of priority.

This thesis addresses this challenge by extending the failure logic modelling approach with the concepts of "mode" and "mode space" and by specifying their relationship with pre-existing concepts. The thesis also shows that different types of reconfiguration are merely particular patterns of a mode space of system models and demonstrates the applicability of the extended approach.

1.2.2 Composition of Multiple Failure Logic Models

The complexity of the modern safety-critical system has another facet that is rarely addressed adequately by safety assessment methods – the complexity of the engineering process. In particular, modern large-scale systems are too complex to be developed as a 'single whole' or even by a single engineering organisation. Whilst large-scale system integrators typically maintain overall responsibility for the safety of complete engineered artefacts, these organisations are not necessarily well positioned to undertake assessment of all of the constituent subsystems that are engineered by external stakeholders in the context of complex supply chains and devolved design responsibilities.

At the same time, large-scale engineering artefacts are typically highly integrated – due not least to their reliance on shared infrastructures and computer-based controllers. This means that safety assessment responsibilities cannot be devolved fully, since a part-wise safety assessment will potentially be incomplete and unable to uncover all of the potential dangerous interactions between the constituent systems and subsystems.

Whilst compositionality of the safety assessment is an unrealistic objective, *composability of safety assessment artefacts* has been a declared goal for most model-based methodologies. The idealised model of such a "composable process" relies on engineering stakeholders developing 'safety models' of their artefacts and maintaining responsibility for the correctness of these models. The platform integrator is left with the responsibility of integration of such constituent models and analysis of the product of this composition.

Indeed, many failure logic modelling methods claim to be 'composable'. However, they all share assumptions about the context in which models must be composed which are likely to be valid

only in the context of monolithic system development and safety assessment. In particular, the composition of failure logic models typically relies on a consistency of interfaces between the models. It is (often implicitly) assumed that the platform integrator is capable of identifying such interfaces and of establishing their format and granularity *a priori*. This assumption is often justified by analogy with coordination between models in the design process (and, sometimes, with the notion of component-based engineering).

There is, however, a significant fundamental difference between modelling in design and safety assessment contexts. The purpose of design models is to *dictate* how systems are to be built and how they should be integrated together. These models are proactive 'blueprints' for further design and production. In contrast failure logic models capture a hypothesis about undesirable behaviour of the system; unlike design models, they are reactive. During construction of these models safety engineers do not simply capture (hypothesised) effects of undesirable events and interactions between systems – they *identify* what these interactions are. To require safety engineers to predict interfaces between failure logic models of different, independently-designed, (sub)systems is akin to requiring them to predict results of analysis before any assessment is undertaken.

Whilst an iterative approach to the definition of interfaces can *reduce* the impact of this problem, in general, the *composition of failure logic models cannot assume fully identified and harmonised interfaces between models of different engineering artefacts.* The challenge of composing failure logic models is further exacerbated by the need to protect the responsibilities of safety engineers over the correctness of the models they define. If a model integrator significantly modifies constituent models in order to harmonise or enrich interfaces, he risks undermining the responsibility (and, possibly, the liability) of the authors of the original models.

Finally, large-scale engineering artefacts may be decomposed into manageable parts in different ways. Some types of relationships between parts – such as relationships between systems and infrastructure or those between different views of the same system – do not yield an intuitive notion of an input-output interface in the context of failure logic models. This poses further challenges for model composition.

This thesis addresses these challenges by proposing an approach to the composition of independently-defined failure logic models in the absence of well-identified or harmonised interfaces. This approach is based on an extension of the notion of model interface and is set in a flexible conceptual framework which rationalises various forms of decomposition for complex engineering artefacts.

1.2.3 Conceptual Integrity and Language Independence

In addition to defining key safety engineering and modelling concepts, most failure logic modelling methods also introduce idiosyncratic languages and notations for the specification of models. Whilst such notations are a useful tool for early research and permit a focus on conceptual- rather than implementation- level issues, in practice they rarely reach the level of maturity necessary for industrial application or even evaluation[3]. Furthermore, the introduction of new notations, specific to a single engineering discipline (e.g. safety engineering), is often met with an understandable resistance in industry. Use of such notations implies significant cost in terms of training as well as in the development and maintenance of modelling and analysis tools. It also makes the engineering skills of the company significantly less flexible and complicates recruitment. Whilst these issues may appear superficial at first, they do represent *significant pragmatic concerns* of the industry. To facilitate adaptation of the model-based paradigm to safety assessment, researchers must recognise *industrial trends for the adaptation of standard specification languages* – such as SCADE, Simulink, StateMate and UML/SysML – across engineering disciplines.

From a more theoretical perspective, problems posed by idiosyncratic languages are symptomatic of a more serious issue: the lack of a clear demarcation between the modelling approach and the model specification language. Informally, the former is concerned with 'what' (addressing such questions as "What properties of the system should be captured?", "Which perspective is advantageous?", and "What is the semantic relationship between the key concepts of such viewpoint?"). The latter is concerned with 'how' (addressing such questions as "How can concepts of the modelling approach be represented within the syntax of the notation?", and "How is the relationship between the concepts supported by the semantics of the language?")

A strong coupling between the conceptual framework and the specification language – typical of current model-based approaches in general and failure logic modelling approaches in particular – poses numerous difficulties from both the research and the industrial perspective. It particular, it is difficult to differentiate the limitations of the conceptual approach taken by a particular method from the superficial constraints imposed by the format of the notation used. For example, HiP-HOPS [115, 117] does not permit the specification of situations in which component failure modes are caused by *combinations* of external and internal conditions. Instead, the tabular format of the notation always implies a *disjunction* between the external causes ("input deviation logic") of an output condition and causes internal to the component ("malfunction logic"). The *conceptual apparatus* of HiP-HOPS, however, does not require this restriction, which seems to be motivated merely by the prevalence of particular patterns of behaviour which the HiP-HOPS

[3] Out of the notations used for model-based safety assessment of which the author is aware, only HiP-HOPS has reached the level of maturity that *may* allow repeatable application to *some* industrial case studies.

author has observed during application of the method. In contrast, the “once failed, always failed” (OFAF) assumption and absence of an explicit notion of component state are fundamental to the HiP-HOPS method and are relied upon by the model analysis (the fault tree synthesis algorithm).

Furthermore, strong coupling between the notation (whether general or custom) and the conceptual modelling approach makes it difficult to transfer fundamental ‘lessons learned’ across different methods adapted by different researchers and therefore to evaluate the relative advantages and disadvantages of different conceptual approaches and their implicit viewpoints. When generic specification languages are used, the problem is sometimes even more acute since reports on the application of these languages often lack a consistent and clear definition of the methodology used to construct the models, and instead appear often to be driven by features of the particular language employed.

It is therefore a key challenge for the discipline of model-based safety assessment that it should not only utilise general-purpose and industrially-mature specification languages (and their associated tools), but should also de-couple the conceptual definition of modelling approaches from implementation considerations without undermining the consistency, coherence and principled definition of the former.

This thesis addresses this challenge by:

- ***defining a unifying Failure Logic Metamodel which defines key concepts of the approach and their interrelationships,***
- ***extending the Failure Logic Metamodel to address the key characteristics of industrial safety-critical systems in a conceptually justified fashion***
- ***demonstrating that the Metamodel (and all of the extensions defined in the thesis) can be instantiated using a general specification language (AltaRica) and utilising an industrially- mature third-party modelling and analysis tool (Cecilia OCAS suite)***

1.3 Motivation

Of course failure logic modelling is not the only model-based system safety analysis approach. Other researchers have sought to achieve a *complete integration* between safety and design processes by using *common models* in both processes. These models capture normal interactions between system components in terms of characteristics such as pressure, voltage or data. The models are, however, extended (either manually [20] or semi-automatically [24]) to reflect the *immediate effect* of failures on the input-output response of system components. Under this approach, the problem of system modes and reconfiguration is dealt with seamlessly. In fact,

provided that reconfiguration logic can be correctly modelled in system design models, modes of operation are irrelevant to the safety analysis tasks in the context of these methods.

However, system safety assessment based on integrated models typically only addresses propagation of failure effects through intentional interactions identified in design models. With this restriction multiple models can typically be composed through predefined interfaces. The restriction, however, highlights the key limitation of safety assessment approaches based on integrated models – an inability to address unintentional interactions between components and sub-systems. However, as it stands now, this limitation can only be traded off against the perceived *inapplicability* of failure logic modelling.

As outlined in the previous section, this thesis demonstrates how key pragmatic challenges posed by modern safety critical systems and their engineering processes can be addressed within the FLM Framework. However, it is important to stress that the thesis does not contend that failure logic methods are universally superior to other model based approaches. On contrary, the author strongly believes that most of the existing approaches have relative strengths and weaknesses and that the decision as to which approach is most appropriate and most effective in a particular context will remain complex. However, it is also believed that this decision should be *well-informed and objective*; the need to provide the rational basis for such decisions motivates the present author's research.

1.4 Thesis Proposition

Based on the challenges presented in the previous sections, and in the context of the motivation indicated there, the key proposition defended in this thesis is as follows:

It is possible to provide a well-defined, sound and pragmatic framework for the failure logic modelling of realistic systems which tackles the problems of dynamic reconfiguration of systems and composition of interdependent system models.

1.5 Thesis Structure

The remaining chapters of this thesis are organised as follows:

- ***Chapter 2*** defines the context of the research by discussing the role of the assessment in the overall safety engineering process. It also presents an overview of key traditional and model-based safety analysis methods, highlighting their key principles, strengths and limitations. The chapter also proposes a classification of model-based safety analysis

methods and reviews two specification languages than have been widely applied in this context: AltaRica and AADL Error Model Annex.

- ***Chapter 3*** presents the Failure Logic Metamodel (FLMM) which unifies and subsumes existing failure logic modelling methods. The metamodel also extends most of the existing techniques to allow the dynamic behaviour of system components to be modelled. Whilst the metamodel is defined in a language-independent format, this chapter also demonstrates that it can be instantiated within a third-party specification language (AltaRica). In the context of the thesis as a whole, the chapter defines a "baseline approach" which is extended by next two chapters.
- ***Chapter 4*** proposes a flexible framework for rationalising different types of decomposition of large-scale engineering artefacts (platforms) into manageable parts (engineering domains). It proposes an extension to the Metamodel which allows composition of Domain-Specific Failure Logic Models (DSFMs) through the notions of "external failure causes", "translation components" and the "translation layer".
- ***Chapter 5*** addresses the challenge posed by reconfigurable systems by further extending the FLMM with the concepts of "mode" and "mode space". This chapter demonstrates that different types of reconfigurations (such as phased operation and failure-handling modes), as well as architectural limitations of systems, can be seen as particular patterns of mode models.
- ***Chapter 6*** presents an evaluation of the FLM Framework proposed in Chapter 3-5. It summarises the evidence obtained through peer review and case studies. The chapter also demonstrates that the metamodel underlying the framework subsumes existing failure logic modelling techniques and can be related to classical techniques such as Fault Tree Analysis (including its dynamic and non-coherent extensions). This chapter concludes with an outline of the limitations of the failure logic modelling approach identified during the evaluation along with an outline of the pragmatic mitigations that were adopted.
- ***Chapter 7*** concludes the thesis by summarising its key contributions and discussing some promising directions for further research.

The thesis is supplemented by five appendices. ***Appendix A*** presents a consolidated overview of the FLM Framework as developed in Chapters 3-5 and formalised in the Eclipse Modelling Framework; ***Appendices B and C*** present descriptions of the case studies discussed in Chapters Three and Four: the Aircraft Wheel Braking System (WBS) and the composition of models of the WBS and Common Computation/Communications Platform respectively. Finally, ***Appendix D*** presents a summary of a review of the Aircraft Fuel System that was carried out as part of the evaluation process and informed selection of the case studies.

Chapter 2: Literature Survey

This chapter clarifies the scope of the research presented in this thesis, and establishes a context and a baseline for it, by presenting a survey of relevant literature. The chapter is organised in five main sections. Firstly, *key terms* relevant to system safety and the processes and activities used to achieve it are defined (predominantly based on industrial and regulatory standards and on some of the key academic texts in the discipline).

Secondly, some of the most prominent *'classical' safety assessment techniques* are described and discussed. This section focuses (although not exclusively) on the three techniques introduced in Chapter 1. As well as defining the context (and, to some extent, a 'benchmark') for any novel techniques, this review of the traditional techniques aims at identifying key characteristics which may have contributed to their popularity and (perceived) strengths.

Thirdly, failure logic modelling techniques, including HiP-HOPS [114] and FPTN [57] as well as other, more recent, methods are presented. The research presented in this thesis unifies and builds upon these techniques.

Fourthly, other alternative *model based approaches to safety assessment* are discussed in section 2.4. Over the past fifteen years, this area has attracted significant interest from academia and industry alike; however, until now little work has been done on the classification of the emerging techniques. The section proposes a preliminary (and basic) *classification* identifying two broad approaches (in addition to failure logic modelling). These approaches are tentatively referred to as '*failure injection*' and '*failure effects modelling'*. The reviewed techniques are variously classified as being one, or the other or some form of '*hybrid approach*'.

Finally, the chapter reviews some *specification languages* that can be used to support model-based safety assessment in general and failure logic modelling in particular. Whilst the section focuses specifically on two languages – AltaRica Dataflow and AADL Error Modelling Annex – it is important to recognise that *throughout this thesis it is argued that a model-based safety assessment method and a specification language are separate (and sometimes orthogonal) entities*. The selection of the method should *not* limit the safety engineers to a particular language and, often, vice versa.

2.1 Safety Engineering, Assessment and Terminology

This section briefly outlines the key definitions of concepts in system safety engineering, typical safety engineering process as well as the role and types of assessments that underpin safety engineering.

2.1.1 Key Terminology of System Safety

Although system safety has emerged as a distinct engineering discipline over the last 50 years, definitions of some key safety terms differ and are, at times, disputed. The disagreement is particularly severe for concepts applied to software and software-intensive systems [106]. For example, van der Meulen lists eleven different definitions of the term "safety" itself [156]. To avoid ambiguity, this section lists the definitions of key safety terms which are assumed in this thesis.

Safety is defined as "*freedom from unacceptable risk*" [72, 109] with **risk** (or safety risk) being defined as "*combination of the likelihood of harm and the severity of that harm*" [149] (with nearly identical definition in [72]). In this context, **harm** is often interpreted broadly along the lines of: "*Death, physical injury or damage to the health of people, or damage to property or to the environment.*" [149] (although [72], for example, takes a more restricted view).

Causally related to harm are the notions of an **accident**[4]: "*an unintended event, or sequence of events, that causes harm*" [149] – and an **incident**: "*the occurrence of a hazard that might have progressed to an accident*". Which, in turn, call upon the key concept of a **hazard**: a "*physical situation or state of a system, often following some initiating event, that may lead to an accident*" [149].

The concept of the hazard (and its potential severity) is the key to distinguishing between two system attributes which are sometimes confused: safety and reliability. Reliability is defined as the "*ability of an entity to perform a required function under given conditions for a given time interval*" [160] and is different from safety in that it considers all possible failures of a system, whereas safety (and safety engineering) is concerned only with the failures that may cause a hazard, and by extension, an accident. The other source of confusion arises from the term "required function" which can be interpreted as "original" (as at the time of commissioning), "designed", "specified" or "intended". The 'correct interpretation' (especially in the context of definitions of terms "fault" and "failure") has given rise to numerous and long-standing disputes between safety researchers (especially between the USA and the UK), as described by Pumfrey

[4] In the USA this is typically referred to as a '*mishap*' [154]

[123]. *For the purpose of this thesis, "required function" and "failure" are always interpreted with respect to the intent*.

This thesis is concerned with the safety of systems in general and of complex, reconfigurable and software-intensive systems in particular. The current version of UK Defence Standard 00-56 defines a **system** as: "A combination, with defined boundaries, of elements that are used together in a defined operating environment to perform a given task or [to] achieve a specific purpose. The elements may include personnel, procedures, materials, tools, equipment, facilities, services and/or software as appropriate" [149]. In the context of this thesis, this definition is too wide and, whilst the author does not dispute the importance of the human, the organisation and the procedures in a system context, the definition used in an older issue of the DS 00-56 is adopted in scoping the research presented here: "a *bounded physical entity that achieves in its domain a defined objective through the interaction of its parts*" [147].

2.1.2 System Safety Engineering, Assessment and Lifecycle

Whilst key safety terms frequently have several varying definitions, concepts relevant to the System Safety Engineering (or just 'System Safety' [49, 90]) process tend to suffer from the opposite problem – largely identical concepts are given different names. Furthermore, different system safety standards and key texts tend to emphasise different aspects of the process.

In general **System Safety** is concerned with "*The application of engineering and management principles, criteria, and techniques to achieve acceptable mishap risk*[5]*, within the constraints of operational effectiveness, time, and cost, throughout all phases of the system life cycle*" [154]. Leveson lists key principles of system safety, among them the statement that: "*system safety emphasizes analysis rather than past experience and standards*" [90]. Whilst most texts broadly agree with this definition and principle, further 'decomposition' of what is meant by 'analysis' differs significantly. Part of the problem is that safety engineering is a 'spiral' process that iterates as the underlying system development process progresses. At each iteration the safety engineering (or, alternatively, "safety management") process proceeds through a broadly identical set of key activities (although the nature of individual activities and their degree of detail and rigour may differ from one iteration to the next). The results of this process are typically captured in an evolving hazard log, which can be described as "*the continually updated record of the hazards, accident sequences and accidents associated with a system. It includes information documenting risk management for each hazard and accident*" [149]– although different industrial sectors and standardisation/certification jurisdictions may use different terms for the same concept. For

[5] In context of the definitions presented in the previous section "*mishap risk*" should be read as simply "risk" or, alternatively, as "*the risk posed by hazards associated with the system*"

example, European Civil Aircraft manufacturers sometimes refer to this as “aircraft safety synthesis” [89].

The current issue of the Defence Standard 00-56 (part 2) [150] emphasises and lists typical activities at each iteration:

a) Hazard Identification
b) Hazard Analysis
c) Risk Estimation
d) Risk Evaluation
e) Risk Reduction
f) Risk Acceptance

The key activity, from the perspective of this thesis, is the *Hazard Analysis*, which is concerned with the identification of causal factors of hazards which are posed by the system (as generated by the *Hazard Identification* activity), establishment of a model (whether an explicit model or an implicit ‘mental model’) of relationships between those factors and estimation of the likelihood of a hazard. It is important to note that Hazard Analysis can be used either *reactively* (e.g. to assess the probability of a hazard based on concrete component data and an implemented design towards the end of the development process) or *proactively* (e.g. at earlier iterations to assess the architecture of the system, to allocate necessary requirements to components and to assess the feasibility of the proposed system design). Hazard Analysis feeds into *Risk Estimation*, which combines the estimated likelihood of hazards with their severity (obtained by a domain-specific assessment of interactions between the system, its environment and its operations) to obtain characterisations of risks posed by the system. The last three activities in the list are concerned with an engineering and managerial judgement on whether such risks are acceptable and/or whether any additional measures for their reduction are necessary (and feasible): the latter may include redesign of the architecture (or, at the extreme, redefinition of the system concept), reallocation or other changes to the derived requirements as well as additions or changes to operational procedures (including training, inspection and maintenance procedures as well as monitoring programmes).

It is important to note that, whilst some analysis techniques can be uniquely mapped to the individual activities, others cover more than one. For example, Functional Failure Analysis (FFA) is a Hazard Identification method and Fault Tree Analysis is a Hazard Analysis method; by contrast, Hazard and Operability Studies (HAZOP) and, to a lesser extent, Failure Modes Effects and Criticality Analysis (FMECA) can be used for both Hazard Identification *and* Hazard Analysis.

As was said above, other key texts on safety engineering emphasise the chronological aspect of iterations. Leveson, for example, identifies four such stages [90] (confusingly, but in keeping with the lack of consensus described above, some of those stages bear names similar to those of the 00-56 activities described above):

- Preliminary Hazard Analysis (PHA)
- System Hazard Analysis (SHA)
- Subsystem Hazard Analysis (SSHA)
- Operating and Support Hazard Analysis (OSHA)

Whilst each stage typically contains each element of the activities in the previous list, the emphasis gradually moves from Hazard Identification to Hazard Analysis and Risk Estimation (with Risk Evaluation, Risk Reduction and Risk Acceptance underpinning all stages) as the stages progress. Ericson proposes a more detailed list, identifying seven iterations for conceptual design, preliminary design, detailed design, system design, operations design, health design and, finally, requirements design [49].

More simple, and more concrete, 'chronological' conceptualisations of the stages and activities of the safety assessment process can be often found in industry-specific process standards and guidance [55, 109, 140, 139, 153]. For example, in civil aerospace, 'recommended practice' documents – ARP 4754 [140] and ARP 4761 [139] – define three key stages of assessment (see also Figure 5):

> **Functional Hazard Assessment (FHA)**: "*A systematic, comprehensive examination of aircraft functions to identify and classify Failure Conditions*[6] *of those functions according to their severity*" [140]. FHA is performed at both aircraft and system levels, and is conducted by means of Functional Failure Analysis [139]; the results between aircraft- and system-level assessments are typically captured in a fault tree format ("aircraft fault tree analysis") although no actual fault tree analysis (as defined in [157]) is being conducted (see sections 2.2.3 below)
>
> **Preliminary System Safety Assessment (PSSA)**: "*A systematic evaluation of a proposed system architecture and its implementation, based on the Functional Hazard Assessment and failure condition classification, to determine safety requirements for all items in the architecture*" [140].
>
> **System Safety Assessment (SSA)**: "*A systematic, comprehensive evaluation of the implemented system to show that the relevant safety requirements are met*" [140].

[6] **Failure Condition** is defined as "*a condition with an effect on the aircraft and its occupants, both direct and consequential caused or contributed to by one or more failures considering relevant adverse operational or environmental conditions*". This can be seen (broadly) as a specialisation of the concept of "*hazard*".

Because of the 'divide and conquer' and system-centred nature of the ARP process the three stages defined above are supported by *Common Cause Analysis* – comprising zonal, particular risk and common mode analyses – the objective of which is to verify the presumed (or explicitly required) independence between aircraft functions, systems or lower-level items [140, 139].

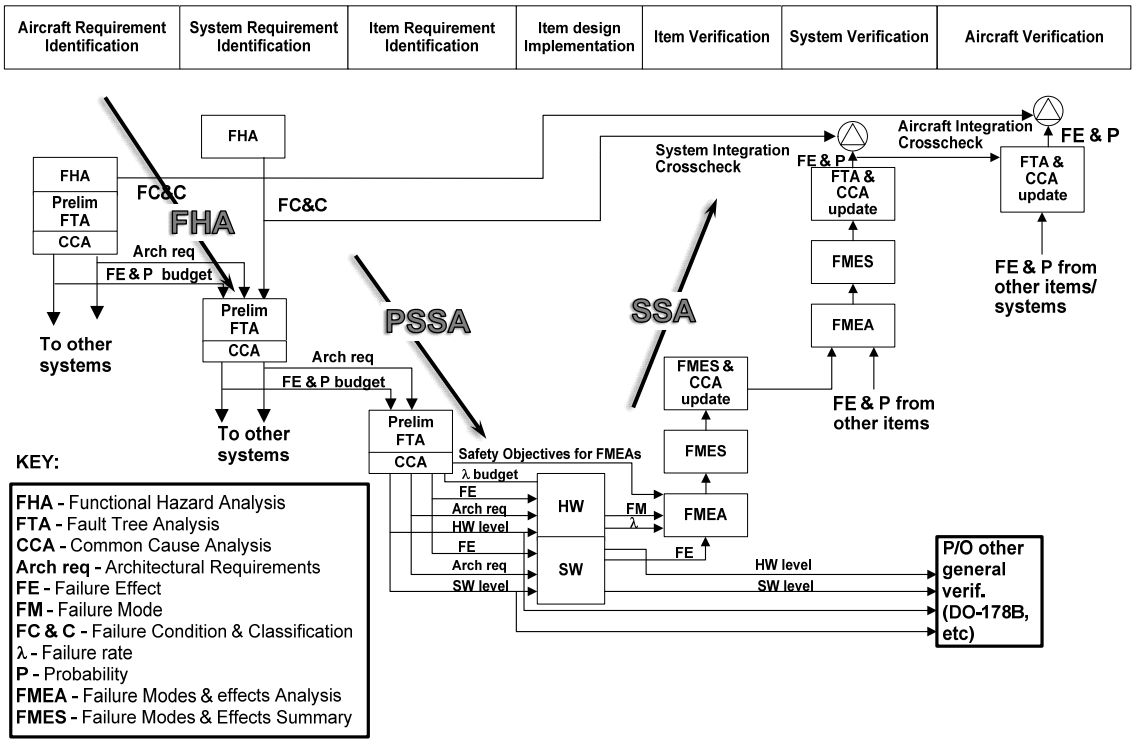


Figure 5 - ARP4761 Safety Assessment Diagram[139]

It is important to note that, from a narrow technical perspective, PSSA and SSA are similar processes and are both conducted by means of Fault Tree Analysis, Reliability Block Diagrams, Markov Analysis or similar techniques[7]. The two assessments, however, differ in their objectives: whilst SSA is confirmatory analysis, performed on the basis of 'real' component data and system design *as implemented*, PSSA is a [development process] "*risk reducing and value adding phase*" of the ARP process, with the primary objective of *driving* the design of an aircraft system [40]. The latter is also envisaged as an *iterative* activity that is refined as the design of the system matures.

In terms of the activities of the DefStan 00-56 (listed above), both PSSA and SSA are predominantly Hazard Analyses; FHA is mainly Hazard Identification but also covers some aspects of Risk Estimation (in that the severity of failure conditions is determined). Overall, in civil aviation risk-related activities are simplified. The reason is that the acceptable probabilities of failure conditions of a given severity (and the severity classification schema) are prescribed by the overarching EASA and FAA airworthiness standard [53] and its associated "advisory

[7] It should be noted, however, that [139] suggests using FTA for PSSA and FMEA for the SSA.

material". In this context, any judgement about the acceptability of risk is essentially made by the certification authority, rather than by an aircraft's manufacturers.

2.1.3 Scope of the Present Research

In terms of 00-56 activities [150], the scope of the research presented in this thesis is Hazard Analysis (although subsequent chapters focus predominantly on the qualitative aspects of the assessment). In Leveson's terms [90] it is both System Hazard Analysis and Subsystem Hazard Analysis. In terms of the ARP process [140, 139] PSSA, SSA and corresponding iterations of the CCA all fall within the scope of the current work; however, the main focus is on the PSSA and especially its early iterations, when less detailed, immature and unstable system design proposals are being assessed and, consequently, when there is a significant opportunity (currently under-utilised in the industrial context [40]) to affect the design in an cost-, time- and effort- effective fashion.

In the remainder of the thesis, these activities and stages of the safety engineering process are collectively referred to as *system safety assessment*.

It should be noted that the research presented here is not specific to the civil aviation domain, even though the thesis refers to terms established by the ARPs (such as PSSA, SSA, CCA and Failure Condition) and uses case studies from that domain. Standards and guidance material from other domains use concepts akin to the ARP. In some cases, (e.g. Eurocontrol Safety Assessment Methodology [55], MISRA Guidelines for Safety Analysis of Vehicle Based Programmable Systems [109], and – currently superseded – US MilStd-882C [153]), these concepts are made explicit, whilst elsewhere, (e.g. UK Defence Standard 00-56 [149, 150]), they remain implicit. The selection of case studies for this thesis has been influenced by the author's collaboration with European aircraft manufacturers and suppliers. Furthermore, systems in the aerospace domain typically exhibit characteristics that are investigated in this thesis: dynamic reconfiguration, multi-phase operation and the composition of (relatively) independently designed artefacts.

2.2 Classical Safety Assessment Methods

This section reviews 'classical' and well-established safety analysis techniques, focussing on Failure Mode Effects and Criticality Analysis (FMECA), Fault Tree Analysis (FTA) and Hazard and Operability Studies (HAZOP).

2.2.1 Inductive Methods

Inductive, or forward search, analysis methods progress from a known or hypothesised cause (typically – a component failure or a failure mode) to identify general system-level effects (i.e. a system-level failure condition or even a hazard). Therefore, many inductive methods, even when they are focussed on system safety analysis, are capable of contributing to the identification of hazards (or to the verification of the completeness of the previously identified hazards).

FMEA / FMECA

Failure Mode, Effects and Criticality Analysis (FMECA) along with its more restricted and reliability-centred variant – Failure Modes and Effects Analysis (FMEA) – are the most widely known inductive safety analysis methods. Originally specified in 1949 as US Military Procedure MIL-P-1629, FMECA has been described in a number of industrial standards, including MIL-STD-1629A[152], IEC60812 [71]and SAE J-1739 [137], and also features in most of the key academic texts on dependability engineering (e.g. [90, 160]).

The objective of FMECA is to exhaustively identify failures of components[8] and their effects on the system. The effects are typically classified according to their severity and probability (i.e. *risk*, although this is typically called 'criticality' in FMECA), which allows prioritisation of the design actions such as re-design or the development of protective barriers or mitigation procedures. The probability of the failure and its effect can be determined either in quantitative or in qualitative form. The severity of the system-level effect can either be determined as part of the FMECA itself [152] (in this case the analysis combines Hazard Identification, System Safety Assessment and Risk Assessment/Consolidation objectives) or linked to the findings of earlier Hazard Identification activities [139].

Whilst the details of the analysis procedure differ between many descriptions of FMECA and FMEA, all procedures reflect four general steps [160]:

1. The definition of the system, its functions and components
2. The identification of the component failure modes and their [immediate] causes
3. The study of the failure modes' effects
4. Conclusions and recommendations

For safety-centred FMECA, the third step can be subdivided into identification of the effects of the failure modes and a detailed criticality analysis (e.g. task 102 in [152]).

[8] Or, to be more precise, failure modes, typically defined as "*The manner by which a failure is observed. Generally describes the way the failure occurs and its impact on equipment operation*" [152].

All FMEA/FMECA standards and descriptions typically prescribe a particular tabular format for reporting analysis results. For example, the original method definition [152] prescribes twelve columns which must all be filled.

Whilst all FMEA/FMECA standards (and many descriptions of the technique) focus on the procedural aspects of the analysis, they often contain significant guidance to facilitate identification of the failure modes and their effects. For example, [152] above states that in identifying failure modes and causes at least the following scenarios must be considered:

a. Premature operation
b. Failure to operate at the prescribed time
c. Intermittent operation
d. Failure to cease operation at the prescribed time
e. Loss of output or failure during operation
f. Degraded output or operational capability

Such lists act essentially as guide words with the goal of ensuring completeness of analysis. Descriptions of FMECA rarely include step-by-step guidance on the systematic identification of the effects of failure. Indeed, given the historical context in which this analysis method has emerged, FMECA typically assumes the existence of relatively simple self-evident (to system and safety engineers) causal dependencies within the system. In other words, the method (implicitly) assumes that the key challenge for safety assessment is completeness (i.e. exhaustive identification of components failure modes, modes of operation and effects) rather than the complexity of system behaviour.

Whilst it is typically applied at the level of relatively detailed design and to concrete known 'initiating' failure modes, FMEA/FMECA can be applied at conceptual stages with hypothesised failure modes. For example, Software FMEA [120] identifies the effects of different *hypothetical* deviations in software behaviour and thus helps to focus and prioritise future development as well as validation and verification activities. Another example is the application of FMECA to the whole system as a 'black box', described in terms of top-level functions. This is known as Functional Failure Analysis and is a prevalent hazard identification technique in the aviation sector [139].

Other Inductive Methods

Although it is perhaps the most prominent inductive analysis technique, FMEA/FMECA is not the only one. Event Tree Analysis [155, 160] is a graphical method that starts with a single initiating event and systematically examines its effects under different scenarios. ETA originated in the Nuclear Power Industry (although it is based on the decision theory of Economics [160]), and is best suited for investigating the effectiveness of well-defined protective barriers. A typical

initiating event is clearly potentially hazardous (e.g. critical over-pressure conditions in the plant, radiation or hazardous chemical leaks), and scenarios are described in terms of successful or unsuccessful operation of successive protection mechanisms (either implemented in plant design or in operational procedures). In particular, ETA is unlikely to be suitable for investigation of effects of the type of 'deeply nested' failure modes of system components that are addressed by FMECA. For this reason it is typically applied in conjunction with Fault Tree Analysis [155].

2.2.2 Deductive Methods

In contrast to inductive safety analyses, where engineers typically start with a known (or assumed) failure mode of the component and *hypothesise* possible system-level outcomes (based on domain knowledge and system descriptions), deductive analyses proceed in the reverse logical direction: starting with a specific undesirable condition defined at the system level (such as a hazard or a failure condition) analysts identify its possible causes systematically. The procedure is typically recursive, whereby analysts first consider the immediate necessary and sufficient causes of the undesirable condition; these identified causes are then themselves considered undesirable conditions and their causes are identified. The recursion continues until a level judged as elementary is reached; this is typically a level at which conditions, such as component failures, can be considered either to be random in nature or to be sufficiently implausible, or at which a boundary of the system has been reached.

Fault Tree Analysis: Introduction and Syntax

The most widely used deductive safety analysis method is Fault Tree Analysis (FTA) [158, 157, 160]. Fault Trees are graphical models of a system which show how low-level conditions gradually combine to cause an undesired *Top Level Event* (TLE). Syntactically, Fault Trees consist of events connected by logic gates. Most of the events in the tree are *Intermediate Events* which are further decomposed into other events. The leaves of the tree are Basic, Conditioning, External (or "House") and Undeveloped Events (see Figure 6). Each gate of the tree specifies a condition over input events (such as logical conjunction or disjunction) that must hold and must be sufficient for the higher level event to occur. Figure 7 shows standard FTA gates (adopted from [157]).

Essentially, each fault tree encapsulates a *hypothesis of system behaviour in 'failure space'* [157]; furthermore, the hypothesis is partial since any tree only captures behaviour relevant to a single top-level event. A fault tree can be analysed to generate a set of Minimal Cut Sets, a quantitative measure of the probability of the top level (and/or intermediate) event or an importance measure of the individual basic events. All such analyses essentially reduce the hypothesis (or a model) to more simple propositions (or views). The graphical notation of fault trees discussed above provides *syntax for expressing the hypothesis*.

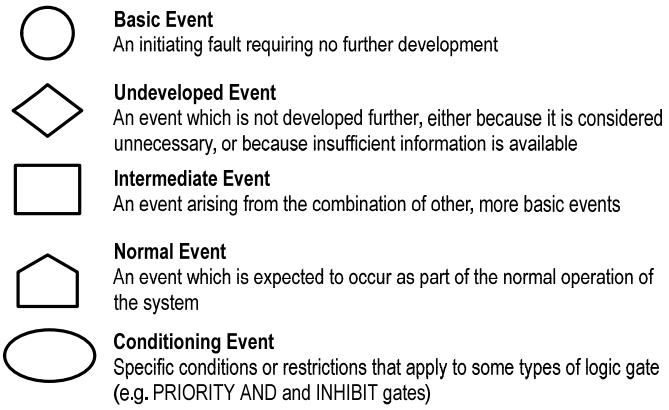


Figure 6 - Principal FTA Event Symbols

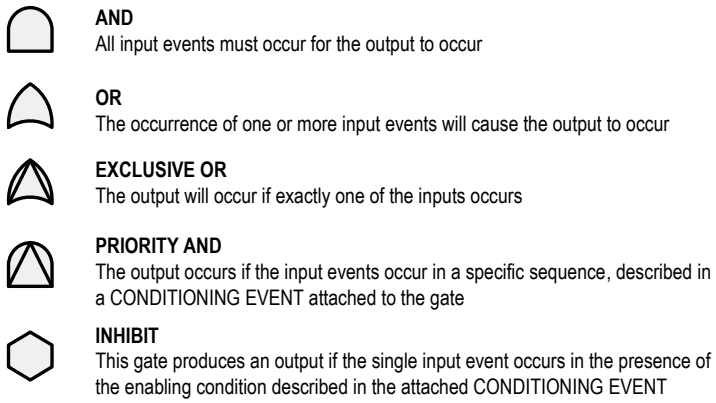


Figure 7 - Standard FTA Logic Gate Symbols

FTA Construction Rules and Principles

However, it is important to stress that Fault Tree Analysis covers *more* than merely the syntax of the models and algorithms for their analysis / traversal (although some FTA descriptions – such as Appendix D of [139] – are, indeed, limited to these aspects). In particular, it includes *rules, key principles and significant guidance for the construction of fault trees*. The importance of this aspect of FTA cannot be overstated; by providing a systematic and repeatable method for system assessment and model construction, FTA not only *facilitates the assessment process* but also provides a degree of *confidence in the validity of the hypothesis* encapsulated in the Fault Tree itself.

The earliest, and still most authoritative, description of FTA [157] lists two ground rules for the construction of fault trees:

I. *"Write statements that are entered in event boxes as faults; state precisely what the fault is and when it occurs";*
II. *"If the answer to the question, 'Can this fault consist of a component failure' is 'Yes', classify the event as 'state-of-component fault'. If the answer is 'No', classify the system as a 'state-of-system fault'".*

The second rule introduces two new semantic concepts of FTA that are not explicitly represented in the notation's syntax. Whilst it may appear that the first rule is primarily syntactical, in fact it ensures the conceptual consistency of the fault tree models and imposes a constraint of describing behaviour strictly in the failure space.

In addition to the two "ground rules" the FTA Handbook [157] lists three procedural statements:

> *The* ***Complete-the-Gate Rule*** which structures the system analysis procedure by mandating a breadth-first fault tree development;
>
> *The* ***No Gate-to-Gate Rule*** which restricts fault tree notation syntax by expressly prohibiting direct connections between the gates;
>
> *The* ***No Miracles Rule*** that states: "*If the normal functioning of a component propagates a fault sequence, then it is assumed that the component functions normally*" [157].

Whilst the last rule effectively prohibits negation and the 'not' operator is not included in the original FTA definition, it has been argued that the failure logic of certain classes of systems (referred to as "non-coherent") can only be described accurately if negation is permitted [10, 11, 37, 75].

In addition to the above rules, the FTA Handbook defines some additional concepts such as "*Immediate Cause*" – discussed at the beginning of this section – and classification of the causes of state-of-component events into *primary*, *secondary* and *command faults*. Under this classification, primary and secondary faults are essentially failures of a component in its intended and unintended environments respectively whereas the command fault "involves the proper operation of the component but at the wrong time or in the wrong place" [157]. In practice, these three classes of faults are used by safety engineers as *guide words* to aid identification of the causes of events and to assure the completeness of the fault tree. Villemeur illustrates and refines these three broad classes as shown in Figure 8.

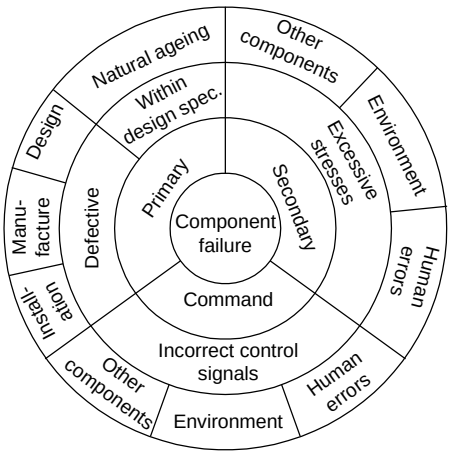


Figure 8 - Villemeur's 'Failure Classification as to Causes' [160]

FTA: Dynamic Extensions

'Classical' Fault Tree Analysis is often perceived to be inappropriate for the assessment of complex reconfigurable and software-intensive systems [90]. Indeed the FTA notation is predominantly based on combinatorial logical operators which do not allow for the expression of timing or sequencing behaviour. The only non-combinatorial gate defined in [157] is 'Priority And' (PAND), however, its semantics are not precisely defined [161]. A further constraint is imposed by the assumption of equivalence between events (fault occurrence) and states (fault existence) which underlies Fault Tree Analysis (see, for example, section V.2 of [157]).

To alleviate these problems, a number of extensions to FT syntax have been proposed (e.g. [44, 79, 113, 161]). The detailed description of all proposed alternative extensions is beyond the scope of this thesis. However, two of the most prominent approaches – *State/Event Fault Trees* (SEFTs) and *Dynamic Fault Trees* (DFTs) are briefly discussed below.

The first approach, proposed by Kaiser and Gramlich [79, 80] revokes the state-event equivalence assumption of traditional fault trees. Consequently SEFTs refine the semantics of the fault tree gates and introduce some new gates which distinguish between the notions of events and states on their inputs and outputs. The semantics of SEFT is defined through representation in Deterministic and Stochastic Petri Nets (DSPN).

Probably the most prominent of the extensions to Fault Trees, Dugan's Dynamic Fault Trees (DFT) approach [12, 44, 158] proposes two new gates – a *Functional Dependency* (FDEP) gate and a *Cold Spare* (Spare) gate (Figure 9, (a) and (b) respectively).

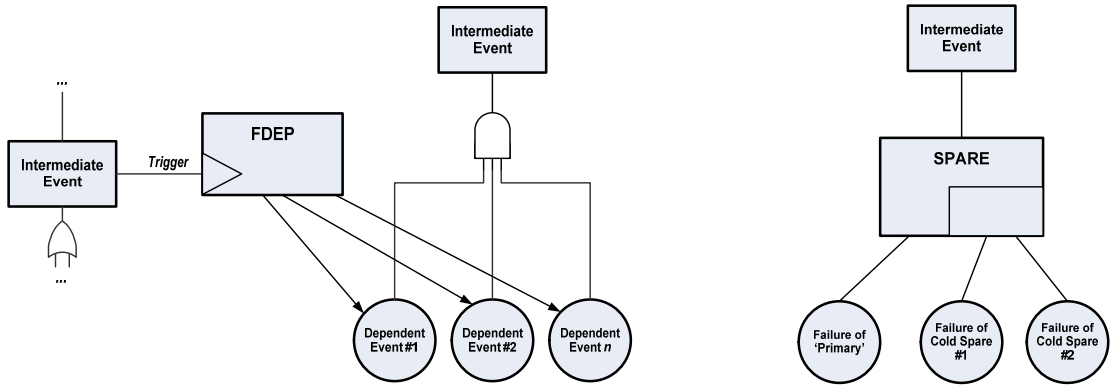


Figure 9 - Dynamic Fault Tree Gates: Functional Dependency (a), Cold Spare (b)

The Functional Dependency gate allows for the expression of common causes of (otherwise seemingly independent) basic events in a fault tree. The gate connects a number of basic events in the tree and a trigger event (which can be either a basic event or an intermediate/top event in the

hierarchical tree). When the trigger occurs, all of the basic events connected to FDEP are forced to occur.

A *Cold Spare* gate is connected to a number of basic events and an intermediate event. One of the basic events is identified as the "primary event", and a sequence between other basic events ("spares") is defined. The gate has two effects:

- It acts as an *AND* gate
- It indicates that, until the primary (and any spares with a higher priority) has failed, the failure rate of a cold spare may vary. This allows for more accurate modelling of the fact that certain stand-by components (e.g. emergency electrical generators) have different failure probabilities, depending on whether they are activated or not. The failure rate of each inactive cold spare is a product of its "hot" failure rate and the dormancy factor (held by the *Spare gate*). If the dormancy factor is one the spare is always hot[9], if it is zero – the spare is cold (cannot fail until it is used); otherwise a spare is considered "warm".

A common feature of both DFT and SEFT approaches is that they provide a 'user friendly' means of access to the underlying more powerful non-combinatorial formalism (Markov models and DSPN respectively) through a set of new or refined fault tree gates. Each of these refined gates essentially captures a particular *pattern* in the underlying formalism that has been adjudged to be frequent or prominent in modern safety-critical systems.

Other Deductive Methods

Whilst Fault Tree Analysis has dominated the deductive approaches 'market', other methods have been proposed. Closely related to fault trees, Reliability Block Diagrams (RBDs) [160] have been quite popular for some time, especially in the European Aeronautics Sector. The expressive power of RBDs is broadly similar to that of Fault Trees [101]; however, the models capture system logic from the 'success space' perspective[10].

Finally, Ladkin's Why Because Analysis (WBA) [88] is another deductive method which relies on a systematic construction of causal dependency models with well-defined semantics. However, WBA is predominantly used by for accident investigations rather than system safety assessment.

[9] Consequently a spare gate with all dormancy factors set at one is equivalent to an AND gate.

[10] In fact Villemeur refers to RBDs as "Success Diagram Method" [160]

2.2.3 'Bowtie' Methods

Hazard and Operability Studies (HAZOP)

Deductive and inductive methods have different strengths, and their combination has been long recognised as beneficial [155]. Consequently, analysis methods combining the two approaches have emerged. The most prominent of such methods is Hazard and Operability Studies (HAZOP) which emerged from the chemical process industry in the mid-1960s and has since then became quite popular across the process sector in general (including the nuclear, chemical and food-processing domains) [85].

HAZOP is essentially a facilitated and structured brainstorming activity that covers both Hazard Identification and System Safety Assessment. The method is different from most other safety assessment techniques in that it is expressly team-based. Consequently HAZOP descriptions typically dedicate at least as much attention to the composition of the assessment teams and procedural aspects of the analysis as they do to the more 'technical' guidance. For example, in the context of the safety assessment of new chemical process plants, Kletz advocates the following HAZOP team composition [85]:

- Project or design engineer
- Process Engineer
- Commissioning manager
- Control system design engineer
- Research chemist
- Independent team leader

Technically, HAZOP is typically applied to piping and instrumentation (P&I) diagrams of the proposed plant design. Centred around the concept of "guide word", the assessment proceeds by considering every flow in the plant (material, energy and/or control) iteratively, using a standard set of guidewords (Table 1) to identify possible deviations of a flow's physical characteristics (such as pressure or temperature) from that intended by the designers. For each viable deviation, the assessment team walks through the diagram deductively and inductively to identify causes (e.g. failures) and ultimate effects (i.e. hazards) of the deviation respectively. The deductive and inductive parts of the assessment can be seen as informal or 'lightweight' Fault and Event Tree Analyses respectively. However, in HAZOP these analyses are not bound by a particular prescriptive procedure or notation – the strength of the method lies in allowing the team to deliberate and investigate deviations freely (achieving what is sometimes referred to as a "system of *imaginative anticipation* of hazards" [105]) . The results of the study are typically reported in a tabular format, although HAZOP descriptions tend to be less prescriptive on the exact format of such tables than FMEA/FMECA standards.

Table 1 - HAZOP Guidewords [90]

Guideword	Meaning
NO, NOT, NONE	The intended result is not achieved, but nothing else happens (such as no forward flow when there should be one)
MORE	More of any physical property than there should be (such as higher pressure, higher temperature, higher flow, or higher viscosity)
LESS	Less of a relevant physical property than there should be
AS WELL AS	An activity occurs in addition to what was intended, or more components[11] are present in the system than should be there (such as extra vapours or solids or impurities, including air, water, acids, corrosive products)
PART OF	Only some of the design intentions are achieved (such as one of two components in a mixture)
REVERSE	The logical opposite of what was intended occurs (such as backflow instead of forward flow)
OTHER THAN	No part of the intended result is achieved, and something completely different happens (such as the flow of the wrong material)

Software HAZOPs

From the perspective of the research presented in this thesis, the key features of the HAZOP are the *focus on flows* rather than components of the plant design along with the concepts of *deviation from intent* and *guide words* which aid in the identification of such deviations. In his work on a software-oriented variant of HAZOP – SHARD – , Pumfrey [105, 123] maintains all three aspects, but notes that standard guide words do not provide adequate coverage of issues of timing and sequencing that are often vital to the safety of control software. Consequently SHARD is based on six generic[12] deviation "classes": Omission", "Commission", "Too High Value", "Too Low Value", "Too Early" and "Too Late" – applied to information flows in software. These classes, which are often grouped in pairs by three 'domains' of service *provision*, service *value* and service *timing* respectively [105], formed the basis of all of the failure logic modelling methods described in section 2.3 below.

UK Defence Standard 00-58 (now obsolete) addressed the application of HAZOP to "systems containing programmable electronics" [146]. It makes a similar observation with respect to timing deviations. However, the Standard's approach is to retain generic HAZOP guide words but extend the list with two pairs of timing and sequencing keywords: "Early" and "Late", "Before" and "After". Part 2 of the standard contains detailed guidance on the application of the guide words. It is interesting to note that the 00-58 guide words are applied not only to the information flows, but

[11] i.e. component substances (does not refer to components in a sense of system equipment)

[12] In [107] McDermid et al present an account of application of HAZOP & SHARD with various sets of guide words (including a larger number of refined deviations classes above as well as extended list of original HAZOP guide words [146]) concluding that the six general classes are more likely to be efficient.

also to the components and software/system architecture itself. The latter is intended to alleviate a known limitation of HAZOP (and SHARD), the focus on deviations of flows (intentional or otherwise) in *intended* plant layout / topology (or, in case of SHARD, intended system architecture).

Other Methods Combining Inductive and Deductive Approaches

HAZOP and its variants do not construct an explicit safety assessment model. Cause-Consequence Diagrams (CCD) – a graphical 'bowtie' analysis technique originally developed in Denmark (by Riso Laboratory) – is, on first examination, closely related to HAZOP. The development of CCDs starts with the specification of a particular "initiating event". The causes of the event are investigated using FTA (and the classical fault tree is constructed). Following the deductive stage the consequences of the event are established, using a procedure similar to Event Tree Analysis but represented using a more compact notation (Figure 10).

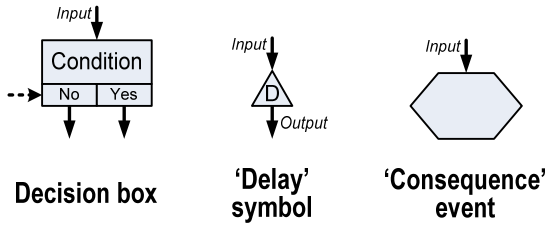


Figure 10 - Specific Symbols of Cause-Consequence Diagrams [160]

The initiating event typically involves "failures of components or subsystems apparently likely to produce dreaded consequences" [160], and consequently the second, inductive, step of the analysis is concerned with the identification of protective barriers and safety systems in the plant under analysis (and their respective malfunctions). Because of this restricted choice of initiating events, CCDs are more closely related to the process that combines fault and event trees[13] as described in [155] (and briefly mentioned in section 2.2.1 above) than they are to HAZOP.

Finally, Lutz and Woodhouse have proposed a *Bi-directional Analysis Method* for the analysis (and certification) of safety-critical software [99]. Their method combines software FMEA and FTA. This method differs from both HAZOP and the CCDs in that complete inductive analysis is conducted first and its outcomes are then used to *prioritise* the construction of fault trees. In this sense bi-directional analysis is a 'two-passes' rather than a 'bowtie' technique.

[13] Although CCDs are more expressive than a simple combination of Fault and Event trees, since a failure mode of every protective system (i.e. not just the initiating event) can be developed into a fault tree "branching off" from the 'no' terminal of the decision box.

2.2.4 Discussion

It is important to note that all of the safety assessment methods that have achieved prominence outside their original industrial sector – FMECA, FTA and HAZOP – share one common feature: their original standards [143, 152, 157] provided significant guidance on the analysis method and included sets of 'guide words' (in one form or another) to facilitate completeness. By contrast, descriptions of techniques that have either faded out over the years or have not escaped the boundaries of a particular industrial domain – such as Reliability Block Diagrams, Event Tree Analysis and Cause-Consequence Diagrams – often focused on the graphical notation, rather than providing a methodology and guidance on how to use the technique. Pumfrey makes this observation and proposes, as his second principle for computer safety analysis, that "*method is more important than notation*" [123]. In studying failure logic modelling approaches, this thesis adopts this principle and focuses on the general *FLM Framework* that underlies diverse failure logic modelling notations and techniques (described in the next section).

It is also important to stress that, regardless of the method, safety assessment models and results will always remain an engineers' *hypothesis* which is typically only weakly validated (if validated at all). Therefore, in practice, evidence that this hypothesis was developed (postulated) as a result of application of a structured, systematic and well-defined method provides an essential (if not the only) basis for justifying the adequacy and the trustworthiness of the safety engineering process as a whole.

Finally, as was mentioned in the previous chapter, most of the 'classical' safety assessment methods share a common 'technical' feature – they explore a causal projection of one undesirable condition at a time. None of the classical methods, therefore, *guarantee* consistency between analyses of multiple conditions (e.g. consistency between different Fault Trees). Furthermore, as modern complex safety-critical systems are typically subjected to a number of different analyses, there is an even lesser 'guarantee' that results across the boundaries of methods will be consistent. Historically, the assurance of consistency between different safety analyses has motivated the research into model-based safety assessment approaches [58] described in the next two sections.

2.3 Failure Logic Modelling Methods

Two of the most prominent early model-based safety assessment methods – Failure Propagation and Transformation Notation (FPTN) and Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) – emerged from the Software Safety Assessment Procedures (SSAP) project at the University of York [57, 58]. The original objectives of this research were, on the one hand, to unify existing 'classical' system safety assessment methods such as FTA,

FMEA/FMECA and HAZOP and, on the other, to adapt them for application to software-intensive safety-critical systems.

As was mentioned in Chapter 1, it was observed that, when applied at the component level, inductive and deductive methods converge. Subsequently, notations were proposed for capturing the results of safety assessment applied to individual components. To capture the behaviour of an entire system, these component-level characterisations are connected in terms of their ability to adversely affect one another's operation. The nature of such interconnections was heavily influenced by HAZOP (or, to be more precise, its SHARD variant) in that component interfaces in HiP-HOPS and FPTN are formed by *deviations of system components' outputs from the intent.* These deviations are often referred to as "failure modes" whereas the *dependencies* in terms of deviations are termed "failure mode flows".

It is important to stress that failure modes are *not* deviations from a component's specification. A component which performs *as specified* may nevertheless exhibit a failure mode as a result of a deviation in some service (provided by some other component) that it relies upon. For example, a hydraulic valve may fail to generate pressure on the output when pressure is intended (an "omission" failure mode) if it either receives no hydraulic pressure from an upstream component (again, an omission failure mode) or is incorrectly commanded to close by the controller or operator (a "commission" failure mode). Of course, deviations from the specification – internal failures or malfunctions of a component – may also cause deviations on the output(s).

The remainder of this section describes existing failure logic modelling approaches.

2.3.1 FPTN

The first failure logic modelling method to emerge, FPTN is a semi-graphical notation. System components (called "modules" in FPTN) are represented as rounded boxes. Modules are either decomposable into other modules (shown within the notation by shading the box) or are already stated at an elementary level. Specification of the module includes a number of standard attributes, such as its name and its criticality. The criticality attribute was mentioned, but never fully described, in the original FPTN publications [58]. It seems that Felenon's intent was to record information similar to the Software Levels defined in DO-178B [124] or Safety Integrity Levels defined in (the now-cancelled) UK Defence Standard 00-55 [148]. He provides no guidance on how this level is derived from – or is otherwise related to – the FPTN model, however.

The most important part of the module specification is a set of failure propagation and transformation equations. Each output failure mode is described by at most one equation that is a

Boolean formula over input failure modes. Failure modes are referred to by both the identifier and failure mode class separated by a colon, for example, *no_hi : Omission* specifies an omission failure mode called "no_hi" (no hydraulic input). The standard set of failure mode classes in FPTN is identical to the set of SHARD guidewords listed in the previous section. However, this default set can be refined, generalised or extended by the user. For example, Fenelon also mentions a general "infrastructure" class [57], although its semantics are not fully explained.

In addition to the propagation and transformation equations, the module specification may include a number of *GENERATED BY* and *HANDLED BY* statements. The former are used to indicate that output failure modes can be caused by (specific) internal failures of the component whilst the later specify any mechanisms that unconditionally prevent input failure modes from propagating (such as a software exception handler for software components or the 'tripping' of an electrical overcurrent circuit breaker).

To illustrate, Figure 11 shows an FPTN characterisation of a simple hydraulic pump. The function of the pump is to provide a constant, defined level of hydraulic pressure on its output ("*o*"). The pump has two inputs: hydraulic ("*hi*") and electrical ("*ei*"). The former is used to supply the pump with (non-pressurised) hydraulic fluid; the latter powers the motor. As far as equations are concerned, omission of either hydraulic or electrical input, clearly results in the omission of the output pressure. However, it is assumed that the pump is not sensitive to any other deviations in its hydraulic input. At the same time, all relevant failure modes of the electrical current (omission, late provision and an incorrect value – too large or too small) generally result in similar failure modes on pump's output, e.g. low current results in a low output pressure.

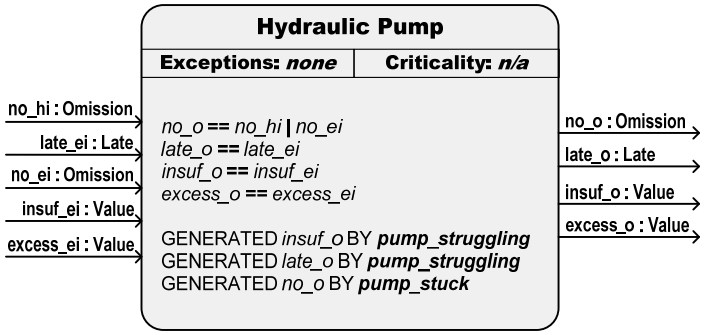


Figure 11 - Example of FPTN Module: Hydraulic Pump

The pump itself can also fail. It can either become fully non-functional (broken) or less responsive to incoming power ("struggling"), e.g. due to increased impedance in the windings. These internal failures would lead to omission of output pressure or low (insufficient) pressure respectively. Finally, a pump which is "struggling" during start-up may provide the required hydraulic pressure, albeit later than expected.

2.3.2 HiP-HOPS

Whilst the papers on the FPTN are widely cited, the research has not been finalised and certain aspects of the notation remain undefined. By contrast, Papadopoulos's Hierarchically Performed Hazard and Operability Studies (HiP-HOPS) [114, 115, 117] is probably the most 'mature' and widely publicised failure logic modelling method to date.

Conceptually, the original HiP-HOPS is very close to FPTN with the major syntactical difference being that, in the former, components are characterised in a tabular format. Figure 12 illustrates HiP-HOPS using the same hydraulic pump example as in the previous section.

Output Failure Mode	Description	Input Deviation Logic	Component Malfunction Logic
O-o	No pressure (i.e. omission) on pump's output Caused by omission of fluid on the hydraulic input ("hi") or omission on the electrical input ("ei") or by the internal failure of the pump ("stuck")	O-hi | O-ei	pump_stuck
V_insufficient-o	Insufficient pressure on pump's output (i.e. value FM) Caused by insufficient voltage on electrical input or by the internal failure of the pump ("struggling")	V_insufficient-ei	pump_struggling
V_excessive-o	Excessive pressure on pump's output (i.e. value FM) Caused by excessive voltage on electrical input	V_excessive-ei	–
L-o	Late reaching required pressure on start-up Caused by late provision of electrical current or by internal failure of a pump ("struggling")	L-ei	pump_struggling

Figure 12 - Example of HiP-HOPS Component: Hydraulic Pump

Also, in HiP-HOPS, failure modes are identified by the name of the output *flow* of the system component and the name of the failure mode class (separated by a hyphen). For example, *V_insufficient-o* identifies an "insufficient value" output failure mode class associated with pump's output (that is, the same failure mode that was previously identified in the FPTN model as *insuf_O : Value*). This difference is not merely syntactical, however: in HiP-HOPS, input and output failure modes are expressly limited to parameters of flows explicitly identified in *design* representation of the system [115]. Whilst this allows HiP-HOPS to reduce the workload of the safety engineers by utilising the structure of design models of the system, it also potentially limits the power of the method since some undesirable interactions between the components (such as propagation of short circuits, hydraulic leaks or synchronous communications delays) may *not* occur in the same direction as flows identified in the system architecture descriptions. It should be noted that in [116] Papadopoulos and Maruhn emphasise this aspect of HiP-HOPS by annotating components

of Matlab Simulink diagrams, thus avoiding the need to remodel the architecture in the HiP-HOPS environment.

The original definition of HiP-HOPS limited propagation equations ("input deviation logic") to logical conjunctions and disjunctions. Recent work by Sharvia and Walker has extended the set of operators to include negation [135] and a set of 'temporal' gates [162, 161] respectively. The latter extension – referred by authors as "Pandora" – essentially refines the FTA's PAND gate.

Also, whilst the components' interfaces in the original HiP-HOPS were strictly *limited to deviations* from intent (failure modes), recent publications on the method suggest that it has been extended to allow some dependencies to be modelled in terms of *normal information flows* (e.g. see discussion on figure 2 in [162] and, in particular, flow *StartA2-M*). This modification of the method is not fully described in any publication, despite the fact that it changes the conceptual nature of HiP-HOPS and moves it closer to the 'hybrid techniques' discussed in section 2.4.3 below.

At the level of the model architecture, HiP-HOPS is based on a hierarchy of interconnected components (Figure 13) which reflects the structure of the design model and is equivalent to the FPTN decomposition of components. In the original description of the method, however, the highest level (root) of the hierarchy represented dependencies between system *functions* constructed through an extended Functional Failure Analysis (FFA) [115]. The functions were mapped onto the next level in the hierarchy (which was the highest level of *structural* decomposition of the system). This provided a strong traceability between the system safety assessment model and the (functional) hazard identification model.

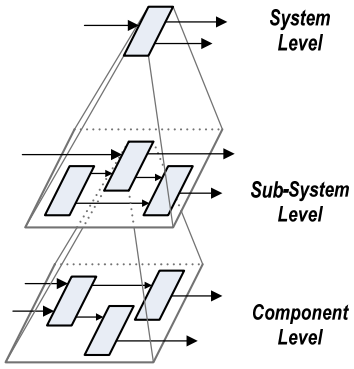


Figure 13 - HiP-HOPS Model Hierarchy (schematic)

Finally, Papadopoulos has implemented an algorithm for the automated synthesis of Fault Trees from HiP-HOPS models [115]. Since component characterisations are logically equivalent to a small 'forest' of very small fault trees (recorded in textual format), the algorithm is essentially a

parser and, thus, yields very fast "analysis" times. A variant of FMEA tables can also be synthesised automatically for the system (or any subsystem) in a similar manner.

2.3.3 Other Methods and Variants

As was mentioned above, one of the key goals of the original failure logic modelling approaches was the unification of FTA and FMEA/FMECA through introduction of the explicit notion of the component (and the component hierarchy). The concept of component-centred or modular FTA is, however, not novel (see, for example, section 3.6.4 of [155] dating back to 1983). More recently, Kaiser has defined a Component Fault Trees (CFTs) [81] notation which was later [80] combined with his dynamic extension to fault trees – State/Event Fault Trees – mentioned in section 2.2.2 above. It is important to stress that, whilst CFTs share some key principles with HiP-HOPS and FPTN, they cannot themselves be regarded as a failure logic modelling approach since the *safety engineering semantics* of the interconnections between components is not defined. By contrast, in this thesis failure logic methods are considered to be characterised by the specification of component dependencies exclusively in terms of deviations of the behaviour under conditions of failure from the intent.

Nevertheless, since the definition of FPTN and HiP-HOPS, other failure logic modelling methods have emerged. As part of his research into issues surrounding Common Cause Failures, Mauri defined *Failure Logic Analysis for Systems Hierarchies* (FLASH) [103, 104] – a tabular method that shares much in common with HiP-HOPS. Wu and Kelly have constructed failure logic models in both Communicating Sequential Processes (CSP) [165] and Object-Oriented Bayesian Belief Networks (OOBBN) [166]. Grunske and Kaiser have combined FPTN with Component Fault Trees [62], whereas Domis and Trapp have integrated both into a general framework of Component-Based Safety Engineering [42, 43]. Similar to this last work, the SPEEDS project[14] has embedded some of the failure logic modelling principles into a general framework of Heterogeneous Rich Components (HRC) [78] and introduced the notion of patterns into failure behaviour specification [38].

One of the recent failure logic methods – Wallace's Failure Propagation and Transformation Calculus (FPTC) [163] – warrants a more detailed description. Closely related to the FPTN, FPTC is based on a fully textual syntax (defined in EBNF). The principal novel contributions of this method are two-fold; firstly, Wallace introduces a notion of variables, wildcards and privatives. These allow for the specification of propagation equations in a very compact and intuitive format. For example, Figure 14 shows three propagation conditions for a component with two inputs and

[14] "Speculative and Exploratory Design in Systems Engineering" - European Commission funded project ongoing at the time of writing [41, 141].

a single output. The first equation uses a wildcard (underscore symbol) and states that if the component is exposed to the commission failure mode on the first input then it will exhibit a commission on the output regardless of any failure modes on the second input. The second equation uses the privative and states that in absence of any deviations of the first input, but under a value deviation of the second input, the component will exhibit an omission failure mode. Finally, the third equation uses a variable (*f*) to specify that when the second input is missing, any failure mode of the first input is propagated to the output (without transformation).

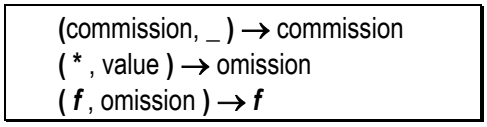


Figure 14 - Example of FPTC Equations

The second and, arguably, the main contribution of the FPTC is in terms of *model analysis*. Unlike most other failure logic modelling methods, FPTC does not suggest the synthesis of fault trees but rather adopts a fixpoint calculation technique (in essence, an inductive strategy). This means that FPTC analysis is apparently possible in presence of loops in the model (which, in contrast, pose significant challenges to most other failure logic modelling approaches). Ge et al [59] further extend FPTC to allow non-deterministic failure behaviour and to analyse and validate the model using a probabilistic model checker.

Finally, over the past decade significant research into application of the AltaRica language to safety analysis of complex systems has been performed by researchers in Office National d'Etudes et Recherches Aérospatiales (ONERA) in Toulouse (France). Whilst some simple AltaRica models used for illustration in some of the publications that emerged from this group (e.g. [21, 132]) can clearly be classed as failure logic models[15], we have seen other models by the same researchers that use (some) non-deviational dependencies / flows between components. Other model-based safety assessment approaches and the AltaRica language are discussed in more detail in the following sections.

2.4 Other Model-Based Safety Assessment Approaches

Model-Based Safety Assessment has been an area of active research since the 1990s. This research and pilot industrial applications have recently attracted the attention of the regulators, the certification authorities and relevant standardisation bodies. For example, the ongoing review of SAE ARP 4754 and APR 4761 documents is likely to result in an explicit reference to model-

[15] Although it is worth noting that these models typically do not include deviations in the *timing* domain (e.g. "early" or "late" failure modes); and the *value* domain is typically represented by a single failure mode – "erroneous"

based safety engineering. Of course, failure logic modelling is not the only approach to have emerged in the past two decades. Indeed, as was mentioned in the previous chapter, one of the key motivations behind the research presented in this thesis is to provide the basis for an informed and rational comparison of those different approaches. This section presents a brief overview of most prominent existing techniques.

2.4.1 Failure Injection Approach

Whilst one of the historical objectives of failure logic modelling was to facilitate *intuitive traceability* between safety assessment and design artefacts, some of the more recent work on model-based safety assessment has sought to *integrate* those artefacts fully. The most prominent body of research in this respect came from two European projects[16]: ESACS [24, 50] and ISAAC [9, 73]. The objective of the work was two-fold: to utilise formal and/or simulatable models of the systems typically generated during the *design* process in the safety assessment and to utilise *formal verification techniques* (namely, model-checking) in model analysis.

Unlike the failure logic modelling approach, Failure Injection (FI) does not require the construction of a separate 'safety model'; instead a "nominal" design model of the system – expressed in a language such as SCADE/LUSTRE [27, 52] or StateMate [65, 69] – is extended by safety engineers, to include behaviour under conditions of failure. The extension is performed essentially by means of a process of injection of simple 'components', which model failure modes, into the data flows of the original model. These components typically have two inputs and one output. One of the inputs is used together with the output to insert the component into the flow; the other – typically Boolean – input is used to 'activate' the component. An inactive failure mode component propagates data from the main input to the output with no change; however, once the failure mode is activated, it disturbs the input according to its predefined logic (i.e. failure mode class) before propagating the result out (in place of the original value vector). For example, an "inverted" failure mode applied to a Boolean flow, when activated, would propagate a negation of the original flow value. Similarly, a "stuck at zero" failure mode would ignore the original flow (i.e. its main input) and propagate zero.

The original model of the system is referred to as the "System Model" (SM) whilst the model after the injection is called the "Extended System Model" (ESM). In addition to the model analysis (described below), tools developed by ESACS project provided a library of predefined

[16] Strictly speaking, both projects have investigated two different approaches to model-based safety assessment that were internally referred to as the Extended System Model (ESM) and the Formal Safety Model (FoSaM). This section focuses on the former, since that was more widely publicised by the projects.

failure mode classes as well as a graphical user interface for the definition of new classes and model extensions. Figure 15 illustrates the model construction, extension and analysis processes.

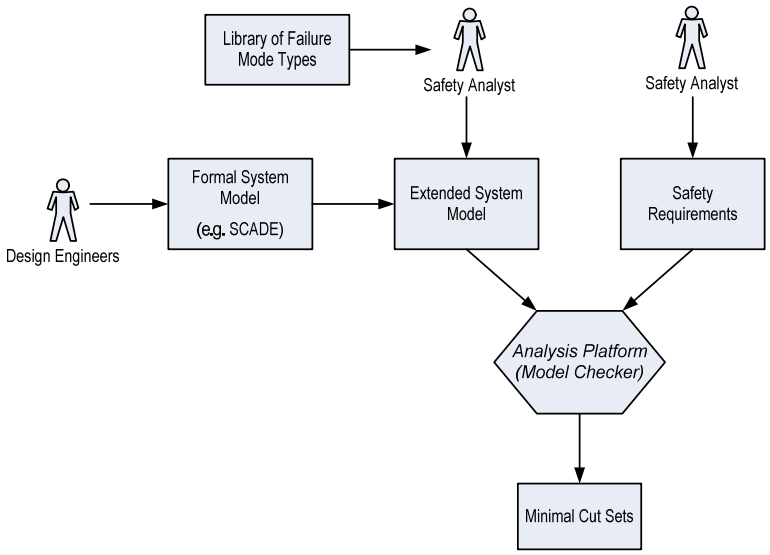


Figure 15 - Overview of the FI/ESM Approach

Once the model is extended, safety engineers need to specify one or more “safety requirements” (proof obligations) – invariants over model variables that describe freedom from a particular unsafe behaviour (such as a system level failure condition). Depending on the system model specification language and the exact tool used, these are specified using either a form of temporal logic [23] or – in case of SCADE models – an observer component [3] (defined in the same language as the model) with a single Boolean output.

Extended system models along with formalised safety requirements contain all of the information necessary to establish whether safety requirements can be violated either by the “nominal” design or by any particular set of failure modes. To analyse models, the ESACS and ISAAC projects have adopted model-checkers – such as NuSMV [23, 31] and Prover Design Verifier [3, 51]. These were augmented since model checkers typically generate a single counter-example in cases when a proof obligation can be violated; however, in the context of safety assessment – where in the presence of failures it is *expected* that a proof obligation will eventually be violated – a single counter example is not sufficient and safety engineers are interested in all sets of failure modes which may cause a potentially hazardous condition (i.e. Minimal Cut Sets).

The baseline failure injection approach and associated analysis tools have been developed in the ESACS project [24, 50]. During the follow-on ISAAC project, interoperability between the tools has been improved and capabilities have been extended beyond the analysis of the (extended) system model[9, 73]. In particular, a methodology for model-based common cause analysis has been established whereby groups of non-independent failure modes in ESM are identified based on three-dimensional geometrical and installation models of the systems (for instance captured in

the CATIA environment) [119]. Other extensions have included mission reliability analysis, failure detectability and diagnosability analysis as well as integration with human error analysis [73, 119]. Whilst noteworthy, these extensions are beyond the scope of this thesis.

The failure injection approach potentially has two key advantages over failure logic modelling:

I. Since ESMs (by construction) contain the complete specification of system design models, their analysis not only identifies failure scenarios that may lead to hazardous effects but also verifies that *system operation in the absence of failures does not lead to unsafe conditions*. In contrast, the failure logic modelling approach abstracts from failure-free behaviour and typically assumes that system design is validated with respect to safety requirements by other means

II. Since 'safety models', ESMs, are obtained by a conceptually trivial transformation of the design model, the consistency between safety analysis results and design models[17] is guaranteed by construction. Demonstration of the consistency between failure logic models and design, on the other hand, is not trivial.

However, this approach also suffers from a number of limitations:

a) Detailed and fully-deterministic system models necessary for the analysis typically emerge *relatively late in the design process,* when the cost of re-design is already high.

b) The completeness of the *analysis is dependent on the completeness of the set of injected failure modes*. To date, no methodological research has been conducted into a systematic method or guidance for identification of the minimally necessary set of failure modes. The problem of completeness of the injection schema is further exacerbated by the fact that each injected failure mode typically models a *concrete* deviation (such as delay for a particular number of time-steps) whilst in failure logic modelling failure modes are typically abstract.

c) The completeness and correctness of the analysis results is potentially *limited by the flows identified in the system model,* as well as by details of characterisation of these flows. Effects of failure modes can only be 'carried' by such flows. At the same time, system models are defined by design engineers for purposes which do not include safety analysis. Consequently, these models are unlikely to identify all possible dependencies between components that may exist under conditions of failure. In more general terms, it can be observed that, by definition, any model is an "*abstraction defined with an intended goal in mind"* [133]; the safety analysis is not part of the intended goal of system models that emerge from the design process. Furthermore, the formal

[17] Also, if design models are used for the automatic generation of implementations, between the safety analysis results and actual implementations.

relationship between the results of automated safety analysis and the design model may give unjustified superficial confidence in the absolute correctness of these results.

As a result it is unlikely that failure injection on its own will be an effective and appropriate basis for system safety assessment. Nevertheless, it can clearly add value to the safety engineering process, possibly as means of partially validating analysis results obtained using other approaches at earlier stages of the design.

Finally, the complexity of the system models of real industrial-scale systems – even before extension – is typically high. Furthermore, these models may contain real-time behaviour and non-linear arithmetic that makes them intractable to the current analysis tools [25, 29].

2.4.2 Failure Effects Modelling Approach

The problems of complexity of actual system models have led to a distinct model-based safety analysis technique which has recently attracted significant attention in industrial experimentation and trials. The approach can be seen as manual definition of the extended system model and is referred to in this thesis as "Failure Effects Modelling" (FEM).

Under this approach, an abstract (typically – simplified) model of the system is constructed. The dependencies between components are captured in terms of characteristics of flows of energy, matter and/or information[18], but at an abstract – typically discrete (and, often, Boolean) – level [20, 100, 134]. For example, electrical dependencies can be captured in terms of the presence or absence of power (or, if more refined level is necessary, the presence or absence of each of the voltage and electrical current 'components' of power). Similarly, hydraulic dependencies are modelled in terms of discrete levels of flow rate and/or the presence or absence of pressure (or refined discrete levels of pressure). By simplifying the model, engineers can, in principle, ensure that its complexity is tractable by existing analysis tools whilst maintaining the relevant aspects of behaviour and/or appropriate level of detail.

The system model is extended with the effects of failures on the input-output response of individual components. However, since the model is constructed specifically for the purpose of safety assessment, this "extended behaviour" can be integrated into the component specification rather than added by the failure injection approach. An example of a failure effects model of an aircraft hydraulic system encoded in the AltaRica language can be found in [20].

---

[18] In this thesis, such flows are referred to as ***nominal*** flows (following the historically-established terminology used in the ESACS, ISAAC and MISSA projects). Nominal flows (or dependencies or interactions) are contrasted with *failure mode* dependencies or flows (see Chapter 3 for definition) which are characterised in terms of a *deviation from intent*.

The approach offers a number of advantages over failure injection:

- The models are obviously less complex and are therefore tractable with currently available analysis tools
- They may be constructed at earlier stages of the design, in parallel with the 'real' system models.

The approach also retains some of the strengths of failure injection:

- Analysis of FEMs is capable of uncovering unsafe (abstract) design specifications *as well as* hazardous effects of failures on system behaviour
- Issues of system reconfiguration are dealt with seamlessly by modelling reconfiguration logic as directly implemented by controller components.

However, some of the key weaknesses of the failure injection approach are retained. Most importantly, only the effects of failures on 'design flows' are captured and new interactions/flows between components that are established by failures fall outside the scope of the approach. Also, since, like extended system models obtained through failure injection, the FEMs capture system behaviour from essentially the same viewpoint as that held by the design engineers, it can be argued that analytical redundancy and conceptual dissimilarity between the safety assessment and the design processes is lost. This is likely to result in less thorough review of the system design, making confidence in safety of the system almost entirely dependent on the automated analysis rather than on the safety engineers' judgement and "creative anticipation".

Finally, the 'guarantee by construction' of consistency between the safety analysis results and the system design model offered by the failure injection approach is significantly weakened, if retained at all, under the FEM approach. The situation is further exacerbated by the fact that, to the author's best knowledge, no systematic method for establishing the appropriate level of granularity of FEMs (such as the appropriate number of discrete value levels or appropriate selection of model's 'time-step') either exists or is being developed.

2.4.3 Hybrid Approaches

The preceding two sections have outlined two distinct approaches to model-based safety assessment: Failure Injection and Failure Effects Modelling. The approaches are different in that, whilst FI relies on a pre-existing model of the system (which is merely extended for the purpose of safety assessment), FEMs are constructed expressly for the purpose of safety assessment. Both approaches, however, share a key feature: they rely on the models that capture *nominal* dependencies between components (such as flows of energy, matter or information). In Vesely's

terminology [158, 157] both can be considered as “success models” (as opposite to “failure models” such as fault trees and failure logic models described in sections 2.2.2 and 2.3 above). However, examples of combinations of these approaches with the principles of failure logic modelling can be found in publications and are briefly discussed in this section.

The FEM approach is frequently combined with failure logic principles to compensate for its inability to capture dependencies between components established by failures. Closer to failure logic modelling in the spectrum of ‘hybrid methods’ is the approach illustrated by Bernard et al in [19]. In their model of the A340 Rudder Control System (expressed in AltaRica), most of the dependencies between components are characterised in terms of two failure modes: “erroneous” (a generic value domain failure mode) and “lost” (essentially an omission failure mode in FPTN or HiP-HOPS terminology) – along with a ‘correct’ privative. However, for modelling a ‘switch-over’ (reconfiguration) between redundant controllers of the system, a nominal Boolean ‘activation’ flow is used. In other words, this flow does not model a deviation from intent, but rather captures the ‘real’ control signal from one controller to the other (as can be found in a design model of the system). The author has learned from discussions with ONERA researchers that the general approach being followed in this example is to model all of the flows which participate in the system’s monitoring and the (consequent) control of reconfiguration actions in the “success domain” whilst modelling all remaining flows in terms of deviations from intent.

Another example of combining FEM and failure logic modelling approaches can be found in [82, 132]. Here, dependencies between the components in the direction from power sources (pumps and electrical generators) towards consumers are modelled in terms of discretised nominal flows. The dependencies in the opposite direction (propagation of short circuits and effects of leaks), which would normally lie outside the scope of the FEM approach, are modelled as well (by definition – in terms of the propagation of failure modes). However, this distinction between two different engineering semantics of flows is not apparent (and requires detailed consideration of example systems and extrapolation of models), and is further concealed by both ‘kinds’ of flows being defined over the same enumerated type.

A conceptually similar but more explicit approach is taken by Joshi and Heimdahl [76, 77] for integrating the principles of Failure Injection and failure logic modelling (the later described by the authors as “error propagation”). Firstly, unlike the failure injection technique developed in the ESACS and ISAAC projects, which is strictly limited to injection *into* flows, Joshi and Heimdahl’s approach injects failure modes into system models by ‘*wrapping*’ components in

additional behaviour [77]. This approach may in some cases[19] expand the expressive power of the model extension, since it allows for the dependency of the deviation being injected into a component's output on the *input of that component* to be modelled. More importantly, the technique enables the modelling of dependencies between the components in presence of failure even if the system model does not identify a dataflow connecting the two components. This is done by overlaying a failure logic model onto the extended system model. However, unlike Papadopoulos's approach, which overlays HiP-HOPS annotation over Matlab Simulink models [116] without integrating their behaviours, Joshi and Heimdahl propose full integration of the two models. Thus the effects of failure are propagated through the nominal flows whenever these are available, or through the "error propagation layer" whenever failures establish new dependency paths.

The authors also propose a special-purpose extension to the Lustre language for capturing the overall layered model [76].

Reese and Leveson proposed a different approach to combining characteristics of failure injection and failure logic modelling – Software Deviation Analysis (SDA) [127, 128]. The objective of this approach is to analyse the robustness of software in terms of its response to possible deviations in the environment that the software is exposed to over its inputs. The software itself and the underlying execution platform are assumed to be fault-free. The approach is similar to failure injection, in that it utilises the model of the software (expressed in RSML [92]) rather than relying on the construction of a separate model. Inputs of the software are associated with sets of possible deviations – all in the value domain.

The approach, however, relies on symbolic execution over a set of (abstract) failure modes propagated through the model as tokens. In this respect, SDA is close to the failure logic modelling approach. To enable such analysis, Reese partitioned numeric (real) value deviations into a qualitative / discrete domain by using a logarithmic scale. The analysis process takes an RSML specification and converts it into a simplified internal causality diagram representation. Reese has developed a catalogue of deviation transformation equations (deviation calculus) associated with each causality diagram operator. This allows for the automated traversal of the diagram to establish (in qualitative terms) the effects of input deviations upon outputs of the software being analysed. Detailed description of the SDA is beyond the scope of this thesis and the reader is referred to [127] for the definition of analysis algorithms as well as the deviation calculus that underlies the SDA method. However, it is important to note that, whilst SDA uses

[19] That is, if the nominal input-output function of the component (for any output) is *not* bijective. Otherwise, any Joshi and Heimdahl failure mode "wrapper" can be formally reduced to an ESACS/ISAAC failure mode "component".

some features of failure logic modelling, this is done specifically to reduce the computational complexity of the analysis and is largely hidden by the analysis tools.

Finally, Heimdahl et al note some limitations of the SDA and propose a modification of the method that uses the NuSMV model-checker [66]. The resulting approach is conceptually similar to the 'pure' failure injection, with the two key differences:

- Failure modes are only injected into inputs of the software model;
- The target of analysis is a discrepancy between the co-analysed nominal and extended models. Whilst conceptually the two models are composed in parallel, in practice, the authors propose a more effective, but mathematically equivalent, model representation that minimises the problem of state explosion. Conceptually, the composed models are analysed for the identical input vector, discrepancies (deviations) are observed across a pair of respective variables in two models selected by the safety engineer.

2.4.4 Model-Based Safety Assessment: Summary

The preceding sections 2.3 through 2.4.3 have described three key approaches to model-based safety assessment – failure logic modelling, failure injection (also known as extended system model approach) and failure effects modelling – along with a brief overview of some of the combined approaches found in the literature. Two criteria can be used to distinguish between these approaches:

a) *The "provenance" of the model:* whether the model being analysed is constructed specifically for the purpose of the safety assessment or whether an existing model of the system used in the design process is being utilised
b) *The engineering semantics of dependencies:* whether dependencies between components are characterised in terms of nominal – "success world" – flows or in terms of deviations from intent (failure modes).

The resultant classification of the model-based safety assessment approaches and techniques is illustrated in Table 2.

In terms of the "classical" safety assessment methods, failure logic models relate closely to fault trees and HAZOP in that they model system behaviour in the "failure space" and are based on the concept of behaviour deviation from intent. Because the perspective of these models is fundamentally dissimilar from that of system design engineers, model construction (which itself is a system assessment activity) provides an opportunity for thorough review of the proposed system architectures and yields a high a degree of analytic redundancy between safety and development processes. This approach can also be applied at early stages of design when design descriptions are incomplete and immature. However, whilst failure logic modelling enables strong traceability

between safety and design model architectures, the consistency between behaviours of two models cannot be formally verified using current techniques. Furthermore, as will be demonstrated in Chapter 5 of this thesis, dynamic reconfiguration of the system poses some significant challenges to the 'purist' application of failure logic modelling.

Table 2 - Classification of the Model-Based Safety Assessment Methods

		Engineering semantics of components dependencies		
		Only nominal dependencies	**Both nominal and deviation dependencies**	**Only deviation (failure mode) dependencies**
Model 'provenance'	**Model is constructed manually specifically for safety assessment**	***Failure Effects Modelling*** e.g. Shaikh [134], Majdara and Wakabayashi [100] techniques, some ONERA models [20]	***Hybrid Approaches*** e.g. some ONERA models [19, 82, 132]	***Failure Logic Modelling*** e.g. FPTN [58], FPTC [163] and HiP-HOPS [114]
	Safety assessment model is partially automatically constructed (e.g. model architecture)	***Hybrid Approaches*** e.g. ESACS/ISSAC ESM approach if underlying system models have to be simplified [24, 25, 29]	***Hybrid Approaches*** e.g. Joshi and Heimdahl [76]	***Hybrid Approaches*** e.g. HiP-HOPS integration with Simulink [116]
	Design model is utilised or safety assessment model is automatically constructed	***Failure Injection*** e.g. ESACS/ISAAC FI/ESM methodology [3, 23, 119]; Heimdahl SDA Approach [66]	***Hybrid Approaches***	***Hybrid Approaches*** e.g. Software Deviation Analysis [128]

By contrast, Failure Effects Modelling is, to some extent, related to reliability block diagrams as well as to some of the historical approaches to automated fault tree synthesis (such as digraphs and flowgraphs). The key advantage of the FEM and failure injection approaches is that the reconfiguration of the system poses no significant challenges: there is typically no need to explicitly recognise the notion of modes, given that reconfiguration rules can be modelled as part of the controller components' specification. In addition, the failure injection approach guarantees by construction that results of the safety assessment are consistent with the system design model[20]. However, a strong link with the design model (or, in the case of FEMs, with the design engineers' viewpoint) may also be considered a weakness of both approaches, as it may constrain safety assessment to the mere consideration of effects of failures which can be propagated through dependencies explicitly identified in system models. Furthermore, these approaches do not

[20] For failure effects modelling, it may in future be possible to verify consistency formally, by proofs of refinement (or its liberalised forms [16, 98]), at least for the digital part of the system. A notable theoretical work in that respect is being carried out by Banach and Bozzano [17].

provide an opportunity for a thorough review of the design proposals – one of the key (although intangible and often understated) objectives of the safety assessment.

Finally, numerous hybrid approaches have recently emerged which attempt to compensate for the limitations of 'nominal' methods. Whilst these have been demonstrated to be technically feasible, a repeatable and systematic method for *construction* of such hybrid models is yet to emerge. In particular, it remains to be shown that different paradigms (such as FEM and failure logic modelling) can be combined *systematically,* avoiding any internal inconsistencies in the model.

It should be noted that that the lack of a systematic construction method is also a concern for 'pure' FEM approaches. In fact, their conceptual proximity to the design models may in practice conceal some of the key challenges to the validity of any analysis – namely the granularity and the level of abstraction of the model. Indeed, whilst FEMs typically abstract and discretise numeric flows in system models, there are currently no systematic methods that would help safety engineers to select the right abstraction level (i.e. the abstraction which reduces complexity of the model whilst not discarding any information that is relevant to the safety analysis).

2.5 Modelling Languages

As was mentioned in the introduction chapter and in section 2.3 above, most of the failure logic methods are intrinsically linked to idiosyncratic specification notations. Use of such notations is invaluable in the formative stages of new research disciplines; however, it is undesirable once a certain level of maturity is reached and is rarely acceptable in an industrial context. On the other hand, many FEM and hybrid methods have been implemented in relatively general languages; however, modelling *methodologies* (i.e. a clear definition of engineering concepts being modelled and systematic model construction methods) for these approaches are lacking. Instead, FEM and hybrid (and some of the failure logic modelling) approaches appear to be driven by the features and constructs of particular specification languages.

Overall, whilst this thesis follows Pumfrey's principle that "method is more important than notation" [123] it does not contend that the choice of notation is irrelevant. However, the objective of the research presented in this thesis is not to define yet another modelling notation or language, but rather to clarify and extend the conceptual framework of failure logic modelling whilst demonstrating that a concrete instantiation of the framework is possible. The latter is achieved by implementation in a relatively general specification language.

As far as the choice of language is concerned, the alternatives range from safety-specific notations (such as Kaiser's component state/event fault trees [81]) to general specification languages widely

used in model-based systems and/or software engineering (such as Simulink, SCADE or even UML/SysML). Both 'extremes' of this scale suffer from significant limitations: the former languages are idiosyncratic and thus do not convincingly demonstrate that separation between a methodology and an implementation language is indeed possible; the latter – industrially mature – languages are typically not (yet) associated with the tools necessary for safety analysis. As a result, this thesis makes a compromise of using languages that have been specifically adopted (by others) in the context of safety assessment, on the one hand, whilst being both 'convertable' into more widely used languages and used by a number of *different* researchers on the other. This section presents a brief overview of two such languages – AltaRica and AADL's Error Model Annex.

2.5.1 AltaRica and Associated Dialects

Developed by researchers at Laboratoire Bordelais de Recherche en Informatique (LaBRI) of the University of Bordeaux in collaboration with a number of French industrial partners, AltaRica is a constraint automata language [13, 122] that has been extensively applied in the context of safety and reliability analyses.

The language recognises the notion of components – called 'nodes'. Each node is essentially an interfaced automaton defined over a number of state variables, flow variables and events. Both state and flow variables are defined over (potentially infinite) discrete domains. State variables model the automata where events trigger transitions between the states; the values of the states are hidden within the boundaries of the node. Flow variables provide an interface of the node (and are, thus, visible both internally and externally).

The behaviour of the node is specified through *transitions* and *assertions*. Assertions specify constraints (invariants) over the values of flow and state variables. Transitions on the other hand, determine the state of the node and consist of a single trigger (as mentioned before – an event) and a guard that constraints the transition; the latter is essentially an assertion over any flow and state variables. Following a change in state variables, flow variables are updated instantaneously (through implicit ε-transitions). To illustrate, Figure 16 shows a 'classical' AltaRica example of an electrical switch (contactor) [61, 122]. The state space of the switch is finite and is modelled by a single Boolean state variable *IsClosed*; the transitions between the two states (values of the variable) are triggered by events *open* and *close* and are specified in the *trans* clause. The switch has two flow variables (interfaces corresponding to two terminals): *f1* and *f2*. The assertion states that when the switch is closed the power on both terminals is identical. Note that this is a non-deterministic specification.

Nodes can be hierarchically organised to reflect system decomposition and architecture. A higher-level ('complex' or 'composed') node specifies instances of (other) node models and communications between them. Nodes can communicate in two ways: through interfaces or through event dependencies. The former is specified as assertions over interfaces and the latter as synchronisations. To illustrate Figure 17 shows a specification of a composite node *SwitchPair,* which contains two switches (as previously defined) connected in parallel and operated simultaneously. Note that the *SwitchPair* has its own flow variables – *p1* and *p2.*

```
node Swith
  flow
    f1, f2 : bool;
  event
    open, close;
  state
    IsClosed : bool;
  trans
    not IsClosed |- close -> IsClosed := true;
    IsClosed |- open -> IsClosed := false;
  assert
    IsClosed => (f1 = f2)
edon
```

Figure 16 - Node Example in AltaRica: Switch

```
node SwitchPair
  sub
    S1, S2 : Switch;
  flow
    p1, p2 : bool;
  assert
    p1 = S1.f1;
    S1.f2 = S2.f1;
    p2 = S2.f2;
  sync
    <S1.close, S2.close>
    <S1.open, S2.open>
edon
```

Figure 17 - Node Composition in AltaRica: Switch Pair

The complete AltaRica language (referred to as AltaRica LaBRI) is supported by a number of analysis tools. However, some language features (such as the non-determinism of the models) introduce significant complexity into analysis. To overcome these issues and to provide a link to dataflow languages (such as LUSTRE/SCADE), which are extensively used in industry, Rauzy et al have developed a restricted dialect of AltaRica – AltaRica Dataflow [22, 125, 126]. The key features of this dialect are that:

- All flow variables of the nodes must be characterised as either input or output;
- Non-determinism between assertions is prohibited. In other words, for every configuration of state variables and input flow variables, the value of each output flow variable must be defined;
- Non-determinism between transitions is also prohibited.

Figure 18 shows an AltaRica Dataflow characterisation of the switch component above; this characterisation assumes that the *f1* terminal of the switch is always connected to the power source and that *f2* is connected to the 'consumer' network.

```
node UnidirectionalSwith
  flow
    f1 : bool : in;
    f2 : bool : out;
  event
    open, close;
  state
    IsClosed : bool;
  init
    IsClosed := false;
  trans
    not IsClosed |- close -> IsClosed := true;
    IsClosed |- open -> IsClosed := false;
  assert
    f2 = ( case { IsClosed : f2,
                  else : false} );
edon
```

Figure 18 - Node Example in AltaRica Dataflow: Switch

AltaRica Dataflow was originally supported by the Combava suite of analysis tools (now obsolete). A restriction of AltaRica LaBRI that is nearly identical to the Dataflow dialect can be translated into Lustre [61].

Most importantly, from the perspective of this thesis, Dassault Aviation has developed a tool – Cecilia OCAS [131] – which implements a nearly identical AltaRica dialect. Cecilia OCAS provides a graphical environment for the construction and simulation of models. It also provides model analysis functionality including the generation of fault trees (based on Rauzy's approach[21] [126]) and sequence generation. The latter is essentially an inductive search tool that generates minimal sequences of events, up to a pre-specified maximum length, that lead to a particular condition defined by the user. Also, work is currently under way under the auspices of the MISSA project [6] to integrate the NuSMV model checker and its associated NuSMV-SA analysis platform [23] into the Cecilia OCAS environment. Currently, the Cecilia OCAS platform can be characterised as a mature industrial prototype. In addition to numerous research and evaluation projects mentioned in the previous sections, the platform has been used for certification (at the PSSA level) of the Flight Control System of the Dassault Falcon 7x aircraft [9, 131].

[21] Note that, whilst they are syntactically valid, the "fault trees" generated by Rauzy are not necessarily valid with respect to the *engineering semantics* of fault trees defined in [158, 157]. In particular, intermediate events represent values of flow variable which, depending on the modelling approach, are not necessarily "faults".

Returning to syntax, AltaRica (including its Dataflow and OCAS dialects) provides a general *extern* clause for tool-specific extensions of the language. These extensions typically take the form of annotations associated with particular constructs of the core language – such as events, states or nodes. A key example of such an extension is the association of events with parameterised probability laws. These law associations are exported along with the analysis results (e.g. minimal cut sets/sequences), thereby allowing quantitative analysis to be carried out using standard reliability tools.

Overall, AltaRica is a powerful and yet syntactically compact formally defined language. Its use for model-based safety assessment has been demonstrated in both research and industrial contexts. However, the language does not inherently restrict the modelling methodology and engineering semantics of models. Indeed, previous sections have demonstrated that different AltaRica models found in literature follow different modelling approaches.

2.5.2 The Architecture Analysis & Design Language

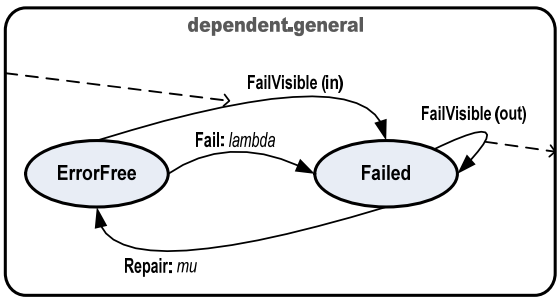
The Architecture Analysis and Design Language (AADL) is a domain-specific language for the specification and analysis of real-time embedded software standardised by SAE [138]. It has evolved from an earlier proprietary architecture description language – MetaH – developed by Honeywell [159]. The underlying semantics of the language is based on the notion of hybrid automata [93]; however, syntactical constructs are tailored to the embedded software domain.

One of the key objectives of the AADL is provision of the unifying framework for "co-design" of architectures of application software and underlying hardware platform. The language provides a standard set of component stereotypes, such as 'process', 'thread', 'data' and 'subprogram' for software as well as 'processor', 'memory', 'bus' and 'device'[22] for hardware [138]. Relationships (e.g. allocations) between software and hardware components are specified through bindings. The language supports various types of interaction between components, such as events and dataflows. Overall, the syntax of the AADL is very rich, reflecting various *specific* patterns of structure and behaviour typically found in the designs of *embedded software*. Furthermore, the language is extensible through customised "annexes". The SAE standard itself defines two such annexes for Behavioural and Error Modelling. The latter is described briefly in the remainder of this section.

The Error Model Annex of the AADL [56, 136] defines additional constructs for specifying component behaviour in terms of the propagation of "faults". In its simplest form, it can be seen as a failure logic modelling language in its own right. A component's failure behaviour is specified as a single stochastic automaton consisting of a finite number of error states. Transitions

[22] Devices are used to model equipment outside the control system such as sensors and actuators.

are triggered either by internal “error events” or by “error propagations”. Error events represent malfunctions and repairs or, in the words of AS5506, are “internal intrinsic events that change the error state of a component” [136]. Error propagations are *effects* of error states of other components on the component in question; in other words, components of an AADL error model communicate in terms of error propagations. In terms of automata semantics, both error events and propagations are events with the only difference being their ultimate source. Figure 19 illustrates this description with a simple component error model which has two states: the initial ‘error free’ state and a *Failed* state. The transition from the initial state to a failed state can be triggered by either an internal event (*Fail*) or an incoming error propagation from component’s environment (*FailedVisible*). The reverse transition can be triggered by an internal error event that models a *repair* action. Once it has failed, a component will generate an error event (i.e. an outgoing error propagation) to its environment (*FailVisible*) with a (fixed) probability *p*. Similarly, internal events are associated with a *Poisson* probability distribution (with parameters *lambda* and *mu* for *Fail* and *Repair* respectively). Note that the definition of the component’s ‘failure logic’ is achieved in two steps: first an error model *dependent* specifies the structure of the component; then a model implementation (*dependent.general*) specifies the actual behaviour of the component.




```
error model dependent
  features
    ErrorFree: initial error state;
    Failed: error state;
    Fail: error event {Occurrence => poisson lambda};
    Repair: error event {Occurrence => poisson mu};
    FailedVisible: in out error propagation
                          {Occurrence => fixed p};
end dependent;

error model implementation dependent.general
  transitions
    ErrorFree-[Fail]->Failed;
    Failed-[Repair]->ErrorFree;
    ErrorFree-[in FailedVisible]->Failed;
    Failed-[out FailedVisible]->Failed;
end dependent.general;
```

Figure 19 - Example of a Component in the AADL Error Model Annex [56]

In addition to propagating (and, in the general case, transforming) errors through a change in the error state, the AADL Error Modelling Annex allows components to channel incoming errors out directly through a *Guard_Out* construct. Such a propagation can be either unconditional or conditioned on the error state of the component and is conceptually close to the *assertion* declaration in AltaRica Dataflow. The same construct is used for filtering or masking outgoing error propagations.

At the level of the system architecture, error propagations are not modelled explicitly but instead utilise connectors and bindings in the 'core' AADL model. Component error models are attached to the core specifications of components (using a predefined "*annex error_model {...}*" construct) and outgoing error propagations are transmitted to all connected components according to rules defined in section 3.5.2 of the Annex E [136]). For example, propagations may occur from a processor to all bound threads, from a component to any outgoing connections and between all sub-components of the same process. Conceptually this is close to Papadopoulos's approach of integrating HiP-HOPS and Matlab Simulink models [116] (although the underlying core AADL model provides significantly richer types of connections and bindings than do typical Simulink models).

As well as reacting to explicitly defined propagations of connected components, error models may access the internal (i.e. otherwise hidden) error state of the connected components through a *Guard_In* property. The same construct is used to adjust error propagation names or to merge incoming errors (e.g. if a sending component sends too detailed error propagations from the perspective of the receiving component).

AADL Error Models are not simply 'layered on top of' core AADL models. The behaviour of two 'layers' can be integrated. For example, the *Guard_Event* property, declared in the error model specification, can be used to generate a 'real' system event (that is, an event on a port declared in the core AADL model); similarly, the *Guard_Transition* property can override the mode transitions declared in the core model. This integration means that the AADL Error Modelling Annex is not just a failure logic modelling language, but rather a more general language which can also be used in the context of hybrid approaches.

Finally, it is important to note that a number of researchers have recently integrated AADL and AltaRica Dataflow. Bieber et al [21] briefly reports two such integration approaches investigated by the ASSERT project [14]. Under the first approach, AltaRica Dataflow models are combined with the core AADL models (the former thus essentially replace the Error Model Annex). To achieve this, AADL models are annotated with references to the AltaRica nodes stored in a library. The tool implemented by ONERA synthesises the overall AltaRica model from the *structure* of the AADL model and the library of AltaRica components.

In contrast, the second approach does not replace the Error Modelling Annex, but rather translates its models into AltaRica. Bieber et al, however, report that the resultant models are significantly more complex than the manually constructed AltaRica models. Another approach to translation between AltaRica and AADL Error Modelling Annex is currently being investigated by Mokos et al at the Aristotle University of Thessaloniki (Greece) [108]. Their approach is based on an *ontology* which unifies the AADL Error Modelling Annex and the AltaRica Dataflow semantics.

There is some similarity between this work and the general metamodelling-based approach adopted in this thesis. However, the ontology of Mokos et al is specifically developed for the two specification *languages* and may, in principle, be applied under different modelling approaches, whereas the FLM Framework presented in this thesis presents an entirely language-independent failure logic modelling domain framework with an instantiation in AltaRica used solely to demonstrate that the framework is consistent, sound and 'implementable' in a third-party specification language.

2.5.3 Language Selection

This thesis uses the OCAS and Dataflow dialects of the AltaRica language[23] to demonstrate the concrete instantiation of the developed Failure Logic Modelling Framework. As the specification language plays only a secondary role in the research presented in the thesis, it was felt that AADL and its Error Model Annex are 'over-configured' to the particular class of embedded systems. Also, whilst the Error Model Annex is a powerful specification formalism which can probably be applied in the context of different modelling approaches, it is constrained by the strong tie with the 'core' AADL.

By contrast, AltaRica Dataflow and AltaRica OCAS are general specification languages with a powerful, yet compact and intuitive, syntax. The language itself is not constrained by any system design or safety assessment features. Also, translation from this language to Lustre (and, by extension, to SCADE) has been demonstrated to be effective. At the same time, the language is supported by relatively mature sets of tools for model construction, management, simulation and sequence generation (as well as for model-checking and fault tree synthesis which are not used in this thesis). Finally, selection of the language has been influenced by the present author's collaboration with the Dassault Aviation (developers of the Cecilia OCAS suite) and ONERA (prominent users of AltaRica).

2.6 Conclusions

This chapter has presented a review of literature in system safety engineering, and safety assessment in general as well as in model-based safety assessment in particular. Although, the latter topic has attracted significant interest in academia and industry over the past two decades, the sub-discipline remains largely unorganised. This chapter has proposed a classification of the model-based safety assessment approaches based on the *provenance of the models* being used and

[23] The OCAS dialect is used for model construction. However, in this thesis the Dataflow dialect is used for illustrations. This is entirely due to the specifics of the export formats and functionality of the Cecilia OCAS tool – the AltaRica OCAS export format requires significant manual re-formatting for integration with the thesis whereas Dataflow export functionality (of the same tool) provides "clean" ASCII format.

the *engineering semantics* of the captured relationships between components. This classification yields three ‘purist’ approaches: Failure Logic Modelling, Failure Injection and Failure Effects Modelling along with a number of ‘hybrid’ techniques.

The provenance of the models is important, since any model is an “*abstraction defined with an intended goal in mind”* [133]. The authorship of the models therefore clearly affects whether they are likely to be appropriate for safety assessment. The engineering semantics is important because it establishes (whether explicitly or implicitly) a particular ‘view of the world’ or a ‘domain’ that encapsulates key concepts and their relationships and which circumscribes the aspects of safety behaviour of the system which can be described.

A key problem in all publications on model-based safety assessment is a focus on a particular technique or, worse, a specification language without a definition and examination of the domain of concepts being represented. Furthermore, the techniques are typically intrinsically linked to *particular* specification languages, or worse, driven by the features of such languages. The constraints imposed by a particular technique or a particular language on the domain are often not examined.

The following chapter takes a broader view, and defines a *Failure Logic Metamodel* which underpins a prominent family of model-based safety assessment techniques in a language- and technique- independent fashion. Examination of this metamodel permits assessment of the fundamental strengths and weaknesses of all of the individual methods that instantiate it.

Chapter 3: Unifying Failure Logic Metamodel

3.1 Introduction

The individual failure logic modelling techniques examined so far have introduced new notations for capturing the failure logic (sometimes referred to as failure propagation logic) of the system and its individual components. At the same time, some significant work has been done on the adaptation of standard specification languages in the context of model-based safety assessment (e.g. [19, 20, 24, 29, 77, 82, 126, 132]). As discussed in the previous chapter, such work has often focused on language-specific features, and the construction of 'safety models' typically has not followed any explicit and structured methodology. The purpose of this chapter is to unify the research work on individual failure logic modelling methods and the use of standard specification languages in a clear and repeatable fashion through the definition of a *Failure Logic Metamodel*. Having clearly defined the 'baseline' approach in this chapter, the metamodel is further extended in Chapters 4 and 5.

The key contributions and structure of this chapter are as follows:

- ***Section 2*** presents a basic *Failure Logic Metamodel (FLMM) which unifies existing failure logic modelling methods* such as HiP-HOPS [114], FPTN [57] and FPTC [163]. The purpose of this basic metamodel is not to improve on existing methods and, at this stage, extensions are kept to the absolute minimum necessary to rationalise the relationships between the various fundamental concepts in a coherent manner.
- ***Section 3*** *extends the expressive power of the FLMM* in two ways. Firstly, the *dynamic behaviour* of components is enhanced. Secondly, extensions are introduced to allow the impact of key aspects of *normal (i.e. non failure-related) behaviour* of the system on its failure logic to be taken into account.
- ***Section 4***, which has been included mainly for the sake of completeness, discusses different approaches to the analysis of failure logic models and identifies some fundamental problems of the automated synthesis of fault trees and FMEA/FMECA tables.
- ***Section 5*** demonstrates that the general Metamodel can be *instantiated in a standard third-party specification language*.
- The chapter concludes with an overview of the *case study* (introduced briefly in Section 3.1.1 below) and outline of some of the limitations of the 'baseline methodology' that are addressed in the remainder of the Thesis.

3.1.1 Introduction to the Illustrative Example & Case Study

Throughout this chapter, an illustrative example of a simplified hypothetical aircraft Wheel Braking System is used. The original example is taken from Appendix L of Aerospace Recommended Practice ARP 4761 [139], and is further simplified here[24].

The main function of the system is to provide safe retardation of the aircraft in its taxiing and landing phases as well as in the Rejected Take-Off scenario by means of the application of hydraulic pressure to the brake assemblies of the main landing gear wheels. However, the design of the brake assemblies is considered outside the scope of the system and the case study therefore focuses on the provision of braking pressure to the actuators.

The architecture of the Braking System is shown in Figure 20. Overall, braking pressure is provided to the landing gear assemblies via two redundant hydraulic channels – Green and Blue. Normally (in absence of failures) both channels are operated simultaneously. However, a single operational channel is fully capable of effecting safe wheel braking.

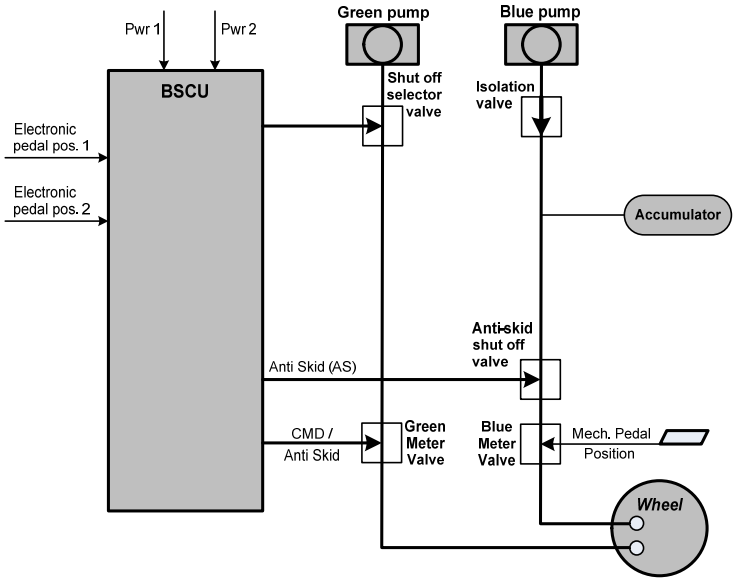


Figure 20 - Architecture of the Hypothetical Aircraft Wheel Braking System

The Green channel is controlled by a computerised Braking System Control Unit (BSCU). The primary output of the BSCU is a single control signal to the Green Meter Valve, which combines braking demands (communicated to the BSCU via two redundant electronic "Pedal Position" signals from the cockpit) and anti-skid constraints. The BSCU also provides the blue hydraulic channel with 'clean' anti-skid controls and produces a "Validity" output which indicates whether the main unit outputs are trustworthy. The outputs are deemed to be untrustworthy if an internal failure is diagnosed or if a discrepancy in the pedal inputs has been detected. The validity output

[24] The original – *not* simplified – WBS design will be addressed in Chapter 5.

of the BSCU is fed into a dedicated Shut-Off Selector Valve (located upstream from the Meter Valve of the Green Channel). In order to prevent an untrustworthy BSCU from interfering with normal braking, the valve prevents hydraulic flow if the control unit is declared invalid.

The BSCU is an internally redundant sub-system (Figure 21) consisting of two identical “sides” or “channels” (BSCU1 and BSCU2). Each side consists of a Command module and a Monitor module; it is assumed that all four modules are implemented in software and that the hardware platform is outside the scope of the WBS. In addition to the two sides, the BSCU contains *Validity Monitor* and *Switch* components which consolidate the Command (CMD), Anti-skid (AS) and Validity outputs of BSCU1 and BSCU2.

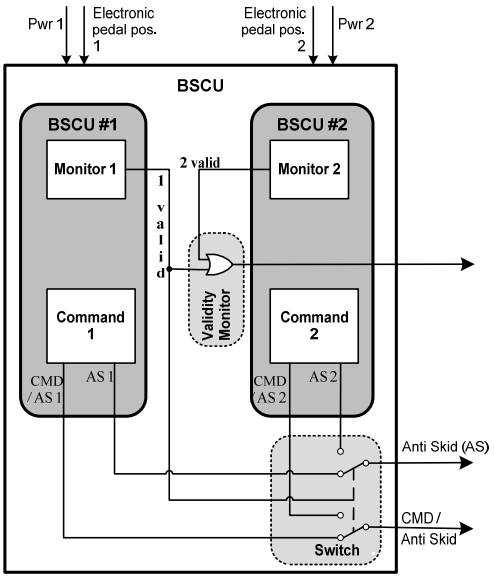


Figure 21 - BSCU Architecture

Returning to the hydro-mechanical part of the system, the blue channel is powered by a separate pump as well as a hydraulic accumulator, with the accumulator providing pressure only if no pressure is otherwise present on the line (e.g. due to pump failure). As the hydraulic accumulator has a finite capacity, it is assumed that it can provide sufficient pressure for a safe *single* execution of the braking manoeuvre (within the operational envelope of the aircraft). The accumulator is assumed to be fully charged in a pre-flight / engine start-up sequence and it is assumed that, in the absence of failures, the accumulator maintains full pressure for the duration of the flight.

Regardless of whether the blue channel is powered by the pump or by the accumulator, the pressure to the wheel is controlled by two valves: a meter valve connected mechanically to the braking pedals in the cockpit and an Anti-Skid Shut-Off Valve driven by the dedicated output from the BSCU.

3.1.2 System Intent & Design

Before the key concepts and the domain model of Failure Logic Modelling are presented, it is necessary to discuss the concept of "Design Intent" (or "System Intent").

Design Intent is never explicitly represented in failure logic models and is often overlooked by the descriptions of particular techniques. However, it is arguably the single most important concept behind the approach which underpins the theory of this kind of modelling. Indeed, the key hypothesis of any failure logic modelling approach can be stated in the form of three (related) propositions:

[For the practically significant proportion of systems and hazards:]

(1) Immediate causes of hazards posed by a system can be described in terms of the deviation of system behaviour from the overall intent;

(2) The inability of the system as a whole to fulfil its intent can be attributed to – and described in terms of – the deviation of externally-observable behaviour of one or more individual components from their respective intents (in the context of the system design);

(3) Every such deviation of component behaviour can be attributed to – and described in terms of – a combination of internal malfunction(s) of the component and/or deviation(s) in the behaviour of some other component(s) that the component concerned is susceptible to.

The above propositions highlight key assumptions about design intent that are made by the failure logic modelling methods. However, it is important to qualify these assumptions to avoid misinterpretation. First, it is clearly assumed that the *intent of the system is inherently safe*.

Secondly, *design of the system decomposes the overall intent into the intents of individual components*. A component's intent is different from its specification. The latter is concerned with a required input-output response of the component and assumptions about its operational environment. The intent of the component is concerned with its desired behaviour when operating as part of the entire system.

Furthermore, the design of a system may not implement its intent fully. In other words, it may assign intents to the individual components which do not together guarantee the overall intent of the system. This thesis calls this "design limitation". In this chapter it is assumed that system design always implements the intent the issues of limitations are addressed in Chapter 5.

The WBS example, presented in the previous section, can be used to illustrate the notion of intent and its decomposition. The overall intent of the system is to provide braking pressure to the wheel assembly whenever the pressure is commanded by the pilots. The top-level design identifies key

components of the system (BSCU, pumps, valves and pipework) and assigns an intent to each of them. Thus, the intended behaviour of the meter valves is to provide pressure to the braking assemblies whenever braking is commanded by the pilots. This, of course, is not the same as specification of the valves, which merely mandates that the valves must open when a respective electrical command is received on their electrical terminals.

Similarly, part of the intent of the BSCU is to provide braking commands to the green channel when braking is requested, and to report (on its validity output) if it is unable to provide these commands. The design of the BSCU decomposes this intent and allocates it to the individual components. However, as will be demonstrated in chapter 5, this design violates the overall intent of the BSCU.

Failure logic modelling's dependence on the notion of design intent is a key characteristic which distinguishes it from other model-based safety assessment methods. It can be seen as both the strength and the weakness of the approach. Whilst the informal nature of the intent does not allow for the automated generation of failure logic models from the formal models of system design, it facilitates a thorough review not only of the design of the system but also of its justification. To construct failure logic models, analysts not only have to consider what a particular component would do in response to particular interactions initiated by its environment but also to analyse what the component was intended to do had the system and its environment not suffered from any malfunctions. Whilst this difference in perspectives may seem subtle for trivial components (like the meter valve) it becomes significant for more complex components and in the context of more complex systems (especially reconfigurable systems). Overall, the difference in perspective ensures a thorough review and understanding of the system design by safety engineers.

3.2. Unifying Metamodel for Existing Techniques

In this section a unifying metamodel for the existing failure logic modelling techniques, such as HiP-HOPS, FPTN and FPTC, is presented. Pidcock defines a metamodel as "*an explicit model of the constructs and rules needed to build specific models within a domain of interest*" [121] and lists three possible perspectives in which metamodel can be viewed:

(1) "as a set of building blocks and rules used to build models"
(2) "as a model of a domain of interest"
(3) "as an instance of another model"

This thesis is concerned with the second objective of metamodelling and the Failure Logic Metamodel could in principle be called a "Failure Logic Domain Model".

As used in model-driven engineering, a metamodel, typically used to describe a modelling language, is expected to define [15]:

- "the concepts from which models are created" (an abstract syntax)
- "the concrete rendering of these concepts" (concrete syntax)
- "rules for the application of the concepts" (well-formedness rules)
- "description of the meaning of the model" (semantics)

However, this thesis is specifically not concerned with a definition of the specification language (instead, a general-purpose third-party language is utilised); consequently the Failure Logic Metamodel does not define any concrete syntax.

The metamodel presented in this chapter generalises the existing failure logic modelling methods through separation of the notions of failure and failure state. The key FLM concepts are defined (providing the engineering semantics 'component' of the metamodel) and their relationship is shown diagrammatically (providing the abstract syntax). The well-formedness rules are omitted from the main text of the chapter, but are listed in Appendix A.

Whilst the FLMM has been defined using the Eclipse Modelling Framework (EMF) [46, 142] for ease of the presentation relationships between FLM concepts are shown as UML class diagrams.

Elementary Concepts

To avoid ambiguity, it is necessary to define the context of failure logic modelling. Failure logic models are models of a system which consists of a number of components.

The previous chapter has defined the system (based on [147]) as:

> **System:**
> *A bounded physical entity that achieves in its domain a defined objective through the interaction of its parts.*

Based on the same source we define:

> **Component:**
> *A discrete structure, such as an assembly, within the total system considered at a particular level of analysis.*

Using these definitions, it is observed that components within the system *interact*. Each possible interaction is associated with some *attribute* (physical or logical property) of a transfer of *energy, material or information* [112]. For example, components in a hydraulic circuit may interact in

terms of pressure, flow/displacement of the hydraulic fluid, or even the temperature of the fluid. Similarly, software components may interact in terms of flows of data as well as in terms of flows of control (e.g. synchronous communication protocols). Interactions can be either intentional or unintentional (e.g. short circuits in the electrical system) [54].

3.2.1 Common Key Concepts

All of the failure logic modelling methods examined in this thesis are based on a notion of interdependent components, where dependencies are specified in terms of the deviation of components' interactions from the design intent. Each component of the model may exhibit deviation or may be sensitive to deviations in its environment (i.e. interactions initiated by other components), yielding notions of Input and Output Failure Modes.

Output Failure Mode:
A particular and externally observable deviation from the intent of a particular interaction initiated by a particular component (including the initiation of an unintended interaction)

A component is said to be *exhibiting* or *generating* output failure modes (or, for short, just 'failure modes' or 'FMs').

Input Failure Mode:
A sensitivity of a component to a particular deviated interaction initiated by some other component in its operational environment (including the initiation of an unintended interaction).

A component is said to be *sensitive* to its input failure modes: when a component exhibits a failure mode, all other components that are sensitive to that failure mode are said to be *exposed* to the FM. In terms of failure logic models, input and output failure modes form the *interface* of a component.

Components in the model can 'communicate' through these interfaces, yielding a notion of ***FM Flow*** *– a binding between an input FM of one component and output FM(s) of one or more other components* – which signifies a dependency between the components established by the system architecture (and implementation technology). FMs and FM flows are associated with particular possible (intentional or unintentional) interactions between components which are often identified in design models (typically as connectors between components which characterise flows of energy, matter and/or information). However, the *nature* of a deviation can be captured

independently of the *particular* interaction of *particular* components yielding the notion of a Failure Mode Class.

Failure Mode Class:
A general definition of a deviation of an interaction. Generalisation of a failure mode.

A Failure Mode Class may or may not be specific to the physical or logical nature of the components' interactions. In general, most failure mode classes can be attributed to one of the following three abstract categories [105, 123]:

(i) *Provision* – e.g. the inability to initiate (or sustain) an interaction when (or, respectively, while) required to do so or the initiation of an interaction when not intended (including initiation of the altogether unintended interaction)

(ii) *Timing* – e.g. the initiation or the termination of an interaction in an untimely fashion, maintaining of the interaction for too long or too short a period of time, etc.

(iii) *Value* – quantitative deviation of some physical (or logical) attribute of the interaction.

Depending on the required level of granularity of the failure logic model and the level of detail of the design description available to the safety engineer, a particular model can use abstract FM Classes (such as "Omission" and "Commission") or more concrete classes derived for a particular technology (e.g. "Leak" and "Short circuit" for hydraulic and electrical systems respectively) or even application-specific classes (such as "False Positive" and "False Negative" for flows of diagnostic information). However, for the purpose of constructing the Failure Logic Metamodel, FM Class is treated as an elementary concept (metaclass).

Finally all failure logic modelling techniques recognise that components are not only sensitive to their environment but may also themselves malfunction and, thus, 'generate' or contribute to their output FMs. This yields a concept of "failure".

Failure:
An accidental and undesirable momentary event which is spontaneous and internal to the component and which may have a persistent detrimental effect on the component's ability to fulfil its operational requirements.

Failures can be seen as being elementary to the failure logic model behaviour; they are the ultimate causes of FMs. Typically assumed to be stochastic in nature, failures can be associated with some probabilistic characterisation (e.g. a particular probability distribution function). Also, it is important to note that whilst Failure Modes are defined with respect to the intended behaviour of the component in the context of the entire system, failures are related to the component's 'local

intent' or specification and design (if it can be assumed that these do not contain systematic errors).

The above concepts define part of the Failure Logic Metamodel (Figure 22) which does not (yet) cover the *behaviour* of the components in the failure domain.

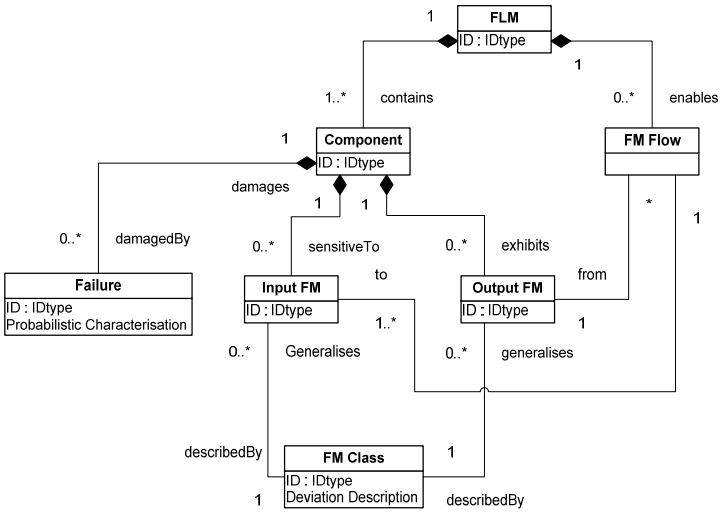


Figure 22 - Partial FLMM: Elementary Concepts

3.2.2 Component Failure Logic

The behavioural specification of the FLM components must show how output failure modes are related to (i.e. caused by) combinations of input FMs and internal failures. Whilst HiP-HOPS links failures (which it terms "malfunctions") directly to the output failure modes, a more principled and structured approach is adopted in this thesis and in the metamodel presented here. It is observed that, whilst failures may have lasting effects, they are themselves momentary phenomena. Furthermore, two or more different failures may have the same effect on the component behaviour even though they themselves have fundamentally different physical (and probabilistic) natures. Finally, the effect of a failure may not immediately cause an output failure mode and may remain dormant (e.g. only resulting in a deviation of behaviour when combined with another failure or an input FM). To capture the immediate effects of failures on a component, the concept of the Failure State is introduced.

Failure State:

An abstract representation of the undesirable persistent condition of the component which, in certain circumstances, has a particular effect on the component's ability to fulfil its operational requirements.

Determination of the failure states of a component is reliant on engineering judgement during definition of the models. Failure states – unlike failures – do not necessarily relate to anything 'real' (such as the description of a physical state of component if it was to be examined after failure) – they may simply encapsulate the effect of failures at the level of abstraction necessary for a particular failure logic model.

To model the failure behaviour of the components, causes of both the Failure States and the output FMs need to be specified. To allow this specification each Failure State in the metamodel is associated with one or more *transition* specification(s) which show(s) the circumstances under which a particular failure may lead to a change in the state. Every such entity contains a specification of a *trigger* (a component failure) and of the circumstances under which a state transition is possible (a *guard*). In general a *guard is a propositional logic sentence over (other) failure state and input failure mode propositions*. The guard allows for the specification of cases in which certain failures only have an effect on component when combined with an input failure mode or when following another failure of the same component. It can also be used to model situations where the presence of a particular input failure mode makes certain failures more likely[25] or, on three contrary, where an input failure mode leads to an 'incidental immunity' from a failure. Finally, a special case is a *void* guard (a guard which always evaluates to *true*), which places no restrictions on entry into a failure state upon occurrence of a particular failure.

As with state transitions' guards, the FLMM associates each output Failure Mode of the component with a *propagation condition* which, like a guard, is a propositional logic expression over input failure modes and failure states. It is important to note that both guards and propagation conditions may include negative as well as positive assertions.

3.2.3 Model Structure and Hierarchical Organisation

The concepts presented so far are, in principle, sufficient to allow the specification of failure logic models equivalent to any HiP-HOPS or FPTN model. However, the Metamodel lacks facilities for the hierarchical organisation of the models that these concrete methods offer. Therefore, to facilitate more efficient model construction and management it is necessary to enhance the

[25] This can be achieved by specifying two failures associated with different probabilistic characteristics but in principle leading to the same failure state (through two separate "state entry logic" specifications). The guard can be used to select the applicable failure for the given (current) failure state and/or input FM(s) of the component. Note that the metamodel essentially "forces" engineers to specify separate failures for each unique probabilistic characterisation of failure associated with different scenarios of component exposure to internal and external threats. This is justified by the fact that evidence supporting such different probabilistic characterisations would be gathered from different sources (e.g. different tests) requiring different assumptions to be reviewed and verified. Therefore each probabilistic characterisation should be treated as a conceptually separate entity.

metamodel with three concepts – *State Space*, *Complex Component* and *Failure Mode Group* – which allow structuring of the failure logic models. The extensions allow the organisation of:

- *Failure States into State Spaces.* Some of the failure states may be mutually exclusive and thus the grouping of such states (along with a special *privative state*) into orthogonal state spaces permits the more efficient and elegant specification of state entry logic (e.g. avoiding some of the repetitive negations of other states in the space). Furthermore, this construct permits intuitive implementation in general-purpose specification languages (addressed later in this chapter), since state spaces can naturally be represented as enumerated variables or parallel state charts. Within each state space, one failure state (typically – the privative) must be marked as an initial state of the component.
- *Components into Complex Components.* Complex components represent collections of components (either basic or complex in themselves) and FM flows between them. In this chapter, it is assumed that complex components have no failures or states of their own (beyond the states of the basic components they contain). Whilst complex components may have input and output Failure Modes these are propagated to or from their constituent components by FM flows and are not associated with any propagation conditions. Note also that the introduction of complex components requires some revision of the FM Flows in the metamodel since it can no longer be assumed that flows connect output and input FMs; for example, a flow may be established between an input FM of the complex component and an input FM of a basic component it contains (Figure 23).
- *Failure Modes into FM Groups.* Input and output failure modes may be grouped according to the interaction, flow or a connector in design model that they are related to. For example, in an electrical system failure modes related to voltage and current deviations can be organised into two groups. These can be also grouped together (forming a hierarchical structure) to reflect the fact that they relate to the same electrical terminal.

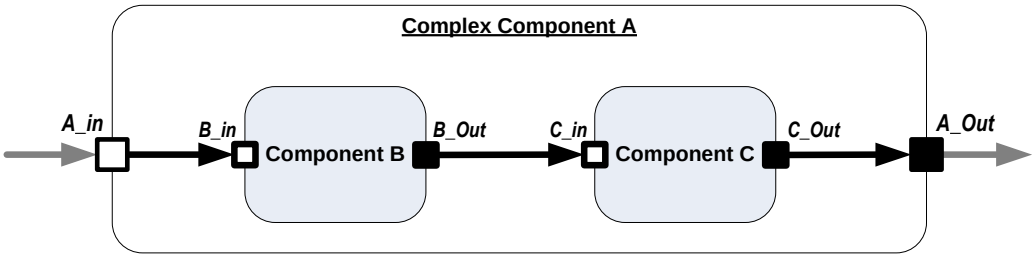


Figure 23 - Permissible Flows in Hierarchical Failure Logic Models

The resultant basic Failure Logic Metamodel is shown in Figure 24.

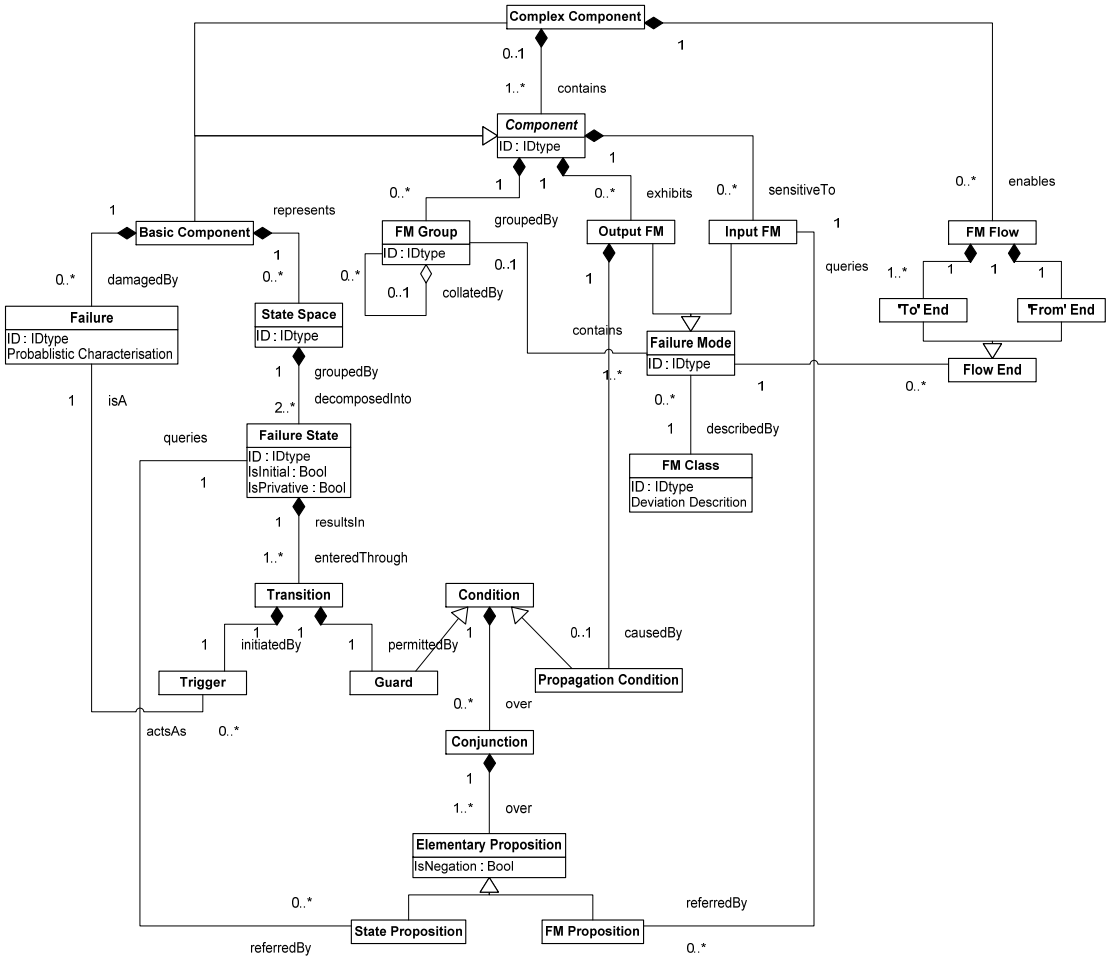


Figure 24 - Basic FLMM: Model Structure and Component Behaviour

3.2.4 Illustration

To illustrate some of the FLM concepts defined above the *Green Meter Valve* of the WBS is used. The valve has three groups of failure modes related to its control input (from the BSCU) as well as its hydraulic input and output.

Considering hydraulic input first, three input failure modes are identified: *Omission of Pressure*, *Too Low Pressure* and *Commission of Pressure*. Whilst the failure mode classes of these input FMs are trivial, the semantics of the commission failure mode highlights the significance of the design *intent* discussed in the introduction. In principle, hydraulic pumps are expected to supply the pressure to the WBS in all circumstances and they therefore cannot exhibit a commission FM themselves. However, the intent of the system design is that the *Shut-Off Selector Valve* (located upstream of the Green Meter Valve considered here) cuts off hydraulic supply whenever the *BSCU* control outputs become untrustworthy (e.g. due to failures of both channels). In such circumstances, the Shut-Off Selector Valve will not exhibit (and the meter valve will not be exposed to) an Omission FM since its behaviour, whilst depriving the green line of pressure, *is* compliant with the intent. However, if the Shut-Off Valve *does not shut-off the supply* when it is

expected to (e.g. due to internal failure or due to being exposed to a failure mode of the BSCU Validity Monitor) it will exhibit a *Commission of Pressure* output FM.

The above three input failure modes are organised into a single FM Group (*HydIn*) to indicate that they relate to the same design model flow. This group also contains one output FM – *Leak* (whose class is a *technology-specific* refinement of the abstract *commission* class).

Turning to the valve's hydraulic output, the component has *HydOut* FM Group, which contains a similar set of Failure Modes as *HydIn* (although the commission here has a different, more trivial, interpretation). Furthermore, this group also contains an additional output FM – *Pressure Too High*. The final group of failure modes (*Ctrl*) is formed by input FMs associated with the control input – *Inadvertent Braking Command*, *Lack of Braking Command*, *Lack of Antiskid* and *Too Little Braking*.

Internally, the valve is assumed to be spring-loaded to its "shut" position and driven to "open" by an electric motor. Internal failures of the valve may affect its operation. In particular the valve may suffer from:

- Spring failure (*SpringSnaps*) or Mechanical Jam in the "open" position (*MechanicalJamOpen*) leading to it becoming stuck in the "open" position (*StuckOpen* failure state)
- Motor failure (*MotorShaftSnaps*) or a jam in the "closed" position (*MechanicalJamClosed*) leading to the *StuckClosed* failure state
- Contamination of the moving parts (*Contamination* failure) that makes the valve less responsive to control commands (*Struggling* failure state)
- Fracture of the casing (*Rupture*) that leads to its developing a leak (*Leaking* failure state)

The Failure States of the valve are organised in two orthogonal groups (with an additional *OK* privative state in each) as shown in Figure 25. Note that for this particular component the transition guards do not include any FM propositions; there is no guard for the *Leaking* failure state, guards of *StuckOpen* and *StuckClosed* merely require the component not to be in the other stuck state (i.e. they each contain a single negative state proposition) and the guard of the *Struggling* state is a positive proposition over the privative failure state (which is equivalent to a conjunction of negations of the two stuck states).

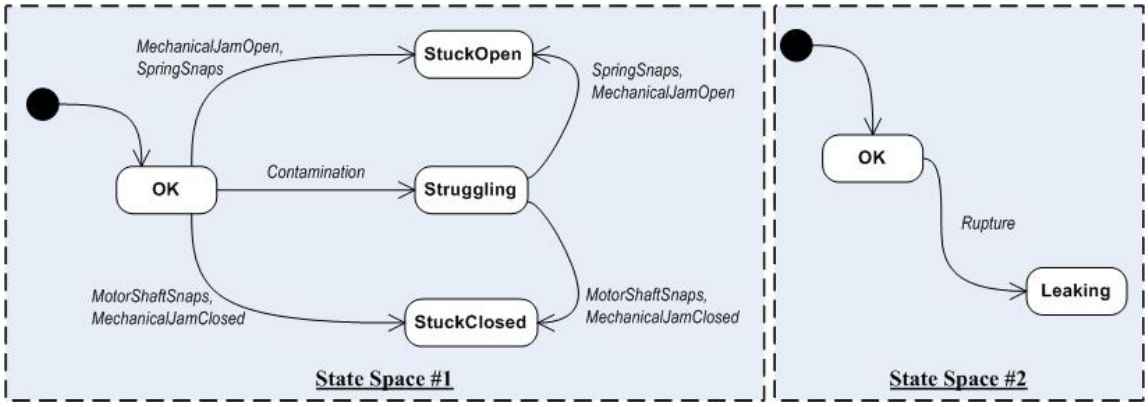


Figure 25 - Failure State Spaces Illustration: Green Meter Valve

Finally, FM propagation conditions can be illustrated with an example of the *Omission of Pressure* output FM of the valve. The FM can be caused internally by the valve being in either *StuckClosed* or *Leaking* failure states. Externally to the valve, the FM can be caused by an *Omission of Pressure* input FM over the hydraulic input or a *Lack of Braking Command* – over the control input (and provided the later is not combined with both a *Commission of Pressure* input FM and the valve's being *StuckOpen*). Furthermore, in the absence of a *Commission of Pressure* input FM, the valve will convert all control input FMs into *Omission of Pressure*. The propagation condition can therefore be formalised as follows:

$$
\begin{aligned}
&HydOut.PressureOmission = \\
&\quad StuckClosed \vee Leaking \vee HydIn.PressureOmission \ \vee \\
&\quad (\ Ctrl.LackOfBraking \wedge \neg(\ StuckOpen \wedge HydIn.PressureCommission\ )\ ) \ \vee \\
&\quad (\ (\ Ctrl.InadvertentBraking \vee Ctrl.LackOfAntiskid \vee Ctrl.TooLittleBraking) \\
&\qquad \wedge \neg HydIn.PressureCommission)
\end{aligned}
$$

The complete characterisation of the valve is included in Appendix B1.

3.3 Extended FLMM: Dynamic and Normal Behaviour

This section discusses the limitations of the "Basic Failure Logic Metamodel" presented in the section 3.2 above (and, by implication, limitations of the existing failure logic modelling methods). In response to these observations, metamodel extensions are defined to overcome these limitations.

The extensions are concerned with modelling the normal and the dynamic behaviour of components. Indeed, the basic metamodel allows for some dynamic behaviour through the notions of persistent failure states on the one hand and potentially transient failure modes on the other. Furthermore, the notion of the state guard permits particular failures (or, more formally, their effects) to be enabled or disabled, depending on the circumstances (e.g. the input FMs components are exposed to). In this section, the notion of state is generalised and two new

specialisations of states (in addition to failure states) are introduced – *Failure Handling States* and *Normal States*.

It is also observed that, whilst failure logic modelling methods have originated to a large extent as modular extensions of traditional Fault Tree Analysis, in comparison to the FTA they lack facilities allowing the specification of non-failure events that are nevertheless significant for modelling the behaviour of the system in the failure domain. To remove this limitation, the notions of *Normal Events* and *States* (mentioned above) are introduced.

However, the section first revisits the relationship between the Input Failure Modes and Failure States of the model components.

3.3.1 Void Transition Triggers

The FLMM presented so far has reflected the fact that the effect a component failure (a failure state) has may depend on whether the component is exposed to any input failure mode at the time of failure. The same feature of the metamodel – the transition guard – could be used to model situations where a component failure is more likely if the component is exposed to a particular failure mode (e.g. an electrical generator being exposed to a higher load than normal). However, so far, failure state transitions can only be triggered by some failure of the component, whilst input failure modes on their own can only have a transient effect on component output FMs.

This restriction of the persistence of the effects of input FMs cannot be justified in the context of realistic system designs. Whilst components may propagate most of the FMs without sustaining permanent damage, certain failure modes – such as “out of bounds” value deviations and unintended interactions (e.g. leaks or short circuits) – may lead to persistent or even permanent effects on the component itself. Without the ability to reflect this dependency of the failure state of the component on an input FM (without the need for a further failure) the failure logic models can be considered incomplete.

This limitation of the Metamodel can be demonstrated on the failure logic characterisations of the Green Hydraulic Pump of the WBS. As was implied in the illustrative example in the previous section, the pump can be exposed to a *Leak* failure mode (originating from the downstream section of the green hydraulic line). The effect of the leak is depletion of the pump’s reservoir, which eventually leads to loss of pressure. In principle, for the purpose of a naïve WBS model, this can be modelled through the pump’s propagation condition whereby *Omission of Pressure* output FM can be caused either by a *TotalLoss* failure state of the pump (only reachable through some internal failure(s) of the component) or by exposure to the *Leak* input FM. However, such a characterisation could be misleading in the wider aircraft context (where other systems may

depend on the pressure supplied by the pump) or if analysis of the failure logic model was to be used to search for possible mitigation measures that could be 'built into' the system design. In particular, simulation or a walk-through of such a model would show that if, following pump exposure to the leak, the Shut-Off Selector Valve were to fail in a closed position, the pump would stop exhibiting the Omission FM (since it would no longer be exposed to the leak). This is clearly inaccurate, and the design modification that ensures that the green line is cut off from the hydraulic pressure supply after the pressure in the system is lost (suggested by the above simulation scenario) is unlikely to serve any useful purpose.

A more adequate representation of the failure logic of this component should move the pump into a *TotalLoss* failure state upon exposure to the *Leak* to indicate the persistence of the effects of this input FM. In general, the FLMM must allow Failure State transitions to be caused by input failure modes (or combinations thereof) without requiring any failures. In other words, the *metamodel must be extended to allow void transition triggers* (Figure 26). The semantics of the transition specification with the void trigger is that the corresponding failure state is entered as soon as the transition guard becomes *true*.

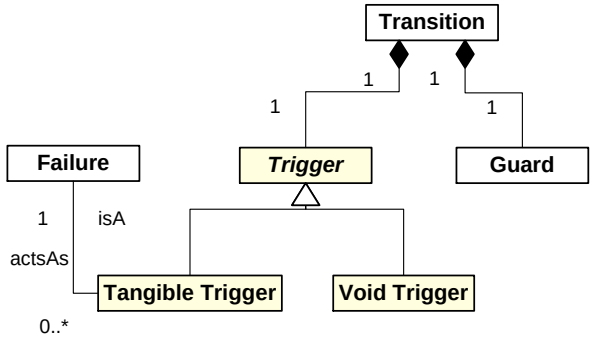


Figure 26 - Void Transition Triggers (in the Immediate FLMM Context)

This extension can be seen as a natural extrapolation from the ability to 'modify' the likelihood of failure depending on whether the component is exposed to input failure modes – in some cases the probability of failure must be assumed to be 1. However, it is different from specifying a different component failure with a probability equal to 1, since in the circumstances described above, no 'qualitative advantage' of a need for any further failure (e.g. of a pump itself in the previous example) can be justified.

Finally, a further justification of extending the Failure Logic Metamodel to allow model components essentially to 'memorise' the history of input failure mode exposures will become evident in Chapter 5, where system reconfiguration is discussed.

3.3.2 Normal Events and States

As was mentioned in the introduction to this section and in the previous chapter the failure logic modelling approach is closely related to the traditional Fault Tree Analysis since both techniques model behaviour of the systems in the failure domain. However, failure logic modelling techniques have so far adopted a significantly more 'purist' approach to modelling this behaviour. In particular, whilst FTA includes a number of facilities (such as External and Conditioning Events) to condition the failure logic of the system on some key aspects of the "success space", all of the elementary FLM concepts introduced at the beginning of the previous section refer exclusively to undesired events and conditions (i.e. to the "failure space").

Whilst too frequent and unnecessary reference to the normal behaviour of the system – such as the precise state of the components (e.g. valves being closed or open) – may undermine the abstraction that both FTA and the failure logic modelling approach rely upon, such references are not always avoidable. In particular, the failure logic of components which rely on a finite and consumable resource (e.g. electrical batteries, hydraulic accumulators and fuel tanks) cannot be modelled without reference to the presence or the exhaustion of that resource. The latter cannot always be considered a "failure" and, in safety analysis, no qualitative advantage can be taken for such normal exhaustion events.

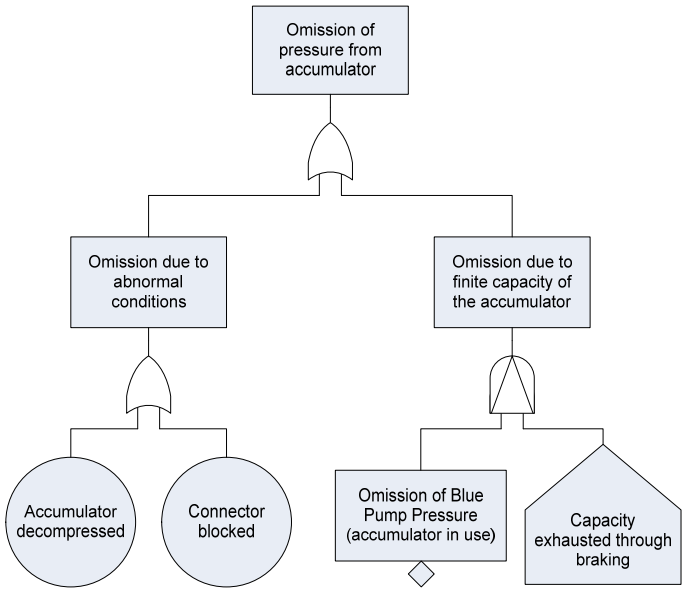


Figure 27 - Fault Tree Segment for Unavailability of Pressure from the Accumulator

This limitation of the failure logic modelling techniques can be demonstrated with the example of the hydraulic accumulator of the WBS, by considering a failure scenario in which both hydraulic pumps fail. According to the WBS description, braking will not immediately be lost and the accumulator will provide sufficient pressure for a safe landing. However, if the pilots were to execute an aborted landing manoeuvre, the accumulator could be depleted leading to a complete

loss of wheel braking on the second landing attempt. Whilst the failure logic model of the system *should* show that the accumulator will produce an omission of pressure, at this stage the system has not suffered from any further failure after both pumps were lost (since touch-and-go is in fact an intended manoeuvre that aircraft are designed to execute). The correct failure logic of the accumulator can be illustrated through a simple fault tree as in Figure 27 (above), but is beyond the expressive power of the Failure Logic Metamodel constructed so far.

To 'match' the expressive power of the FTA, the FLMM must be extended with two new concepts – *Normal State* and *Normal Event* – defined as follows:

Normal State:

An abstract representation of a persistent condition of the component which may, under certain circumstances, occur in the course of the intended operation of the component but which may have a significant effect on component behaviour in the failure domain.

Normal Event:

A momentary event which, under particular circumstances, can normally be expected to occur.

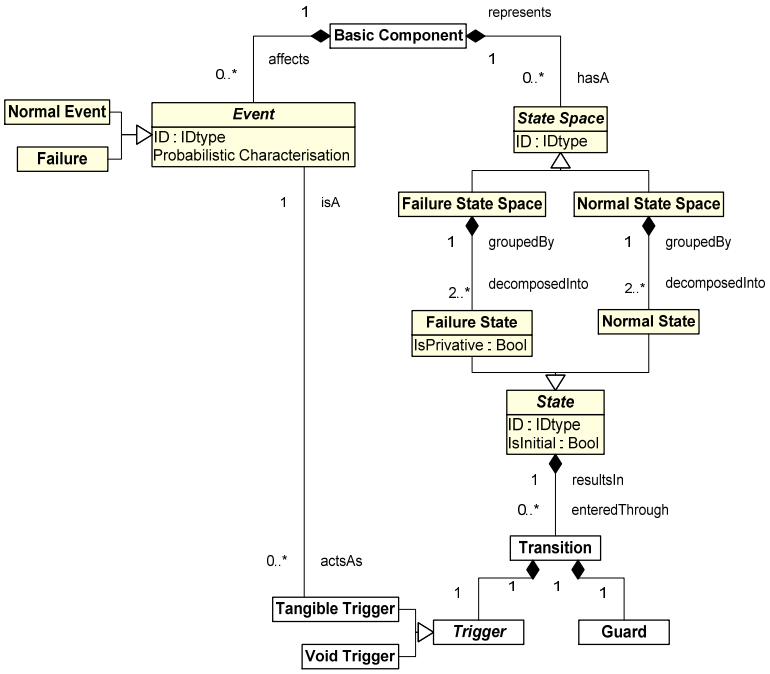


Figure 28 - Normal States and Events (in the Immediate FLMM Context)

It is observed that Normal States have some common characteristics with Failure States – they are organised in state spaces and entered through similarly structured transitions. Similarly, Normal Events share some features with Failures – both may trigger state transitions and both can in

principle be assigned probabilistic characterisations. These commonalities are captured in abstract *State Space*, *State* and *Event* metaclasses in the revised FLMM (Figure 28, above)

3.3.3 Failure Handling States

Allowing for limited representation of normal as well as failure behaviour of components greatly improves the expressiveness of the failure logic models. However, system designs often foresee certain known threats associated with particular implementation technologies (e.g. leaks and short circuits associated with hydraulic and electrical systems respectively) and call for dedicated components to mitigate them. The behaviour of these 'failure handling components' often cannot be classified intuitively as either 'failure behaviour' or 'normal behaviour'.

Examples of such components include leak arrestors and (overcurrent) circuit breakers. These components are designed to 'trip' on exposure to particular failure modes; in a tripped state failure handling components prevent propagation of the threat often at the 'cost' of exhibiting another output Failure Mode (often – an omission) which is deemed less harmful by systems engineers. For example, a circuit breaker in electrical power distribution systems prevents propagation of short circuits at the cost of cutting off electrical power supply to the network segment affected by the short circuit in the first place.

On the one hand, it is unintuitive to consider a tripped state of a leak arrestor or a circuit breaker as a failure state. These components are *designed, expected and required to trip* when facing leaks and short circuits. Furthermore, these components may suffer from internal failures that either prevent them from tripping when they should or make them trip when they shouldn't. Clearly it would be beneficial to separate the latter failure behaviour from the former 'designed in' behaviour.

On the other hand, tripped states cannot be intuitively considered as components' Normal States: they are never expected in a failure-free system and, after tripping, components often exhibit a deviation from the overall system intent (which may, in some circumstances, have hazardous consequences).

Therefore, it is beneficial to enhance the Metamodel with a separate notion of a *failure handling state*. Typically, the trigger of such a state will be *void* and a functioning component will enter the state as a result of an input failure mode (both the circuit breakers and the leak arrestors mentioned above would follow this pattern); however, in some cases failure handling states may be triggered by normal events. The later may be necessary, for example, to represent the time it takes for a component to detect and react to a dangerous condition (or, alternatively, representing

a non-perfect coverage of failure handling mechanisms). During this period of time, the system may be particularly vulnerable to additional failures.

Introduction of normal and failure handling behaviour into the FLMM has inevitably permitted a certain degree of redundancy of the models (since normal and failure behaviour is not fully orthogonal). This redundancy needs to be managed carefully by the safety engineers, to avoid inconsistent or incoherent specification of the component behaviour.

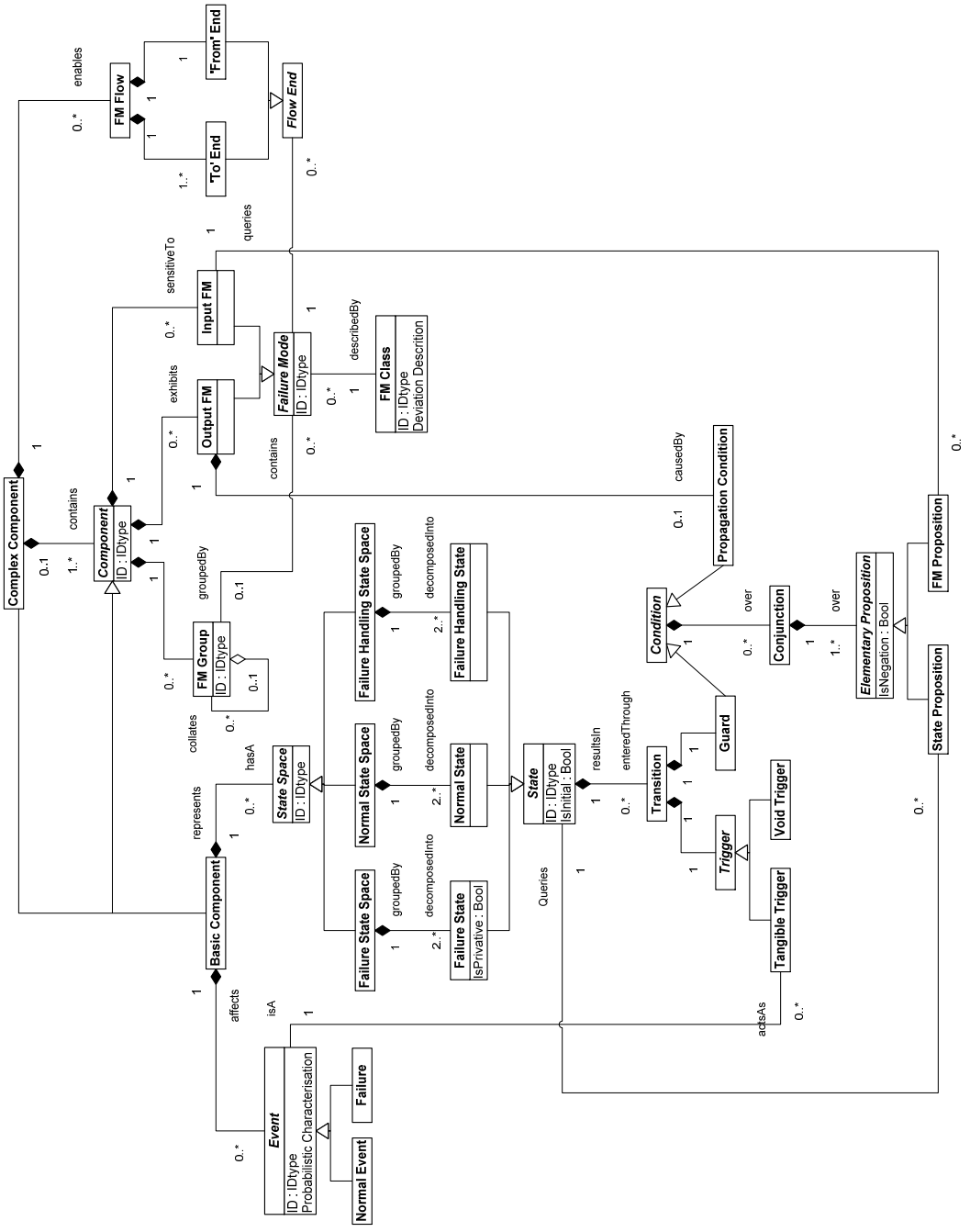


Figure 29 - Complete Baseline FLMM

This section concludes with the diagrammatic representation of the complete Failure Logic Metamodel defined in this chapter (Figure 29, above). The model is reproduced in Appendix A

along with all associated well-formedness constraints[26]. The constraints are divided into "hard" and "soft", whereby hard constraints exclude fundamentally inconsistent models (for example, a specification of a component state space with either more or less than one initial state) whereas soft invariants exclude structures judged as atypical by the author (e.g. failure state transitions with a void trigger and a guard that can be satisfied without the component's being exposed to any failure mode). Violation of the later invariants indicates highly unusual behaviour and verification of these constraints may partially contribute to the validation and review of the failure logic models.

Finally, Appendix B1 presents a complete failure logic model for the Wheel Braking System recorded in a tabular, specification-language-independent, 'pseudocode' form.

3.4 Model Analysis

This section briefly discusses some of the approaches to the analysis of failure logic models.

3.4.1 Fault Tree Synthesis and Sequence Generation

The original failure logic modelling techniques – such as HiP-HOPS and FPTN – focussed on fault tree generation (synthesis) as means of analysing the models. Indeed, the failure logic characterisation of each basic component in a system can be seen as being broadly similar to a forest of small local fault trees. Figure 30 shows an example of a local fault tree for the *Omission of Pressure* output failure mode of the WBS's Green Meter Valve. Whilst the tree was constructed manually, it was derived from the FLM characterisation of the valve (Appendix B, pages 285-286) semi-algorithmically and reflects a typical outcome of an automated fault tree synthesis algorithm. To maintain correspondence with the original model the fault tree notation is relaxed: negation is introduced, gate outputs are allowed to feed directly into gate inputs as well as forming a direct connection between two events (the later two 'anomalies' can be removed by a relatively trivial post-processing of the tree).

FM flows between components effectively establish equivalences between the top events of one local tree and the undeveloped events in one or more other trees. Consequently, provided that failure logic models are recorded in a structured, consistent and computer-readable form, it should be possible to traverse (parse) models recursively and to construct FT-like representations.

[26] Note, however, that both the metamodel and the constraints described in Appendix A reflect extensions introduced in the following two chapters of the thesis.

However, a Fault Tree representation of the model can only be guaranteed to exist under a set of restrictions of the FLMM. These necessary restrictions include:

(i) An absence of Failure Mode flow *loops* in the failure logic model architecture;

(ii) Fully orthogonal failure states (i.e. for every failure there must exist a distinct failure space with exactly two states and with the initial state being a privative);

(iii) An absence of recovery transitions from failure states to privatives.

As well as being theoretically infeasible in certain circumstances, it is not entirely clear why fault tree synthesis is desirable. Fault Trees are models of system behaviour that is an *alternative* to failure logic modelling. Further, as fault trees for realistic systems are very large, such a synthesis cannot be considered to be "analysis of the model" – but rather a *transformation of one model representation into another*. Furthermore, synthesised fault trees frequently cannot be easily reviewed without reference to the original model. In particular, being automatically generated, these trees *cannot* contain intermediate events that are not readily available in the model whilst they *may* potentially contain semantically redundant logic (which is nevertheless correct syntactic parsing of the model).

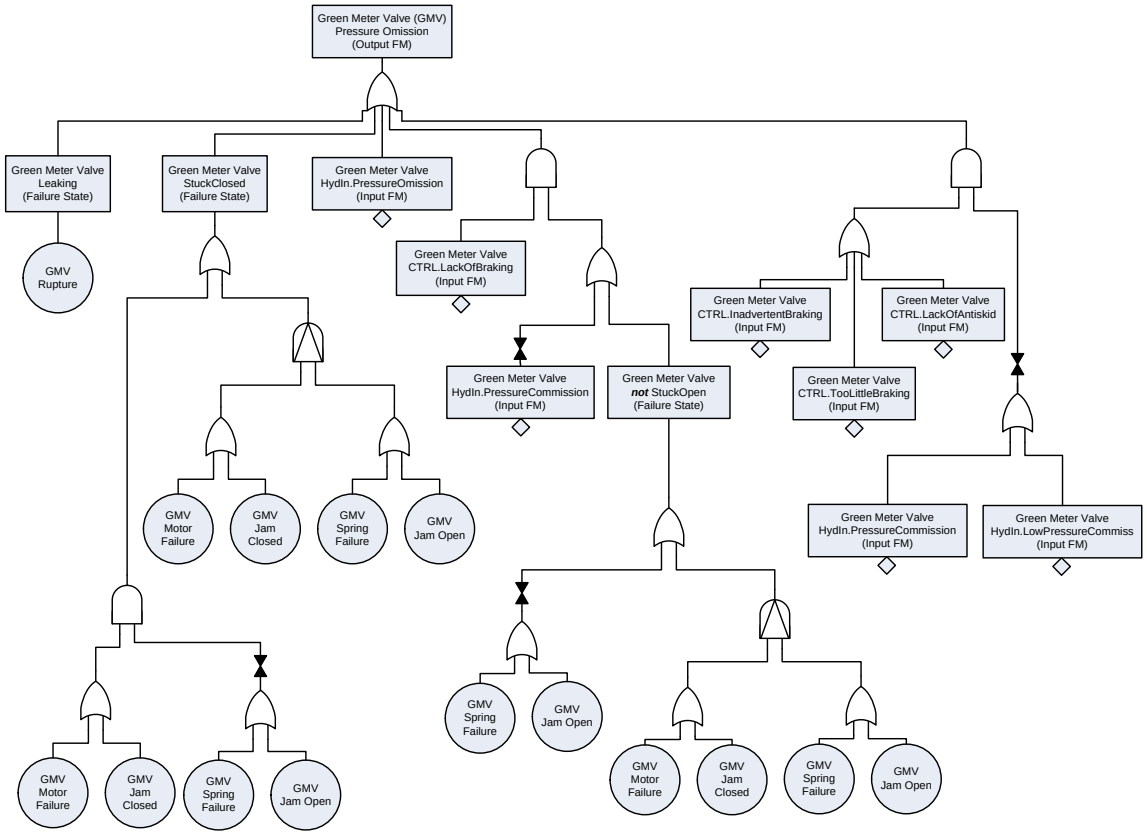


Figure 30 - Possible Fault Tree for Pressure Omission Failure Mode of the Green Meter Valve

Figure 30 can be used to illustrate this last criticism as the sub-tree for *Green Meter Valve StuckClosed (Failure State)* intermediate event is not intuitive to a human reader. The left branch of this sub-tree essentially states that the component may move into this state as a result of either

GMV Motor Failure or *GMV Jam Closed Failure* provided that no failure that leads to the valve getting *StuckOpen* (i.e. *GMV Spring Failure* or *GMV Jam Open*) takes place.

However, on its own this scenario is too restrictive. The valve could end up in the *StuckClosed* state even if the failures that could, in principle, cause it to enter the *StuckOpen* state have happened, but provided such failures have happened *after* the valve has already moved to the *StuckClosed* state. In other words, if failures leading to *StuckClosed* (i.e. *GMV Motor Failure* or *GMV Jam Closed Failure*) happen *before* failures that potentially lead to *StuckOpen* (i.e. *GMV Spring Failure* or *GMV Jam Open*) the resultant state is, in fact, the former *StuckClosed*. This additional scenario is captured by the 'Priority AND' gate of the right branch of the sub-tree.

Whilst being a correct fault tree parsing of the original failure logic model of the valve, the resultant tree, is not only unclear but also contains an apparently contradictory condition – a conjunction between *GMV Jam Closed* and *GMV Jam Open* failures. The information that these two failures are contradictory is, however, purely semantic and cannot be recorded in (and, thus, extracted by any synthesis algorithm from) a failure logic model.

Finally, the synthesis of fault trees and their submission to certification authorities or independent safety auditors may, unless managed very carefully, give the illusion that a fault tree analysis (FTA) has been performed. With FTA being a comprehensive and structured analysis procedure – that is both well-documented and tried-and-tested in industry – this may contribute to unjustified confidence in the adequacy of the model.

At the same time, it should be noted that fault trees are themselves analysed to produce Minimal Cut Sets, quantitative estimations of the likelihood of top-level event and importance measures. As failure logic models hold similar information to the fault trees, it is possible to generate those results directly from the models by-passing the intermediate FT synthesis step.

Focussing on a qualitative analysis, instead of deductively traversing the model (from effects to basic causes), it is possible to search through the different permutations of the component failures, normal events and model-level input FMs systematically and to establish for each permutation, inductively, whether it leads to a system-level condition of interest. The collection of permutations that do lead to the undesired condition can be seen as equivalent to the list of minimal cut sets for that condition.

The search through the elementary causes of deviations can result in a brute-force strategy of iterative simulation for every possible permutation of failures contained in the model. However, for systems of realistic complexity the timing of the exhaustive analysis may be prohibitive and the search may need to be limited to a pre-set maximum cardinality of the minimal cut

sets/sequences. Alternatively, sequence generation may optimise internal model representations, yielding a more time-efficient analysis. For example, the ESACS and ISAAC projects [9, 24, 50, 73] have adapted standard model checkers for the task of sequence generation from models encoded in various specification languages such as StateMate and SCADE. One of the resultant analysis tools based on the NuSMV model checker [23] is currently being adapted (as part of the MISSA project [6]) to the AltaRica language and Cecilia OCAS tool discussed in the section 3.5.2 below.

3.4.2 Simulation and FMEA/FMECA

Claims are sometimes made about the ability of model-based safety assessment methods (whether based on failure logic or any other type of model) to automate FMEA or FMECA. Whilst it is true that it is possible to generate FMEA/FMECA *tables* automatically, such process in no way a substitute for Failure Mode and Effect Criticality Analysis.

Generation of FMECA tables from Failure Logic Models is essentially a sequence generation process as described above, conducted with respect to multiple conditions of interest. The search can be conducted to a maximum sequence size of one (yielding traditional tables) or to any larger size (yielding expanded tables and claims of FMECA "improvements"). However, the *targets of the analysis must be predefined*. This contradicts the explorative nature of the FMECA and its role as either a hazard identification technique or a tool for the validation of results of previously conducted hazard identification.

To conduct analysis comparable to FMECA, safety engineers need to *simulate* failure logic models systematically (provided that these models are expressed in a specification language and environment that facilitates such simulation) as well as to *consider and to classify* the effects of each simulation based on their experience, domain knowledge and engineering judgement. Although it is important to note that even in a simulation-driven analysis process, the model will inevitably restrict the scope of the assessment (compared to the traditional 'manual' and informal FMECA)

3.5. Instantiation of the FLMM

Previous sections have presented a unifying metamodel of the failure logic modelling approach and discussed some aspects of models in general. The purpose of the discussion so far was to identify a set of concepts that must be captured by the models and the relationships between those abstract concepts; the purpose of this section is to describe how the models *themselves* can be effectively and adequately captured.

As discussed in Chapter 2 (section 2.3), most of the existing failure logic modelling techniques introduce a custom notation for specifying the models. This section demonstrates that failure logic modelling does not require idiosyncratic notations and that the framework can be instantiated in a general purpose, well-defined third-party language. As discussed in Chapter 1, implementation in such a language can yield significant benefits in the industrial context. From the academic perspective, it allows the separation of concerns of the adequacy of the domain model from concerns about the adequacy and expressiveness of a particular language or notation.

3.5.1 Specification Language Requirements

As far as the specification language is concerned, the requirements posed by the Failure Logic Metamodel are relatively trivial. The specification language should:

- support the notion of components and allow the hierarchical decomposition of components;
- support the notion of communications between components (i.e. input/output failure modes, failure mode flows);
- support the specification of components' internal behaviour in terms of their reaction to communications received from other components and internal events (i.e. spontaneous or unbounded behaviour specific to a single component);
- support the notion of persistent local conditions (states);
- support annotations of modelling constructs, such as the assignment of probabilistic characteristics to internal events and descriptions of deviations to communications;
- support the designation of events as "failure" or "normal" and states as "failure", "normal" and "failure handling" in a way that is accessible to analysis tools.

Highly desirable, but in practice not essential, characteristics of a specification language include:

- the ability to declare an abstract type for communications. This is necessary to declare Failure Mode Classes explicitly and to allow for specification of the deviation once when it is declared as a type rather than for every single failure mode of a class;
- the ability to declare communications between components in a structured (hierarchical) fashion (to capture "failure mode groups" and, thus, better to reflect the relationships between the system design and failure logic models);
- a distinction between the concepts of momentary *events*, persistent *states* and potentially transient input and output *conditions*.

These basic requirements are fulfilled by a large number of mature specification languages of different paradigms. For example, Statechart [64] variants such as StateMate [69] and StateFlow

[144] naturally implement the notions of states and events and allow for the parallel composition of state charts (which may reflect the decomposition of components). Communications between different charts are achieved through events synchronisation, so the events would have to be used for representing failure modes as well as failures and normal events[27]. From a pragmatic perspective concerning the maintainability of failure logic models implemented in a Statechart or stochastic automata language, it is likely that orthogonal states of basic components (normal states, failure handling states and orthogonal groups of failure states) would have to be specified as separate charts; further charts may be necessary for specification of the propagation conditions. Therefore the decomposition of the overall failure logic model will, below a certain level (corresponding to Basic Components), no longer reflect the *structure of the system*, but instead will follow the *structure of the component behaviour specification* as shown in the metamodel. On the one hand this can be seen as unintuitive representation; however, on the other hand, this low-level structure is likely to be highly reusable from one component to another.

Similarly, the FLMM can also be implemented in dataflow-centred languages, such as Simulink [145] or SCADE [52]. Such languages also provide a natural facility for the specification of components and the hierarchical decomposition of structure. Further, they naturally model *dataflows* between the components – which is a natural and intuitive representation of FLMM's failure mode flows. Furthermore, these languages often allow for flows to have both structure and type – thus representing failure mode groups and classes respectively. Whilst dataflow languages often do not have intrinsic notions of event and state, both can be represented. Events – i.e. metamodel's failures and normal events – can be modelled as additional inputs of the components which, unlike input failure modes, are not connected to outputs of any other model components[28]. States can be implicitly reconstructed if a language permits loops in flows along with a "unit delay" or "latch" operators.

However, since neither Statechart nor Dataflow- languages seem to represent certain aspects of failure logic in an immediately intuitive way, using a hybrid of the two approaches might be beneficial. A combination of MATLAB Simulink and StateFlow would be an example of such a hybrid language.

As was discussed in sections 2.5.1 and 2.5.3 above, another language – AltaRica – is used in this thesis. The Dataflow and OCAS dialects of AltaRica allow a similar dualism between states/events and flows to that offered by the MATLAB solution (although integration between these two aspects of behaviour is seamless in AltaRica). The language is supported by the Cecilia

[27] This is the approach taken in the AADL Error Model Annex [56, 136] (see section 2.5.2 above).

[28] Depending on the precise syntax and well-formedness rules of a specification language these "free variables" may, however, lead to significant clutter in the models as reported by Joshi et al [76, 77]

OCAS workbench – an industrial-strength tool that provides a graphical user interface, facilitates definition of the models and allows simulation and analysis of the models. Finally, the analysis facilities of Cecilia OCAS [39, 131] have been specifically developed for the context of safety assessment (although not necessarily for failure logic models) and can readily be reused. Access to developers of the Cecilia OCAS – Dassault Aviation – has been a significant pragmatic consideration for selection of the tool.

3.5.2 Overview of Cecilia OCAS Tool

Cecilia OCAS provides a graphical interface for the definition of nodes (i.e. it reconstructs node parameter declarations from data entered by a user through the GUI as shown in Figure 31) and for the establishment of flows between the nodes (using a "drag-and-drop" facility between the input and output flows of the components). The tool also performs some syntactical and consistency checking on models and provides graphical facilities for the animation of models during simulation. Tailored to the needs of safety assessment, Cecilia OCAS allows for the association of events with probability laws and parameters (such as maintenance and inspection intervals – which relate to exposure intervals in FTA). In most cases, probability parameters are ignored by simulation and sequence generation tools. However, one specific "probability law" is an exception – *Dirac(x)*.

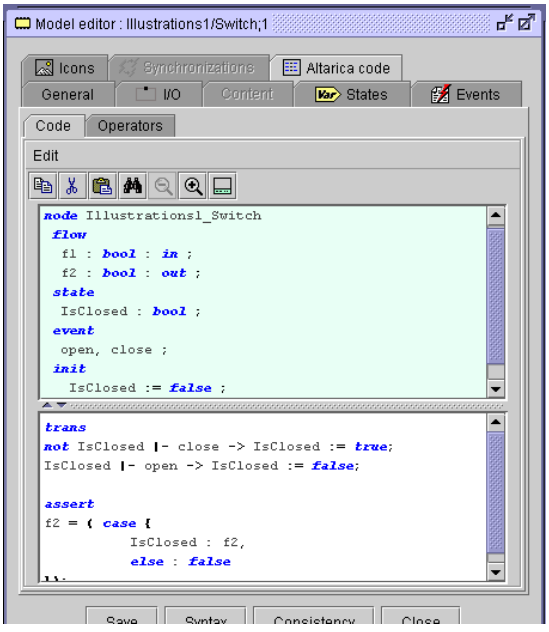


Figure 31 - Cecilia OCAS Interface: Behaviour Specification

Usually used with a parameter zero, the *Dirac* law allows for the specification of non-random events. Events marked with *Dirac(0)* are 'updates' which are fired as soon as the guards they are associated with become *true*. They allow for situations when component state(s) change only in response to input flows to be modelled. A higher parameter can be used to specify events that are

similarly non-random, but which are expected to occur some time after the guard of the associated transition becomes *true*. An important characteristic of *Dirac(0)* events in the context of model analysis is that they are not regarded as events: the model simulator triggers these events automatically at the earliest opportunity and the sequence generator does not include instantaneous events in its results.

In terms of the textual format of the AltaRica language, additional safety- and reliability-information – described above – is stored in a dedicated *extern* clause.

Cecilia OCAS implements a particular dialect of the AltaRica language (referred to in this thesis as AltaRica OCAS). Other dialects include AltaRica Dataflow [22] (which is very similar to the OCAS dialect), AltaRica LaBRI [13, 87], which is generally more expressive, and the temporal dialect introduced by Pagetti et al [28, 111]. Finally, it is important to note that OCAS introduces some additional terminology. Complex components, which are merely nodes in other AltaRica dialects, are termed "equipment" in OCAS; similarly, a special case of equipment – "system" – is explicitly introduced.

3.5.3 Representing FLM Concepts in AltaRica / OCAS

Notions of flow, state and event make implementation of the FLMM's concepts intuitive in OCAS. Table 3 below shows the mapping between Failure Logic concepts and OCAS constructs.

Table 3 - Mapping Between FLM Concepts and Appropriate AltaRica/OCAS Constructs

FLMM Concept	OCAS construct	Additional constraints and conventions
Basic Component	Component	Distinction between "Component", "Equipment" and "System" is specific to OCAS (other AltaRica dialects consider all three as a single construct – "node")
Complex Component	System (for complex components which are not contained in any other complex component) Equipment (for all other cases)	
Input Failure Mode	a) Boolean Input Flow or b) A literal in an enumerated input flow	Two possibilities are open in OCAS/AltaRica: • Each FM can be modelled as a Boolean flow (more

Output Failure Mode	a) Boolean Output Flow, or b) A literal in an enumerated output flow	faithful implementation) • Each FM can be considered as a literal (enumeration symbol) of an enumerated type; the flow assigned to such an enumeration type models an FM Group. This is a more 'efficient' implementation, but it assumes mutually exclusive FMs in a group.
FM Flow	An assertion on the equipment or system level	Can be graphically represented in OCAS environment or, alternatively, 'hidden' within textual representation.
FM Class	Explicitly predefined and documented Boolean type, or A documented literal in a predefined enumerated type	Dependent on the chosen representation of Input and Output Failure Modes (see above). OCAS provides facilities for documenting (in a free-text format) the nature of the FM Class that is represented by the predefined type
FM Group	A predefined enumerated or structured (record) type	If FMs are represented as Boolean flows (see above) FM groups can be represented as records. However, no further grouping is permissible in OCAS (yielding a constraint on the FLMM) If FMs are represented as literals, the overall input and output flow model an FM group (under a constraint that all FMs in the group must have the same orientation). Such FM groups can be further grouped by predefined record types (with no constraint on orientation); however no further (yet higher level) grouping is possible.
Propagation Condition	Assertion for the corresponding output flow (in *assert* clause of the component)	
State Space	A single state variable assigned either a Boolean or an enumerated type	Distinction between Failure, Normal and Failure Handling nature of the state space is made by naming convention only.
State	Enumeration symbol in a variable type (if enumerated) *true* value for Boolean variables	A privative state must be modelled either as *false* value for Boolean variables or as an explicit enumeration symbol (typically "*OK*") for enumerated variables
Transition	A single statement in *trans* clause of component characterisation	Syntactically transition specification distinguishes between the guard and trigger (using "**|-**" separator)

Void Trigger	Event	Must be assigned *Dirac(0)* law
Failure	Event	May be assigned a probabilistic law (except *Dirac*)
Normal Event	Event	Identified as non-failure through naming convention. Can be associated with the *Dirac (x)* law (where x > 0).

To illustrate the mapping, Figure 32 shows a Failure Logic characterisation of the Hydraulic Accumulator of the Wheel Braking System in AltaRica. (The tabular pseudocode characterisation is reproduced from Figure 33 in Appendix B for convenience)

```
node WBS_AccumulatorBool
 flow
  In : WBS_AccumulatorFMs : in ;
  In^PressureOmission : bool : in ;
  In^TooLowPressure : bool : in ;
  In^Leak : bool : in ;
  Out : WBS_AccumulatorFMs : out ;
  Out^PressureOmission : bool : out ;
  Out^TooLowPressure : bool : out ;
  Out^Leak : bool : out ;
 state
  FailSt : {OK,Isolated,TotalLoss,Leaking,PartialLoss} ;
  NormSt : {Full,Empty} ;
 event
  ConnectorBlocked, PartialDecompression, CompleteDecompression, Rupture,
  NORMAL_UsedOnce, Void ;
 init
  FailSt := OK ;
  NormSt := Full ;
 trans
  // Failure state entry logic
  FailSt != Leaking and FailSt != Isolated |- CompleteDecompression -> FailSt := TotalLoss;
  FailSt != Leaking and FailSt != Isolated and NormSt = Full
   and In^Leak |- Void -> FailSt := TotalLoss;          //Note void trigger
  true |- ConnectorBlocked -> FailSt := Isolated;       //Note void (tautology) guard
  FailSt != Isolated |- Rupture -> FailSt := Leaking;
  FailSt != TotalLoss and FailSt != Leaking
   and NormSt != Empty |- PartialDecompression -> FailSt := PartialLoss;

  // Normal state entry logic
  NormSt = Full and FailSt != TotalLoss
   and FailSt != Leaking and FailSt != Isolated
   and In^PressureOmission |- NORMAL_UsedOnce -> NormSt := Empty;
 assert
  //Propagation conditions
  Out^Leak = (FailSt = Leaking);
  Out^PressureOmission = ((FailSt = Isolated) or (FailSt = TotalLoss) or (NormSt = Empty));
  Out^TooLowPressure = ((FailSt = PartialLoss) and (NormSt = Full));
                                  // Note no need to specify absence of other failure states
 extern
  law <event NORMAL_UsedOnce> = Dirac(1) ;
  law <event Void> = Dirac(0) ;
edon
```

Figure 32 - AltaRica Characterisation of the WBS Accumulator (FMs as Booleans)

Accumulator
[Blue Channel]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
PressureOmission	Omission	In
TooLowPressure	LowValue	In
Leak	Commission	In

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
ConnectorBlocked		UsedOnce	Normal event Assume probability = 1
PartialDecompression			
CompleteDecompression			
Rupture			

Normal States & Entry Logic

Normal State	Trigger	Guard
Empty	UsedOnce	Full & PressureOmission & ~Isolated & ~TotalLoss & ~Leaking
Full	[Initial state, no re-entry]	

Failure States & Entry Logic

Failure State	Trigger	Guard
Isolated	ConnectorBlocked	~TotalLoss
TotalLoss	<void>	Leak & ~Leaking & ~Isolated
	CompleteDecompression	~Leaking
Leaking	Rupture	~Isolated
PartialLoss	PartialDecompression	~Empty & ~TotalLoss & ~Leaking

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
PressureOmission	Omission	Out	Empty | Isolated | TotalLoss
TooLowPressure	LowValue	Out	PartialLoss & Full & ~Isolated & ~TotalLoss & ~Leaking
Leak	Commission	Out	Leaking

Figure 33 - Failure Logic Characterisation of the Accumulator (Pseudo-code)

The following observations should be made:

- As was indicated above AltaRica does not permit the omission of an event in the specification of transitions. To model a void trigger, an explicit event should be introduced (in this example it is called "*Void*") and assigned an 'instantaneous' *Dirac(0)* law.
- There is no facility in the language to distinguish between different types of states and events – the distinction is made merely by naming conventions (e.g. see Normal Event "*NORMAL_UsedOnce*").
- In the case of an accumulator a Normal event "UsedOnce" is expected to occur some time after the supply of pressure from Blue Pump has ceased. This 'temporal' nature of this event is highlighted by assigning it a *Dirac(1)* law.
- A special state – "OK" – is added to a *FailSt* state variable that represents the Accumulator's failure state. This value represents the absence of any defect.

Finally, it is important to note that the accumulator characterisation in Figure 32 essentially implements each Failure Mode as a Boolean flow. This is a literal implementation of the Failure Logic Metamodel. However, an alternative representation is possible and is often more efficient: failure modes can be represented as literals in an enumerated type (with a special case *OK*

enumeration symbol that represents absence of a failure mode). This representation, however, will prohibit simultaneous output failure modes within the same FM Group. Figure 34 shows an OCAS screen-shot with an alternative characterisation of the *Accumulator* component.

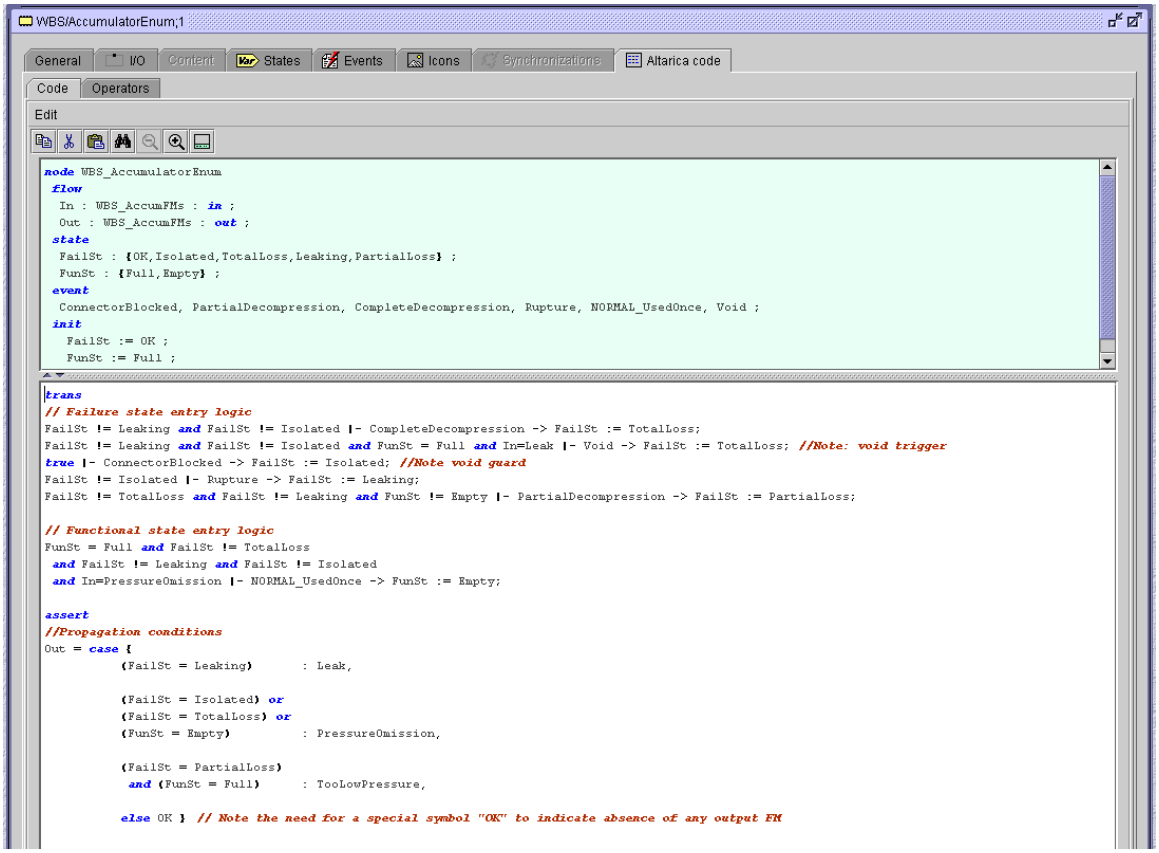


Figure 34 - OCAS Characterisation of Accumulator (FMs as Enumeration Literals)

3.6 Case Study: Wheel Braking System

To demonstrate that the Failure Logic Metamodel can be adequately instantiated in AltaRica language and the Cecilia OCAS tool, a pseudocode characterisation of the complete Wheel Braking System is translated into AltaRica equivalent. The complete AltaRica Dataflow model (automatically exported by the Cecilia OCAS tool) is presented in the Appendix B2. In most cases, the translation is self-evident with only few aspects requiring further discussion.

3.6.1 Predefined Types: Failure Modes and FM Classes

First, as was suggested in previous section the choice is made to represent failure modes as literals in enumerated flows. Furthermore, it can be observed that the output failure modes of most hydraulic components adhere to a single pattern. In particular, all hydraulic components can generate deviations of pressure. For each specific component, possible deviations are a subset of: *PressureOmission*, *PressureCommission*, *TooLowPressure*, *TooHighPressure* and *LowPressureCommiss*

(although not all components are capable of producing all failure modes – for example, no components on the Blue channel are capable of exhibiting either *PressureCommission* or *LowPressureCommiss*).

In addition to pressure deviations, each hydraulic component is capable of generating a *Leak* failure mode (which generally 'flows' in the opposite direction to pressure FMs).

This observation is utilised by declaring a general flow type in OCAS (Figure 35) and reusing it for most hydro-mechanical components. The *HydraulicFMs* type is a record consisting of two enumerated fields:

- The *FWD* field captures pressure failure modes
- The *BWD* field captures a *Leak* failure mode[29]

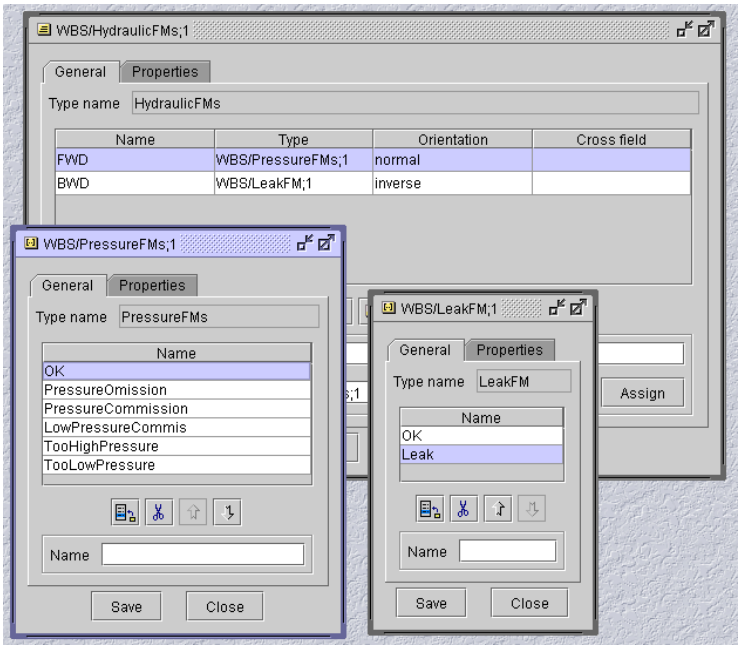


Figure 35 - OCAS: HydraulicFMs Type Along with Field Types (PressureFMs & LeakFM)

The two fields are each given a different "orientation" in OCAS. This means that whenever an input, say *HydIn*, of the *HydraulicFMs* type is declared in a component, an input flow of *HydIn^FWD* will be declared along with an output flow of *HydIn^BWD*.

On a component level, inapplicable input failure modes (with no viable interpretation) are simply ignored and the characterisation ensures that these output failure modes are never generated.

[29] The BWD field could be equally modelled as a Boolean. The reason for modelling it as a two-valued enumerated type is merely to maintain uniformity in the component characterisation and thus improve readability of the code.

However, the *HydraulicFMs* type is not adequate for characterising the interface between the Pumps and the *Isolation Valve*. In addition to generating pressure failure modes and to being susceptible to the *Leak* (input) FM, pumps can themselves leak. This is particularly important for the *Blue Pump* whose leak can propagate to the *Accumulator* causing it to empty prematurely. Consequently, the *Isolation Valve* can propagate leaks in both directions.

To capture this additional failure mode but to maintain as uniform a characterisation of components as possible a separate flow type is introduced – *PumpFMs*. It has the same record structure as the *HydraulicFMs* type; however, the *FWD* field is given a new enumeration type – *PumpPresFMs* – that includes a 'forward leak' described above (Figure 36).

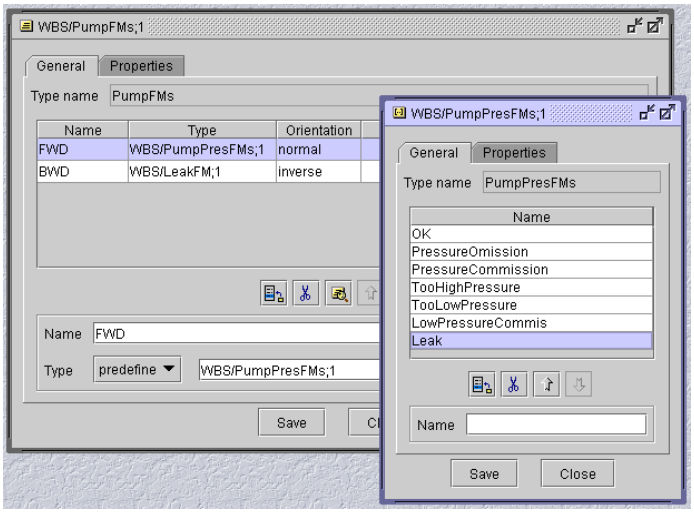


Figure 36 - OCAS: PumpFMs Type

Similarly, general flow types are introduced for all other groups of failure modes:

- *the ControlFMs* type encapsulates all of the failure modes which are associated with the BSCU *CMD* output, the two *Pedal* inputs and the output and input of the *Mechanical Pedal Position;*
- *the AntiskidFMs* captures failure modes associated with the *AS* output of the BSCU;
- *the ValidityFMs* model BSCU *validity output* as well as the outputs of individual monitors;
- *the PowerFMs* – describe the BSCU's electrical *power inputs*.

In addition to providing a uniform structure to the AltaRica model, definition of these general types ensures the consistency and correctness of the model (since OCAS does not permit the establishment of connections between dissimilar flows).

3.6.2 Model Architecture and Examples of Components

Figure 37 shows an AltaRica characterisation of the isolation valve (which is fully consistent with the pseudocode characterisation presented in Appendix B1). Note that, since Valves generally have two orthogonal failure state spaces – related to leakage and controllability – two state variables are introduced (*FailSt1* and *FailSt2*) in each component characterisation.

Figure 38 shows the top-level architecture of the WBS failure logic model in Cecilia OCAS. The tool allows the user to associate components with a set of graphical icons, which facilitates intuitive representation of the architecture and the representation of animation during simulation.

```
node WBS_IsolationValve
 flow
   In : WBS_PumpFMs : in ;
     In^FWD : WBS_PumpPresFMs : in ;
     In^BWD : WBS_LeakFM : out ;
     Out : WBS_PumpFMs : out ;
     Out^FWD : WBS_PumpPresFMs : out ;
     Out^BWD : WBS_LeakFM : in ;
 state
   FailSt1 : {OK,StuckClosed,StuckOpen} ;
   FailSt2 : {OK,Leaking} ;
 event
   Contamination, Jam, Rupture ;
 init
   FailSt1 := OK ;
   FailSt2 := OK ;
trans
   // Failure state entry logic
   true |- Contamination -> FailSt1 := StuckClosed;
   FailSt1 != StuckClosed and (In^FWD = PressureOmission or In^FWD = Leak) |- Jam -> FailSt1 := StuckOpen;
   true |- Rupture -> FailSt2 := Leaking;
assert
   // Propagation conditions
      // Output FMs to downstream section of WBS:
   Out^FWD = case {
      FailSt2 = Leaking or
      In^FWD = Leak and FailSt1 = StuckOpen                    : Leak,

      FailSt2 != Leaking
       and (FailSt1 = StuckClosed or In^FWD = PressureOmission or
            (In^FWD = Leak and FailSt1 != StuckOpen))          : PressureOmission,

      In^FWD = TooLowPressure and FailSt1 != StuckClosed
       and FailSt2 != Leaking                                  : TooLowPressure,

      else OK};

     // Output FM to upstream (pump) section:
   In^BWD = case {
      FailSt2 = Leaking or
      Out^BWD = Leak and FailSt1 != StuckClosed                : Leak,

      else OK};
edon
```

Figure 37 - AltaRica: Failure Logic Characterisation of the Isolation Valve

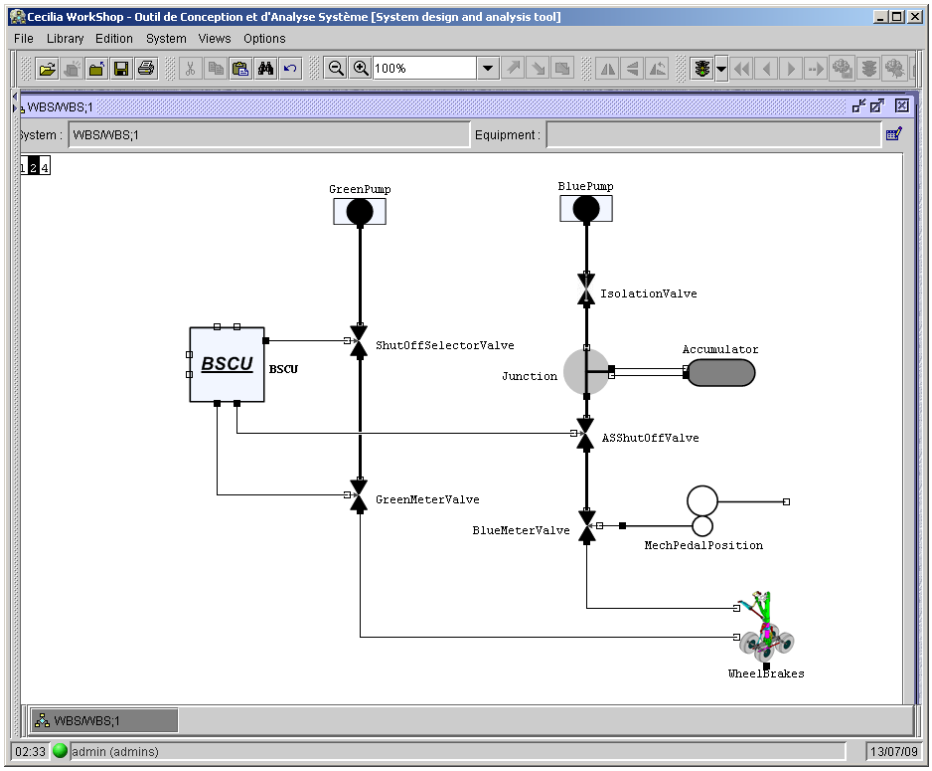


Figure 38 - OCAS: Top-Level Failure Logic Model Architecture of WBS

3.6.3 Virtual Components: Model-Level Input FMs and Observer

The Wheel Braking System has a number of system-level inputs. However, OCAS does not permit the analysis or simulation of models with free input flows. To simulate the WBS model in OCAS, it is therefore necessary to define a number of "virtual components" which emulate system inputs. These observers have a simple structure – they all have one output of the appropriate type (e.g. *PowerFMs*) and a state variable – *FailSt* – of the same type. For each literal in the type except the *OK* privative, a 'pseudo-failure' is declared. Each such event moves the component into a corresponding state; the state is propagated to a component output. Figure 39 shows the characterisation of such a "virtual component" for power inputs. Overall, the WBS model contains seven input 'stubs': three pedal inputs, two power inputs as well as leak inputs for the green and the blue meter valves (the latter two emulate leaks in the wheel assemblies, since the internal structure of the assembly is outside the scope of the model) – all of these inputs are shown in Figure 40.

Another type of "virtual component" that is often found in AltaRica Models is the *Observer*. An observer may be necessary if the system level failure condition(s) of interest involve more than one output flow. This is clearly the case in the WBS – since braking is determined by the combination of the Green and Blue pressure outputs. The *WheelBrakes* component has the role of observer in the WBS failure logic model.

```
node WBS_PowerStub
 flow
   Out : WBS_PowerFMs : out ;
 state
   FailSt : WBS_PowerFMs ;
 event
   INPUT_PowerOmission ;
 init
   FailSt := OK ;
 trans
   FailSt = OK |- INPUT_PowerOmission -> FailSt := PowerOmiss;
 assert
   Out = FailSt;
edon
```

Figure 39 - Characterisation of the 'Virtual' Input Component for *PowerFMs* Type Input

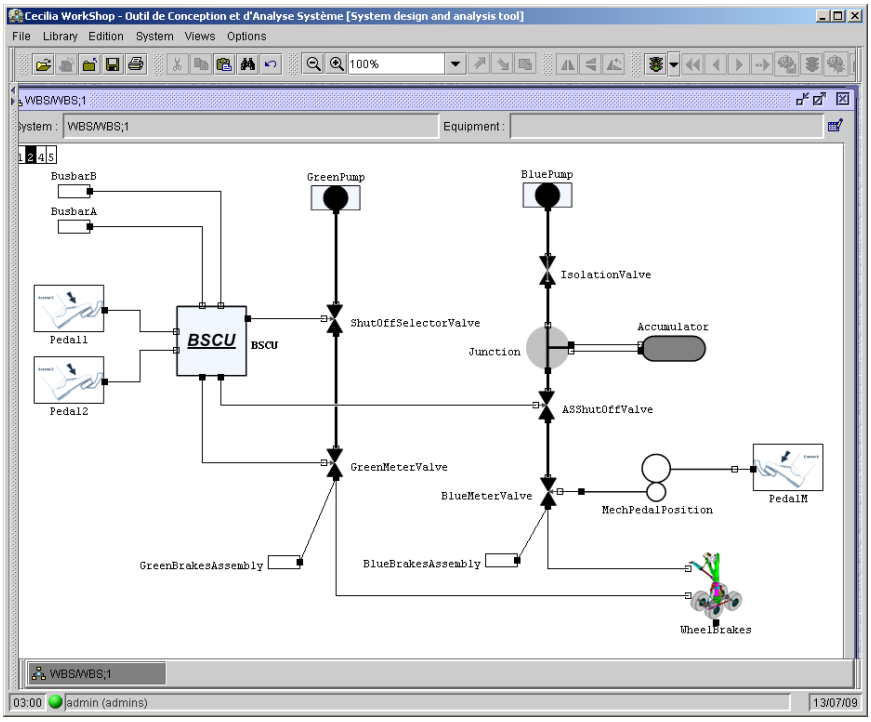


Figure 40 - OCAS: WBS Model Architecture with 'Virtual' Input Components

3.6.4 Model Simulation and Analysis

Once all of the inputs are bound and the observer(s) is defined, it is possible to simulate the model to perform inductive analysis interactively (similar to a, potentially multi-failure, FMEA/FMECA). Figure 41 shows an OCAS simulation of the WBS model where the user has triggered two failures in the *BSCU* as well as a failure of the Accumulator's *connector* and a blockage of the *Isolation Valve*. The result is an omission of the wheel braking shown graphically by the simulator's GUI (using user-defined icons and colours schema).

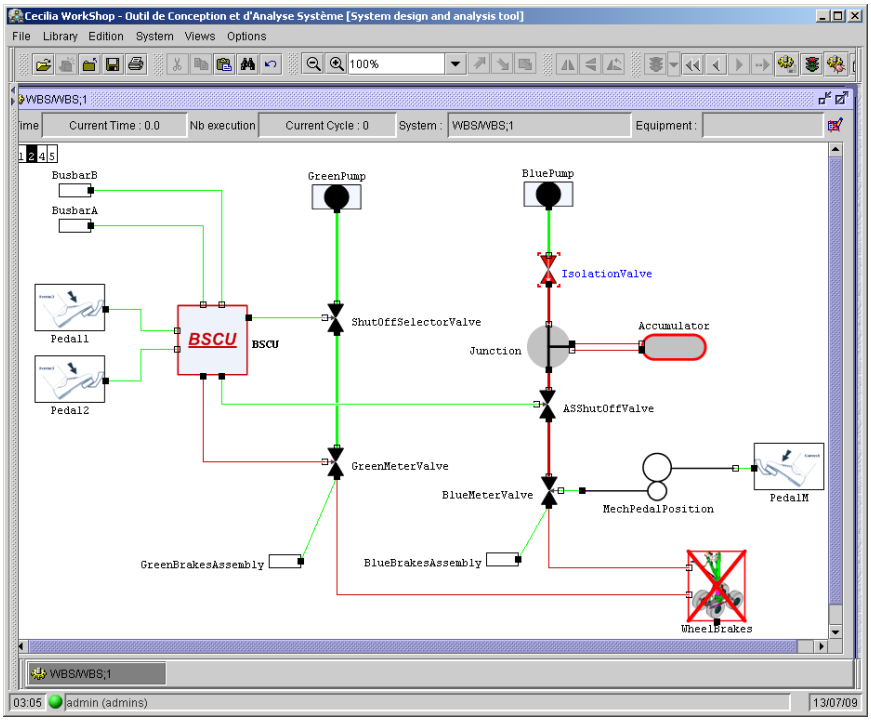


Figure 41 - OCAS: Model Simulation

In addition to simulation, sequence generation can be used to search through the permutations of events (up to a particular size) exhaustively and to identify all causes (similar to FTA's Minimal Cut Sets) that lead to a particular system-level condition. Table 4 shows the list of MCSes up to size (cardinality) of three failures for Inadvertent Braking (a potentially catastrophic failure condition in WBS) generated by the OCAS sequence generator.

Table 4 - Minimal Cut Sets for Inadvertent Braking

Sequence / MCS generation results	FC: **Inadvertent Braking**
Single points of failure:	
{BlueMeterValve.JamOpen}	
{BlueMeterValve.SpringFailure}	
{GreenMeterValve.JamOpen}	
{GreenMeterValve.SpringFailure}	
{MechPedalPosition.Jam}	
{PedalM.INPUT_InadvertentBraking}	
Double-failures:	
{BSCU.BSCU1.COM.CMDprocessStuck, BSCU.BSCU1.MON.ProcessStuck}	
{BSCU.BSCU1.MON.ProcessStuck, Pedal1.INPUT_InadvertentBraking}	
{BSCU.BSCU1.MON.ProcessStuck, Pedal2.INPUT_InadvertentBraking}	
{BSCU.BSCU2.MON.ProcessStuck, Pedal1.INPUT_InadvertentBraking}	
{BSCU.BSCU2.MON.ProcessStuck, Pedal2.INPUT_InadvertentBraking}	
{BSCU.ValidityMonitor.ProcessStuck, Pedal1.INPUT_InadvertentBraking}	
{BSCU.ValidityMonitor.ProcessStuck, Pedal2.INPUT_InadvertentBraking}	
{Pedal1.INPUT_InadvertentBraking, Pedal2.INPUT_InadvertentBraking}	
{Pedal1.INPUT_InadvertentBraking, ShutOffSelectorValve.SpringFailure}	
{Pedal2.INPUT_InadvertentBraking, ShutOffSelectorValve.SpringFailure}	
Triple Failures:	
{BSCU.BSCU1.COM.ASprocessStuck, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck}	
{BSCU.BSCU1.COM.ASprocessStuck, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck}	
{BSCU.BSCU1.COM.ASprocessStuck, BSCU.BSCU2.COM.CMDprocessStuck, ShutOffSelectorValve.SpringFailure}	
{BSCU.BSCU1.COM.ASprocessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck}	
{BSCU.BSCU1.COM.ASprocessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck}	

{BSCU.BSCU1.COM.ASprocessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU1.COM.CMDprocessStuck, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck}
{BSCU.BSCU1.COM.CMDprocessStuck, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck}
{BSCU.BSCU1.COM.CMDprocessStuck, BSCU.BSCU2.COM.CMDprocessStuck, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU1.COM.CMDprocessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck}
{BSCU.BSCU1.COM.CMDprocessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck}
{BSCU.BSCU1.COM.CMDprocessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU1.MON.ProcessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck}
{BSCU.BSCU1.MON.ProcessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck}
{BSCU.BSCU1.MON.ProcessTerminated, BSCU.BSCU2.COM.CMDprocessStuck, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU1.MON.ProcessorError, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck}
{BSCU.BSCU1.MON.ProcessorError, BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck}
{BSCU.BSCU1.MON.ProcessorError, BSCU.BSCU2.COM.CMDprocessStuck, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck, BusbarA.INPUT_PowerOmission}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck, Pedal1.INPUT_LackOfBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck, Pedal1.INPUT_TooLittleBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck, Pedal2.INPUT_LackOfBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.BSCU2.MON.ProcessStuck, Pedal2.INPUT_TooLittleBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck, BusbarA.INPUT_PowerOmission}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck, Pedal1.INPUT_LackOfBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck, Pedal1.INPUT_TooLittleBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck, Pedal2.INPUT_LackOfBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BSCU.ValidityMonitor.ProcessStuck, Pedal2.INPUT_TooLittleBraking}
{BSCU.BSCU2.COM.CMDprocessStuck, BusbarA.INPUT_PowerOmission, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU2.COM.CMDprocessStuck, Pedal1.INPUT_LackOfBraking, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU2.COM.CMDprocessStuck, Pedal1.INPUT_TooLittleBraking, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU2.COM.CMDprocessStuck, Pedal2.INPUT_LackOfBraking, ShutOffSelectorValve.SpringFailure}
{BSCU.BSCU2.COM.CMDprocessStuck, Pedal2.INPUT_TooLittleBraking, ShutOffSelectorValve.SpringFailure}

3.7 Role of the FLMM in the Safety Case

By definition, results of the system safety assessment provide evidence for a system's safety case (or, otherwise, indicate that the system is not adequately safe). In the context of model-based safety assessment, the integrity of these results clearly depends on the 'validity' of the model. Consequently, justification of the model's adequacy must form a crucial part of the overall safety case of the system [63].

However, as was stressed in Chapter 2, any 'safety assessment model' is in essence a *hypothesis* posed by the safety engineers, and as such can never be fully and formally validated. In this setting, a structured and clear framework for deriving the model becomes indispensable.

The Failure Logic Metamodel presented in this thesis (along with the guidance developed by the author[30]) provides the basis for constructing an argument of model adequacy which should be incorporated into the overall system safety case. Furthermore, the separation between the FLMM and the language-specific instantiation facilitates a structured approach to safety case construction capable of improving management of the safety case by identifying reusable arguments and by defining the scope of valid reuse. In particular, the overall secondary safety case of a system may be separated into a number of largely independent arguments:

[30] See section 6.4.2 (Chapter 6)

I. Argument of the adequacy of the failure logic modelling approach as a whole for the purpose of analysis of a particular system;
II. Argument of the acceptability of the constraints introduced by the instantiation of the approach in a particular specification language;
III. Model-specific argument of the adequacy of the modelling assumptions introduced by safety engineers.

In practice, considering the first argument will yield a standard set of criteria for application of the failure logic modelling approach (irrespective of the details of a particular technique). One such criterion is the ability to represent system design adequately as a set of discrete components and interactions over well-defined paths. Provided that these criteria are met, the first argument will be standard for all systems and for all engineering organisations.

The second argument will yield further language-specific conditions and will generally be stable and reusable *within* an engineering organisation across similar systems. For example, to argue the adequacy of the schema for the FLMM instantiation in AltaRica OCAS, it must be demonstrated that the lack of facilities for representing non-deterministic behaviour in the failure domain (e.g. alternative outcomes of failures) does not adversely affect the accuracy of the WBS model. Further aspects that may require justification are the lack of ability to construct deep hierarchies of FM Groups and the strict constraint of mutual exclusivity of Failure Modes modelled as literals within a single enumerated type.

The last argument is clearly specific for each safety critical system.

3.8 Limitations of the ‘Baseline’ Approach

Preceding sections of this chapter have unified most of the existing failure logic modelling approaches and introduced natural extensions to previously exclusively combinatorial and *FM*-focused techniques. Whilst the extensions increase the expressive power of the techniques such as FPTN and HiP-HOPS, the notation-agnostic definition of the failure logic metamodel makes it easier to identify some, previously unreported, problems shared by the existing techniques. We use the simple tank system example – introduced in Chapter 1 – to illustrate.

Introduction of the notions of *state* and *normal event* permits more accurate representation of the failure logic of the system. Firstly, it allows certain sequences of failures to be disregarded – such as sensor’s calibration failing after the sensor has developed an open- or short- circuit (and was, thus, disregarded by the controller). Secondly, and more importantly, the failure logic model can capture the fact that, following the *omission* of fluid inflow into the tank, the hazardous condition

(inability of the tank to supply fluid on demand) will only occur *after* the fluid stored in the tank is used-up by the consumers. Whilst the latter is a normal event, its recognition not only improves the superficial accuracy of the analysis, but is also capable of providing valuable feedback to the design of the process plant as a whole. In particular, given that the intent of the system is to *raise the alarm after any safety-significant failure*, this signal can be provided to systems that consume the fluid to avoid non-essential demands for the limited resource and, if necessary, to initiate procedures for controlled plant shut-down.

This overall intent of the alarm to forewarn any hazard, however, also highlights the limitation of the failure logic modelling framework as presented in this thesis so far. It is reasonable to assume that annunciation of the impeding hazard (whether an overflow or emptying of the tank) affects its potential severity. "*Unannunciated Emptying of the Tank*" is therefore a reasonable refinement of the original hazard and can be specified as a conjunction of the *Too Low* failure mode of the tank level and the *Omission* failure mode of the alarm. However, analysis of the failure logic model of the tank system (constructed along the lines of Chapter 1, but adapted to the metamodel presented in this chapter) would *not* identify *Stuck Closed* failure of the valve as a sufficient cause for this condition.

At the level of failure logic model specification the problem is that there is no physical dependency path from the Valve to the Alarm. Whilst a failure mode flow between the two components can be defined, in the absence of such a path, its semantics is not clear.

Similar problems exist with the *Rupture* of the tank and *Omission* of the fluid supply to the system. Problems, of *non-local dependencies between components in the failure logic domain*, become particularly severe in reconfigurable and multi-modal systems (and are addressed in Chapter 5 of the Thesis).

Further limitation of the failure logic modelling methods comes from a restricted notion of component interface embedded in the metamodel. In particular both states and events (normal and failure) of components are currently hidden within boundaries of component characterisations. Consequently all dependencies between components must be specified as explicit failure modes and failure mode flows. In the tank system this may pose a pragmatic problem for the controller component. Chapter 1 has stipulated that this component is implemented in software and details of the hardware platform were considered to be outside the scope of the assessment. Considering the implementation platform may highlight dependencies of the controller on new external factors (for instance on the provision of the electrical power). Furthermore, it may also identify new dependencies between existing components; for instance, if sensors are connected to the same input/output card, a short circuit from one sensor may cause damage to the card, thus, rendering the other sensor dysfunctional (due to the inability of the controller hardware to provide an

excitation signal). Similarly, the controller hardware may be sensitive to the short circuits of the alarm and/or the valve. Finally, considering the zonal allocation of the equipment may reveal yet more ‘covert’ dependencies and, thus, components’ input and output failure modes.

Overall, the ‘conceptual’ failure logic model of the system presented in Chapter 1 cannot be presently composed with the models of plant zones and hardware platform. Instead, to conduct a more complete safety assessment, safety engineers either need to remodel the entire system or to have foreseen all external dependencies between all such different models in the first place (and coordinated the failure mode interfaces between different models throughout their specification). Neither of the approaches is practicable in the context of large-scale safety critical systems engineering. This highlights issues with the claims of composability and compositionality of the current failure logic modelling techniques which are discussed in the following chapter.

3.9 Conclusions

This chapter has demonstrated that existing failure logic modelling techniques can be generalised by a single unifying Failure Logic Metamodel (FLMM) that defines key concepts of the approach and outlines their inter-relationships. The metamodel has been further extended to cover dynamic and normal behaviour in the failure domain (necessary for modelling realistic systems). The extension also enables the relationship of failure logic modelling techniques to some, state-focused, languages that have been used in the context of model-based safety assessment (such as SEFT [80], AADL Error Model Annex [136] and, of course, AltaRica [22]).

The chapter has further demonstrated that the language-independent FLMM can be instantiated in the context of a third-party general specification language. However, it was argued that the metamodel-based approach is capable of separating conceptual- and implementation- level concerns which may be beneficial for the construction and maintenance of the safety cases within an engineering organisation.

Similarly, from the perspective of this thesis, the definition of the FLMM permits one to abstract from the details of individual techniques and investigate fundamental properties of the failure logic modelling approach as a whole. To this extent, in addition to the contributions listed above, this chapter has defined a ‘baseline approach’ that is further investigated and extended in the following chapters. In particular, it has often been claimed that individual failure logic modelling techniques are more effective than traditional analysis approaches and yield compositional and, often, reusable models. Chapters 4 and 5 of this thesis investigate these claims further.

Chapter 4: Composition of Multiple Models

4.1 Introduction

The previous chapter presented the generalised Failure Logic Metamodel (FLMM), along with the claims made for individual failure logic modelling techniques (e.g. see [162, 163, 165]) concerning the compositionality and reusability of the modelling artefacts. This chapter investigates the validity of such claims in more detail in the context of large-scale engineering projects.

Claims of the compositionality of failure logic models are typically made in the context of a single system development. Furthermore, it is often assumed that the system development is a fully-integrated process and that the safety engineers have complete visibility of all of the development activities and unrestricted access to all of the design materials and engineers involved in the development. In such a context, it may be reasonable to assume that the safety engineers are capable of comprehending the system as a whole at some level of abstraction sufficient for the identification and definition of all failure mode interfaces between individual components. Consequently, under such assumptions the failure logic models appear to be compositional. Examples cited are typically relatively small-scale stand-alone systems or individual systems of larger engineered artefacts such as aircraft, automobile, maritime vessel or a process plant. In this chapter, the issue of the compositionality of failure logic models is addressed in the *context of such large-scale artefacts*. To emphasise the qualitative difference in scale this thesis adopts Pumfrey's term of "*platform*", defined as "*the largest engineered artefact (e.g. a complete aircraft, train or chemical process plant)*" [123].

Unlike the development of individual systems, the design of platforms is too complex an engineering activity to be carried out in a highly-integrated manner. Consequently, the overall design is typically broken down into manageable parts. These parts are engineered in relative isolation: their design utilises different engineering disciplines, methods and tools and the process is often carried out by separate engineering organisations. It is also important to stress that a platform is often decomposed in several orthogonal ways. On the one hand, the decomposition can follow 'structural' criteria, such as the decomposition of an aircraft into individual systems or decomposition of an airframe into major zones, whereby the different elements of a design are 'sorted' into a number of – typically non-overlapping – groups. On the other hand, the decomposition criteria can be based on relevant engineering concerns, disciplines and classes of interactions being considered (e.g. the propagation and effects of leaks being considered separately from issues of pressure provision by a hydraulic power distribution system). In practice, both of these broad types of decomposition are combined to form complex hierarchies of

the platform. To address issues of the construction and composition of failure logic models in the context of platforms this chapter first rationalises platform decomposition using a flexible concept of “engineering domain” – a generalisation of the “view” concept established in the field of Software Architectures [33, 67, 70].

Clearly, in order for the failure logic modelling approach (or indeed any safety assessment method) to be practicable in the context of safety critical platforms development, the technique must be shown to be compositional with respect to the organisation of the engineering process and the decomposition of the platform. In this context, many of the assumptions on which claims for the compositionality of current failure logic modelling methods are based no longer hold. In particular, it is unrealistic to expect that all dependencies between Domain-Specific Failure Logic Models (DSFMs) will be systematically identified in a top-down fashion before individual models are constructed; therefore the naïve approach to failure logic model composition, based on the connection of failure mode interfaces, is unlikely to be sufficient. This chapter systematically examines how failure behaviour captured in one DSFM can affect and manifest in other models. An extended approach to the composition of models is then presented and illustrated by means of a case study concerning a common computational platform and its relationships with the Wheel Braking System failure logic model constructed in the previous chapter.

4.1.1 Illustration of the Problem Addressed by the Chapter

To illustrate a problem of system failure logic model composition in the context of large scale safety-critical platforms this section returns to the WBS case study introduced in the previous chapter.

For the hydro-mechanical part of the system, the previously presented failure logic model of the WBS has only considered failure modes due to intentional dependency paths between components established by the hydraulic pipework. However, the spatial organisation of the components within the airframe may also inadvertently result in interactions between the WBS components as well as their interactions with components of other systems. For example, if components of the blue and green hydraulic lines are installed in physical proximity, a leak of the flammable hydraulic fluid in one line combined with excessive heat (or sparks) generated by any valve (of the same line) running dry may result in a fire that can destroy equipment of seemingly redundant hydraulic lines. Further interactions between the WBS equipment in terms of phenomena such as vibration, electro-magnetic interference (EMI) and even temperature of the hydraulic fluid may affect safe operation of the system. Furthermore, the failure state of the equipment can be affected by components outside the WBS: for example valves or pipework may be destroyed by fragments generated by aircraft tire burst or engine disc failure.

The baseline failure logic metamodel presented in the previous chapter stipulates that all such unintentional interactions should be recognised (and modelled) as input and output failure modes of the hydro-mechanical components. Not only is such model unlikely to be maintainable, but input and output failure modes of the WBS components would not be fully identified before the aircraft installation model is finalised by the design engineers and extensive zonal and particular risk analysis is undertaken by the safety engineers. Therefore, as design progresses the specification of the failure logic model – and, thus, the safety assessment – of the WBS would either have to be postponed (reducing the opportunity for effectively influencing the design before it matures) or would have to undergo extensive modifications.

Furthermore, this 'supermodel' approach undermines a key principle of modern system development and safety assessment approaches – separation of concerns between different engineering organisations.

Similar problems can be found when WBS interactions with other aircraft systems, especially those providing aircraft-wide infrastructure, are considered. In addressing issues of the compositional failure logic modelling in the context of safety-critical platforms, this chapter uses the case study concerned with the composition of independently defined models of the WBS and Aircraft Computation and Communications Infrastructure. Consistent with the notion of Integrated Modular Avionics (IMA), the latter is typically considered to be an aircraft system in its own right. The IMA and WBS can be designed by different suppliers and should be initially analysed in isolation.

However, as it was previously stipulated that the WBS controller (i.e. the BSCU) is predominantly implemented in software, the two systems clearly interact. Consequently, failures of the IMA can affect the correct and timely computation of braking commands, whilst – depending on the details of scheduling and communication protocol – failure modes of the BSCU software may affect the computation and communications infrastructure and, potentially, propagate to the application software of other aircraft systems (such as landing gear extraction/retraction or fuel management systems). At the early stages of the design it is unrealistic to expect safety engineers responsible for the wheel braking system to fully identify feasible failure modes of the infrastructure that can affect the BSCU or, conversely, failure modes of the BSCU that are relevant to the infrastructure (since these engineers will not necessarily have sufficient knowledge of the infrastructure design and, thus, failure logic). This chapter extends the failure logic modelling methodology to allow multiple failure logic models – the DSFMs – to be specified in relative isolation and composed at the later stages of aircraft development.

4.1.2 Objectives for the Composition of Failure Logic Models

In providing a practical approach to DSFM construction and composition the following objectives were established:

I. **Support any organisation of the development process:** the boundaries of DSFMs will not necessarily coincide with intuitive boundaries for the safety assessment process. In such cases it should be possible for DSFMs' boundaries to be dictated by the development rather than by safety assessment considerations.

II. **Do not undermine the allocation of responsibilities and accountability (and, even, the liability) of different organisations:** the composition of failure logic models must be as non-intrusive as possible. Any additional modelling necessary for the integration of DSFMs should be isolated from the constituent models themselves.

III. **Facilitate the proactive management of project risks associated with safety-significant findings:** the composition process should allow for the composition of models at various levels of maturity throughout the development process. In particular it should be possible to compose immature and 'lightweight' DSFMs which do not fully or systematically identify interfaces with other domains.

IV. **Reduce the impact of imperfect interfaces and the evolution of domain dependencies:** it should be possible to compose DSFMs even in the absence of well-defined FM interfaces and, as far as practicable, without requiring redefinition of individual constituent failure logic models.

These four objectives have shaped the approaches to the rationalisation of platform decomposition and the integration of failure logic models presented in this chapter.

4.2 Views and Domains of Safety-Critical Platforms

In this section, the decomposition of the platform (and platform engineering process) is considered more closely and rationalised by a flexible conceptual framework. The traditional decomposition model, also known as a containment hierarchy (whereby a platform is decomposed into systems and structures), is combined with the notion of viewpoints (established in the software architecture community) to yield a simple but flexible concept of the "engineering domain".

4.2.1 Views and Viewtypes

Having emerged as a consolidated discipline in the late 1990s, the study of Software Architectures [18, 67] has taken an approach to managing the complexity of software systems and the separation of concerns that is conceptually different from the traditional containment hierarchy.

Often attributed to a 1974 paper by David Parnas [118], the view of the software architecture community is that software cannot adequately be represented in a "simple one-dimensional fashion" [33] and as a single hierarchy. Instead the architecture should be rendered through a number of different loosely coupled views, each addressing particular *aspects* of the software and particular *concerns of the stakeholders*.

The IEEE's Recommended Practice for Architectural Description [70] describes how software architectures can be documented through a number of views organised by *viewpoint* (defined as: "*a pattern or template from which to construct individual views. The viewpoint establishes the purposes and audience for a view and techniques or methods employed in constructing a view*"). The standard suggests that viewpoints are determined according to the concerns of relevant stakeholders. This is a useful observation for the purpose of this thesis. Indeed, different groups of engineers with different responsibilities, expertise and modelling and analysis toolsets can be seen as different stakeholders for the platform (or its parts). The IEEE's conceptual model allows for the 'filtering' of aspects of the platform design that are relevant for a particular group of engineers.

The general notion of the viewpoint is specialised further by the SEI "Views & Beyond" approach. This approach identifies three *viewtypes* (general families of views): *Component-and-Connector (C&C)*, *Module* and *Allocation*. Each viewtype "defines the element types and relationship types used to describe the architecture of a software system from a particular perspective" [33]. The first two viewtypes document units of execution (and, thus, behaviour) and principal units of implementation (and, thus, the organisation of the development process) respectively. The allocation viewtype acts as 'glue' and documents various relationships between the software (as represented in both C&C and Module views) and its environment (both in terms of development and execution).

The concept of the viewtype is extremely general and does not in itself facilitate definition of the views (i.e. representations of the particular architecture). To bridge the gap in levels of abstraction, SEI researchers propose a notion of "*style*" defined as the "specialization of element and relation types, together with a set of constraints on how they can be used"[31] [33]. Individual views are an instantiation of a style and a style can be seen as a refinement of one (or more) of the three viewtypes (Figure 42). Styles are described through "style guides" which capture both the conceptual metamodel and, if applicable, the syntax of the views that adhere to the style.

[31] It is important to note that this definition of style (and its description in [32]) amalgamates two potentially dissimilar concepts: the style of the architecture and the type of an architecture description (or view). In this thesis the author focuses exclusively on the later interpretation.

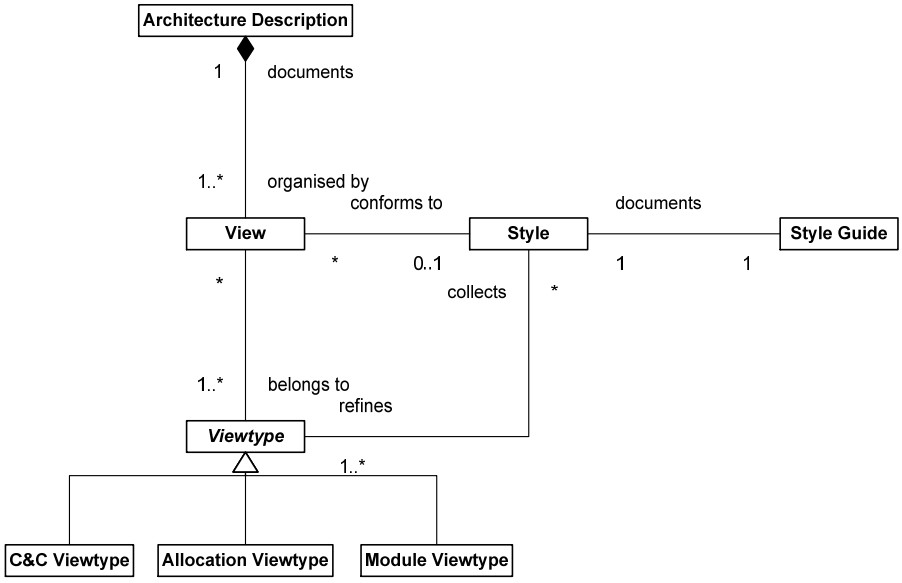


Figure 42 - SEI "Views and Beyond" Framework

The following section shows how the SEI notion of viewtype, when combined with a traditional systems engineering containment hierarchy, can be utilised to provide a flexible framework for the rationalisation of the decomposition of large-scale safety-critical platforms.

4.2.2 Domains and Domain-Specific Models

To support the composition of failure logic models, it is first necessary to model the different types of decomposition that can be found in the engineering of complex platforms. This thesis defines a “platform” as a collection of elements which participate in some interactions in order to deliver the necessary behaviour both in terms of functional and non-functional properties (as shown on the right hand side of Figure 43). It is also observed that the interactions between the components can be typically grouped into general “interaction types” and that some (more complex) elements can be decomposed into finer-grained elements. Note that the elements are not necessarily limited to equipment or components of the system – they can equally represent structural elements (e.g. zones or volumes).

As was stated above, the complexity and scale of the platforms mean that they cannot be engineered as a single artefact. Instead, platforms are decomposed into manageable parts which can be designed and analysed relatively independently. The concepts of views and viewpoints provide a useful facility to (partially) rationalise the decomposition of the platform into representations focussed on particular engineering concerns and aspects of the platform. In other words, the viewtype allows for the selection of a particular set of interaction types which are relevant to a particular set of engineers. However, the concept of the viewtype is not sufficient to capture the decomposition of the platform fully. Indeed, platform development is typically carried out by an extended enterprise – a loosely coupled group of companies and organisations often

organised into a hierarchical supply chain. In such a hierarchy, an individual company (especially one located towards the top of the hierarchy) is likely to have the scope of its responsibilities bound by some *physical* part of the platform, such as a particular system or a major part of the structure in the standard containment hierarchy model. Views and viewpoints do not define such a physical scope clearly (for example, the IEEE standard defines the view as "a representation of a *whole system*[32] from a perspective of related sets of concerns" [70]).

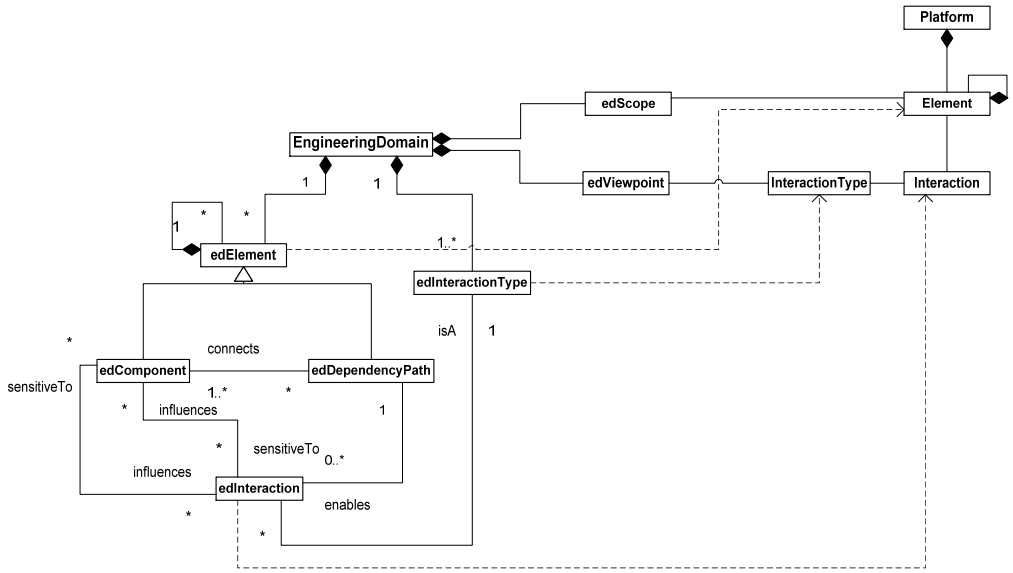


Figure 43 - Engineering Domain: Internal Structure and Traces to Platform

To reflect both types of platform decomposition it is necessary to introduce a new concept of the *Engineering Domain*. Characterised by *Scope* and *Viewpoint*, the engineering domain captures the view of the particular engineering organisation of the part of the platform over which it has responsibility. The scope of the engineering domain 'selects' the platform elements that are relevant to the domain and the viewpoint 'selects' the relevant interaction types (as shown in the upper section of Figure 43). Of course the scope and viewpoint are not fully independent entities and have to be consistent such that, for example, at least some elements within the scope can interact in the way selected by the viewpoint. However, details of this interdependence are beyond the scope of discussion in this thesis.

It is interesting to note that engineering domain is closely related to the SEI component & connector viewtype. Domain representations identify elements (from the domain's perspective), classify those into components and paths (functionally passive components which enable interactions) and show how components interact over paths (as shown in the bottom-left of Figure 43). Domain elements and interactions clearly relate to the elements and interactions of the

[32] Our emphasis. Note that, given the contexts of the standard and this thesis, "whole system" should be interpreted as "whole platform"

platform. However, this relation is relatively weak (shown as a “dependency” relationship), since a number of domains can represent the same (or overlapping) set of platform elements and the same (or overlapping) interaction types at different levels of abstraction.

4.2.3 FLMM in Engineering Domain Framework

Having presented the framework for rationalisation of the platform decomposition and design process, it is necessary to show how failure logic modelling relates to the framework. The relationship is, in fact, two-fold: failure logic models *are* themselves an engineering domain and they *relate to* other domains.

Firstly, the failure logic modelling approach is itself a viewpoint or, returning to the SEI terminology, an architectural style, and the previous chapter of this thesis can be seen as the style guide. Showing interaction between components in terms of failure modes, the ‘FLM style’ clearly falls within the C&C viewtype and – once supplemented with the scope in the context of platform – yields an engineering domain on its own. In principle, the failure logic modelling *approach*, itself, does not restrict the scope of this domain – it can be applied to the platform as a whole or to any subset of its elements. However, for failure logic modelling to be compositional and practicable in the context of an industrial development process, the scope of failure logic models must be set to reflect the scope of other engineering domains.

This constraint establishes the relationship between failure logic models and the domain framework and yields a notion of the Domain Specific Failure Logic Model (DSFM) – a failure logic model with a scope identical to that of some (other) engineering domain. Indeed, the structure of the DSFMs should relate closely to the structure of the engineering domain being modelled and analysed. To illustrate this relationship further, it is necessary to refine the conceptual model of the engineering domain to explicitly represent sensitivities and influences (through which components participate in interactions) as shown in Figure 44.

To conclude this section it is important to make two brief observations.

Firstly, the direction of the trace relationships between the FLMM’s and Engineering Domain concepts indicates that DSFMs are, to an extent, ‘secondary’ engineering domains – they typically reflect (part of the) the platform through the proxy of design descriptions contained in other (‘primary’) domains.

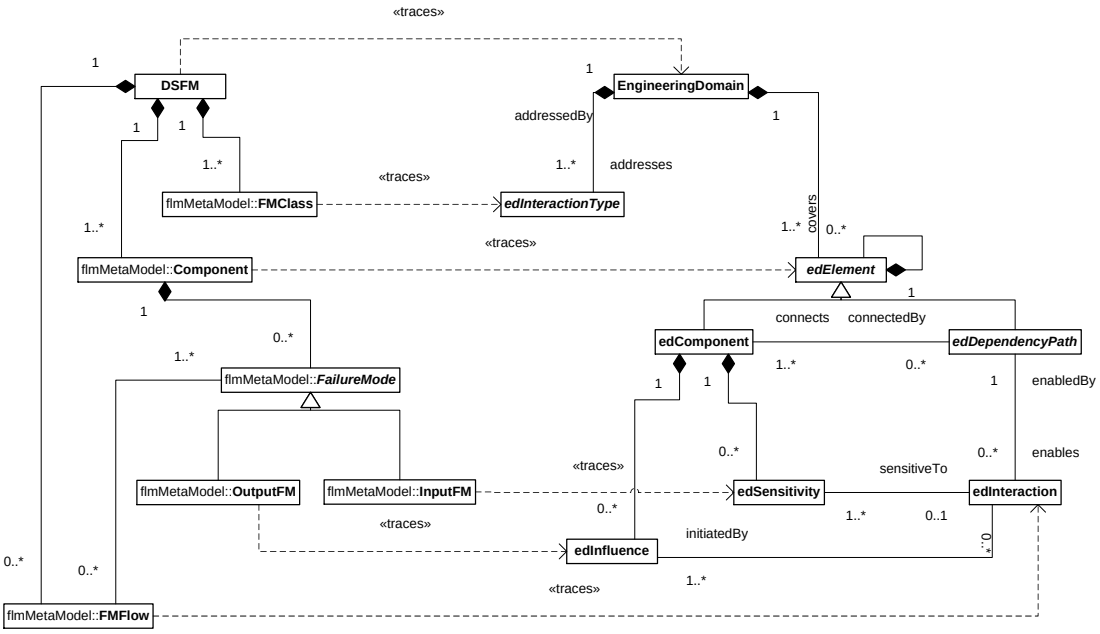


Figure 44 - DSFMs and Engineering Domains

Secondly, the FLM Component metaclass traces to the Engineering Domain's Element (rather than the component). This reiterates a need to model passive components under the failure logic modelling approaches, since these may have a failure logic of their own (e.g. they can fail). Furthermore, as is explained in the next section, elements which appear to be passive in one engineering domain may be functionally active (and, thus, have non-trivial failure behaviour) in another domain.

4.3 Allocation Viewtype and Composition of DSFMs

The previous section has presented a framework for rationalising the decomposition of safety-critical platforms and has demonstrated how the products of decomposition – engineering domains – can be reflected by domain-specific failure logic models. The decomposition allows different organisations and stakeholders to 'ring fence' their responsibilities for platform design and assessment and to discharge these in relative isolation from other stakeholders. The ability to define local DSFMs facilitates the early safety assessment of part of the platform and, thus, allows for the proactive control of project risks associated with the late identification of safety concerns.

This section turns to the interactions between different engineering domains and the resultant dependencies between corresponding DSFMs. The goal is to establish approach to composing multiple DSFMs in a consistent but non-intrusive fashion.

4.3.1 The Allocation Domain as the Unifying Concept

Whilst engineering domains provide a facility for managing the complexity of platform development and for separating overall design activity into manageable and relatively independent parts, the design of the platform is clearly not merely a collection of engineering domains (and their respective design artefacts). Interdependencies between engineering domains are unavoidable and have to be engineered just as domains themselves are engineered.

Perhaps the simplest case of dependencies between engineering domains arises when two (or more) domains reflect the traditional containment hierarchy decomposition of a platform into a number of systems. “Peer systems”, whilst designed largely independently, may interact and influence each other. The binding of interfaces between these systems clearly needs to be managed at a higher level of decomposition and, in this scenario, it can be expected that the interfaces are well-identified at the earliest stages of the design. However, less clear relationships between domains may also exist. For example, even within the traditional containment hierarchy model, it is often possible to group systems into two broad classes of *resource management systems* (e.g. power generation and distribution systems) and *user systems* (e.g. aircraft wheel braking system). In this scenario, during the early architectural design, the interfaces may be only partially identified. Indeed, in the WBS model of the previous chapter the BSCU’s dependency on the electrical power distribution system was identified. However, it is likely that the WBS’s hydraulic valves also require electrical power – these interfaces were omitted from the system description (and, consequently, from the DSFM).

Finally, as was discussed in the previous section, engineering domains do not necessarily follow the containment hierarchy model: they may capture platform decomposition from fundamentally different perspectives (such as system decomposition and structure decomposition into zones) or may address the same scope from different viewpoints. In such cases, interdependencies between engineering domains are almost never (and in many cases cannot be) captured through a simple notion of interface. As a result, these dependencies cannot be managed in a federated fashion from ‘within’ the engineering domains and require an altogether new type of engineering activities for inter-domain coordination.

Returning to the SEI model of three generic viewtypes of an architecture, this observation should not come as a surprise: the notion of the engineering domain has merely addressed a single component & connector viewtype, whilst the very premise of viewtypes is that practical systems cannot be captured from a single conceptual perspective alone (however general it may be). In particular, Clements et al define an *allocation viewtype* which documents “the relationship between system software and its development and execution environments” [33]. Abstracting from software, and adopting the terminology consistent with the previous section, it is possible to

define a notion of the "Allocation Domain". From the perspective of an individual engineering domain, allocation domains address the relationship between the engineering domain and its development and execution environments (captured in other engineering domains). In other words, and from the perspective of the entire platform, allocation domains relate two or more engineering domains by relating their respective elements (see Figure 45).

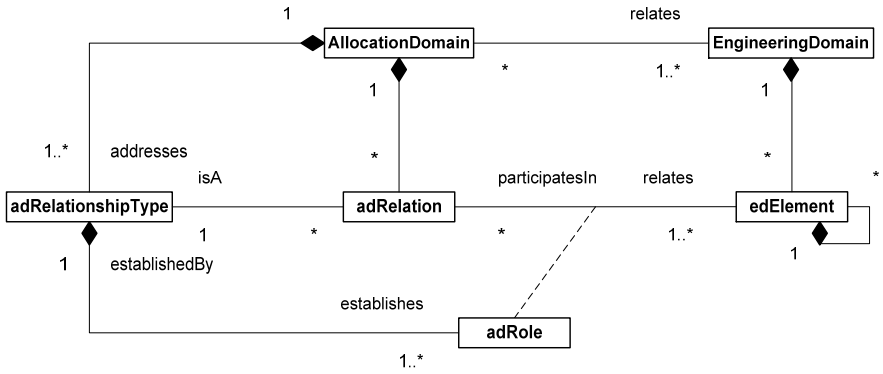


Figure 45 - Allocation Domain

Examples of engineering artefacts which fall within the Allocation Domain include interface control documents, busbar allocation databases, IMA blueprints and the zonal allocation of equipment. The predominant design concerns associated with these domains are distribution and the 'fairness' of allocation.

Table 5 - Stereotypes of Allocation Domains (depending on the scope and viewtype)

		Engineering Domains' Scopes	
		Significantly overlapping	**No significant overlap**
Engineering Domains' Viewpoints	**Equivalent / similar**	*Inconsistent / incoherent allocation of concerns to engineering domains* Allocation domains may be impossible to define (requiring manual consolidation of the 'elementary' engineering domains and their models into a new engineering domain) or are defined on entirely ad hoc basis	*"Peer" domains* Engineering domains represent different products of a single platform decomposition hierarchy. The allocation domain is predominantly concerned with the identification and binding of interfaces (that may not be explicitly identified within the engineering domains, especially at the earlier stages of design)
	Dissimilar	*Alternative views* Engineering domains represent different (partial) views of the same artefacts (e.g. a fuel provision versus leaks study 'views' of aircraft fuel system). The allocation domain relates elements in different views (either through explicit mapping or by adopting a consistent naming convention) and, in some cases, establishes consistency constraints (e.g. mutually exclusive conditions) across engineering domains	*Different semantic spaces* Engineering domains represent products of orthogonal decompositions of the same platform (e.g. system decomposition versus structural decomposition or conceptual decomposition of controller software versus 'physical' organisation of computational platform & communications network). Allocation domains typically explicitly define both the relationship between elements and constraints across both engineering domains.

Whilst the precise nature of the relationship is entirely context dependent and may range from a one-to-one interface binding to spatial relationships, based on the scope and viewpoints of the engineering domains being related it is possible to recognise three stereotypes of relationships (see Table 5, above):

- "Peer" domains;
- Alternative views;
- Different "semantic spaces" (orthogonal hierarchies).

Identification of these three stereotypes of allocation domains provides a facility for rationalising how failure behaviour of one engineering domain may be perceived from the perspective of another DSFM and, consequently, how DSFMs can be composed to reflect relationships of an allocation domain.

4.3.2 DSFM Interfaces and Composition

Whilst, as was mentioned earlier, the definition and analysis of failure logic models for individual engineering domains is beneficial, such analysis always requires some simplifying assumptions to be made about the context of the engineering domain. These assumptions can be explicit (e.g. recorded as derived safety requirements and communicated to the appropriate stakeholders) or implicit. An example of an implicit assumption can be found in the WBS case study used in the previous chapter. In that failure logic model, the characterisation of the Braking System Control Unit (BSCU) included two input failure modes associated with the two power inputs – PWR1 and PWR2. In the analysis of the WBS, these system-level input FMs were included in the minimal cut sets in a similar way to failures of WBS components. The implicit assumption (in its weakest form) that underpins such analysis is that no single failure outside the boundaries of WBS could lead to the system being exposed to both input failure modes.

One approach to the verification of the assumptions made at the level of individual DSFMs is progressive composition of the individual models and analysis of the results of such composition. This approach is more powerful than traditional common cause analyses in that it can address circular dependencies between two or more DSFMs. To illustrate in terms of the WBS, such a circular dependency may arise, if a common cause of simultaneous loss of both power inputs is dependent on some condition of the WBS itself (such as hydraulic leak potentially causing a widespread short circuit in electrical system)

However, this section demonstrates that the integration of DSFMs cannot rely merely on the pairwise binding of input and output failure modes of individual models – a broader perspective on the notion of the DSFM interface needs to be adopted. The three stereotypes of relationships

between engineering domains – identified above – are used to guide and structure the discussion on relationships between DSFMs and typical patterns of composition.

4.3.2.1 Peer Domains

Peer domains are arguably the simplest stereotype of the relationship between engineering domains. It is also the only stereotype that is addressed in existing publications on failure logic modelling methods, where it is (implicitly) assumed that the relationship between the DSFMs of different peer domains is similar to the relationship between different complex components of a single DSFM. In other words, it is assumed that the DSFMs of peer domains can be composed through a 'trivial' pairwise binding of FM interfaces.

Considering the WBS model, it is possible to see how this assumption is apparently justified. The system can be divided into two engineering domains (or major subsystems): a hydro-mechanical domain (comprising valves, pumps and brakes of two redundant lines) and controller domain (comprising BSCU) – since the design is likely to be carried out by different teams of engineers. However, the two DSFMs are clearly composed through failure mode interfaces.

This justification is clearly flawed. The DSFMs of the BSCU and the hydro-mechanical section of the WBS are only composable because the FM interfaces of all relevant components are coordinated in the WBS model. If the model were to be composed from *independently constructed* segments, it would not be possible to guarantee the equivalence of input and output failure modes; for example, there could be mismatches in the level of granularity of failure modes adopted on the BSCU and hydro-mechanical 'sides' of the DSFMs boundary leading to an inability to define a naïve one-to-one FM interface 'binding'.

In general, it can be observed that *when two or more DSFMs are composed through identified FM interfaces it may be necessary to 'convert' failure mode flows from the FM Class vocabulary established in the source model into the different vocabulary established in the target model*. Whilst they are in many cases trivial, such translations are not necessarily one-to-one mappings: different output failure modes (of one DSFM) may be translated into a single input FM (of another DSFM), output FMs may be unmapped or may translate to more than one input FM[33]. For example, in the WBS model each of the power inputs of the BSCU (*PWR1* and *PWR2*) is associated with a single *omission of power* input FM. The output FMs of the electrical system associated with the busbars may, however, be more detailed and may include too low and

[33] This typically requires the definition of one or several normal events at the point of translation in order to "select" one of the mutually exclusive interpretations (the engineering semantics of such normal events is similar to that of conditioning events and inhibit gates of the traditional fault tree analysis)

intermittent/unreliable a power provision in addition to the omission; all of these detailed FMs will be mapped onto a single input FM of the WBS DSFM (Figure 46) whilst the DSFMs of other systems may be able to utilise more detailed electrical FMs directly.

Finally, the assumption that FM interfaces between two DSFMs of 'peer' engineering domains will be always identified is in itself too strong. At the early, conceptual, stages of a design it is likely that the efforts of design and safety engineers will be focussed on a subset of 'primary concerns', while the interfaces with other systems (or domains) of the platform, if perceived as being well-understood and posing low project risk, may be postponed to later design iterations. The interfaces between the WBS valves and the aircraft's electrical power system may be seen as an example: being often controlled by low voltage electrical stimuli (e.g. from the BSCU), such valves are likely to require electrical power to amplify control signals and to overcome resistance forces posed by internal loading, friction and high hydraulic pressure. Composition of the DSFMs of peer domains in the absence of a clear definition of FM interfaces is similar to the composition of the DSFMs associated with different views discussed in the next section.

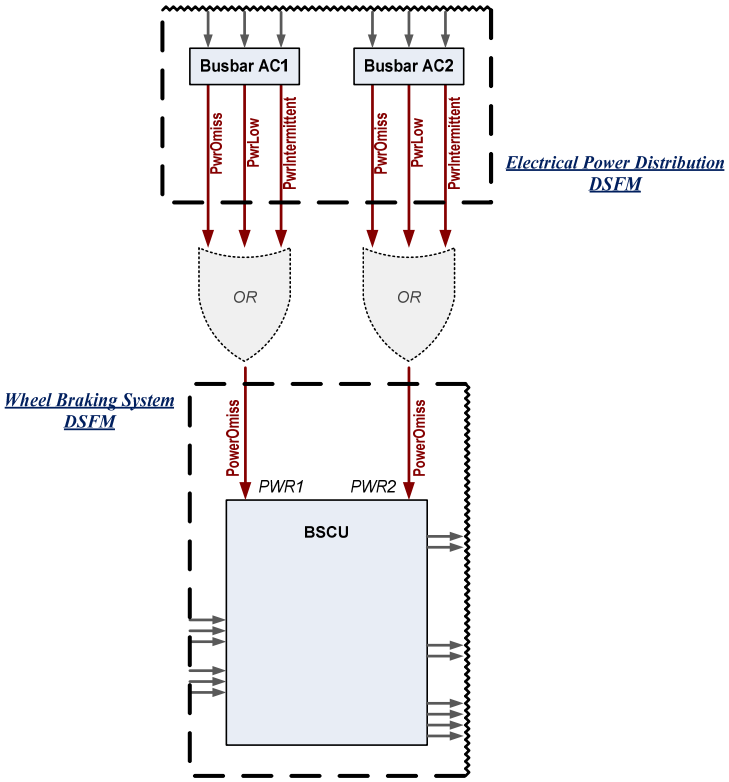


Figure 46 - Peer Domains: Composition of Electrical and Wheel Braking Systems' DSFMs

4.3.2.2 Alternative Views

Composition of the DSFMs of engineering domains which address alternative views of the same scope poses significant challenges and highlights a limitation of the FLMM defined in the previous chapter. In this context it is unlikely that dependencies between the DSFMs are captured

in terms of input and output Failure Modes. Indeed, for well-aligned engineering domains, such identification could require the definition of 'free' input FMs for every component of the DSFMs concerned.

Nevertheless, failure behaviour in one 'view' could clearly affect corresponding components in another view's DSFM. However, in the absence of a FM interface, the effect will appear to be spontaneous from the perspective of the later DSFM. To accommodate such a relationship between components (of different DSFMs) it is necessary to enhance the definition of *failure* presented in previous chapter[34].

Failure (revised):
An accidental and undesirable momentary event which is – from the restricted perspective of the scope and viewtype of the relevant domain – spontaneous and internal to the component and which may have a persistent detrimental effect on the component's ability to fulfil its operational requirements.

The new definition highlights that failures can be considered spontaneous only from the perspective of a particular engineering domain. When DSFMs are composed in absence of a well-defined FM interface, the *failures and failure modes of one DSFM can manifest themselves as (or 'cause') failures in another DSFM.*

For example, if the hydraulic section of the WBS was modelled in two DSFMs, one dedicated to the propagation of hydraulic pressure and another to the propagation of leaks, then the failures and failure modes of the "Leaks DSFM" would appear as failures in the "Pressure DSFM". In particular a *Rupture* failure of the *Green Meter Valve* in the Leaks DSFM would result in propagation of the *Leak* FM through this model and, at the same time, in an apparent *JamClosed* failure of the *Green Meter Valve* of the Pressure DSFM (since the immediate effect of a leak is a loss of pressure downstream of the source). Furthermore, the *Leak* FM would propagate through the former DSFM until it reaches the *Green Pump* where it would 'cause' an apparent *TotalLoss* failure of the pump in the Pressure DSFM (see Figure 47).

The metamodel clearly needs to be extended to allow for the failure of components to be associated with an external cause which may be another failure, a normal event or a logical condition over one or more output failure mode propositions (provided that the cause and the 'target' failure belong to components in different DSFMs). A failure may have a number of such external causes in addition to having the capacity to occur spontaneously (as specified in its probabilistic characterisation). It is also important to stress that, in general, the relation between a

[34] The modified part of the definition is underlined.

failure and its causes (even when these are other internal failures) is *not* symmetric. For instance, in the above example, a mechanical failure of the green meter valve (captured as *JamClosed* failure in pressure DSFM) clearly would not manifest as a *Rupture* in the leaks' DSFM.

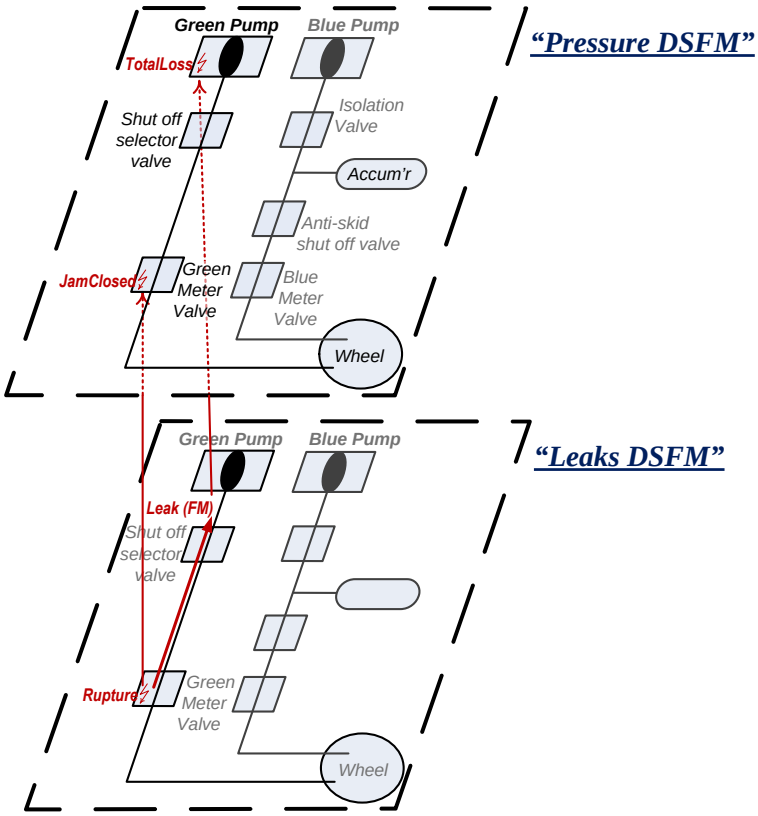


Figure 47 - Alternative Views: Composition of Leaks and Pressure DSFMs (WBS)

4.3.2.3 Different Semantic Spaces

The nature of the relationship between the DSFMs related to products of different platform decomposition hierarchies is similar to the relationship between different views discussed above: failures and failure modes captured in one DSFM will generally manifest themselves as one or more apparently spontaneous failures in another model. However, the significant difference in the semantics of the domains (and of their respective DSFMs) makes the relationships between causes and effects more complex. There are two broad aspects to this complexity:

- The relationships are rarely one-to-one and, unlike one-to-many relationships between peer DSFMs, often cannot be localised. For example, a single zone of an aircraft is typically related to components in a large number of systems. These components may not naturally form a coherent subsystem from the perspective of the system models and often belong to different systems altogether.
- The relationships may themselves be constrained and may thus give rise to non-trivial behaviour. For example, whilst from the perspective of an engineering domain concerned with fire propagation through aircraft zones, combustion can be treated as a spontaneous

failure, in order to integrate the DSFM with models of the aircraft systems it is necessary to recognise that combustion may require the presence of inflammable material (associated with hydraulic fluid or fuel leaks) as well as either high temperature for auto-ignition or an explicit source of ignition (e.g. a short circuit). Furthermore, in the later case the presence of inflammable material must precede ignition to result in combustion.

As a result it is often necessary to introduce additional structure and, in some cases, dynamic behaviour in order to 'bridge the semantic gap' between two engineering domains and to make relationships clear and easier to review. FLM components – with their failure modes, states and failures as well as the ability to nest components – already provide the necessary facilities for structuring complex relationships. When used for integrating DSFMs, however, these components do not 'trace' to any platform element, but rather capture more abstract concepts; to distinguish these FLM components from the components that *do* represent concrete elements of the platform the former are referred to as "virtual" or "translation" components. For two or more composed DSFMs, translation components, together with interface bindings and causal links to and from DSFM failures, form a "translation layer" between (and, thus, external to) the DSFMs themselves. Whilst the individual DSFMs relate to the engineering domains, the translation layer relates to the allocation domains. The translation layer, in principle, facilitates non-intrusive composition of the DSFMs; however, to achieve this, the layer has visibility of all of the events and failure modes in the DSFMs.

An example of such a translation layer is presented in the case study (section 4.6) where DSFMs of the Wheel Braking System and (simplified) Integrated Modular Avionics (IMA) are composed into a single failure logic model. The translation components in this case study are *Partitions* and *Virtual Links* which provide a 'conceptual bridge' between the WBS 'world' of software components and dataflow paths and the 'IMA world' of computation modules, end nodes and network switches.

It is important to point out that the composition of 'peer' DSFMs and models which capture different views of the same scope may require similar translation components (for instance, see the example in Figure 46 above, where each OR gate is effectively a simple stateless component). The only difference between this and the DSFMs defined for different semantic levels is the typical complexity of the translation.

Finally, whilst the relationships between DSFMs may be complex, it is observed that their structure is typically regular and repeated for all relationships of the same type. This regular structure can be exploited for the semi-automated generation of the translation layer (and, thus, the semi-automated composition of DSFMs) from various engineering artefacts of allocation domains (such as IMA blueprints and various allocation databases).

4.4 Defining Composable DSFMs

The DSFM composition approach presented above effectively extends the notion of the FLM component interface. In addition to traditionally recognised 'hard' interfaces[35] formed by components' input and output failure modes, the approach stipulates that component failures should be considered as 'soft interfaces' in the context of DSFM composition. Together with the notion of "virtual components" and the "translation layer" this extension improves the composability of failure logic models.

However, the approach itself does not guarantee that particular DSFMs will be composable: even with the extended *notion* of the interface and with the capacity to 'fix' mismatches in the translation layer, it will not be possible to compose *concrete* models that do not have a sufficiently expressive interface. Therefore, composability characteristics must be, to some extent, 'built into' the DSFMs at the time of specification.

This section outlines some broad principles which should be followed in order to improve the composability of the models.

4.4.1 The Consistency of FM Interfaces

In the previous sections it was stated that the simplest form of DSFM composition is the binding of FM interfaces (such as are typically found in the composition of DSFMs of "peer domains"). However, the effort required for this composition can be increased by a mismatch between the vocabularies of FM Classes employed in different DSFMs. Whilst proactive steps taken to ensure consistency between vocabularies can clearly facilitate effective composition, the existence of too stringent characterisation schemas – which dictate precise FM classes to be used for all DSFMs of the platform – could undermine the very purpose of platform decomposition into domains such as the independence of engineering organisations and the effective separation of concerns. Consequently, the degree of coordination of FM Class vocabularies for a particular platform should be a result of a considered trade-off between ease of composition and effectiveness of (individual) DSFM construction.

However, as a minimum, some broad principles for the elicitation of FM Classes can be stated for the vast majority of platforms, systems and engineering domains and can ensure a shared strategy of elicitation of FM classes whilst providing sufficient tactical flexibility for the detailed specification. Some examples of such general principles are listed below, based on the notions of

[35] In the sense that such input interfaces are expected to be bound to the outputs of some components and should not remain as "free inputs" after all of the models for the platform are integrated.

deviation, *interaction* and *dependency path*. These notions are related in a hierarchical fashion with deviations (or FM classes) being defined with respect to particular interactions and interactions being enabled by particular dependency paths (often formed by functionally passive components). *It is beneficial for that hierarchy to be reflected in the hierarchical organisation of FM Groups in DSFMs.*

Dependency Paths: both Intended and Unintended Paths must be considered

Whether a dependency path is intended or unintended may depend on the view point of the engineering domain. For example, the pipework of the hydraulic section of the WBS is an intended path for interactions in terms of pressure but can be seen as an unintended path from the perspective of electrical interactions (such as the valves' control signals) which can be exploited in a DSFM by short-circuit conditions.

Interactions: both Intentional and Unintentional Interactions must be considered

The correct behaviour of the system does not merely depend on a lack of deviations in intentional interactions: entirely unintentional interactions may take place, cause component damage and – ultimately – result in unsafe system- or platform- level conditions. Unintentional interactions are notoriously dependent on the implementation technology and may range from leaks and short circuits (for hydraulic and electrical dependency paths) to jitter and writer locking (for computer networks and software communications). Unintentional interactions are typically represented as commission failure modes in DSFMs; however, further refinement of this general FM class is often beneficial. Finally, unintentional interactions can take place in the *same or opposite direction* to the intentional ones.

These first two principles can be found in the CENELEC 50129 standard [54].

Interactions: in identifying possible interactions, flows of Energy, Matter and Information must be considered

The three types of flows are not mutually exclusive (for example, whilst the primary purpose of computer network is to enable information flow, this is achieved through flows of energy). Considering all three types of flows not only helps to clarify the viewpoint of the engineering domain and the scope of the DSFM, but also ensures that all aspects of interactions are considered and that the DSFM FM class vocabulary is likely to be sufficiently rich.

Deviations: use standard sets of Guidewords to identify possible FM Classes

The adopted sets of guidewords can be derived based on experience in a particular industrial sector or, alternatively, generic guidewords – such as those suggested for HAZOP analysis [85] – can be used.

Particularly advantageous in ensuring consistency between DSFMs is the adaptation of *hierarchical* guideword- and FM classes- schemas. At the top of such a hierarchy are three broad 'domains' of deviation: *Provision, Timing* and *Value.* At the next level are abstract guidewords: *Omission, Commission, Early, Late, Too High Value* and *Too Low Value*. Lower levels of the hierarchy provide interpretations of abstract guidewords in the context of a concrete implementation technology and – lower yet – application-specific FM classes (where issues such as detectability and the potential for permanent damage to other components are considered).

Finally, whilst Functional Hazard Analysis is incapable of identifying *all* dependencies between a platform's engineering domains, it can still be used in the context of DSFM composition to identify some of the key dependencies (typically in terms of intentional interactions between peer domains) and, thus, to further minimise the likely incidence of unidentified or incompatible interfaces between models. Ideally, FHA should be conducted iteratively and incrementally throughout most (if not all) of the design process alongside the DSFM-centred platform assessment.

4.4.2 Richness of 'Soft Interfaces'

As stated above, component failures provide soft interfaces between DSFMs which improve the models' composability characteristics. However, care should be taken during DSFM construction to provide a sufficiently rich and not over-constrained interface.

In particular, in identifying failures of components, *events arising from other domains have to be considered*. The WBS case study has included examples of such events – 'failures' of software BSCU components. If taken to the extreme this principle may be interpreted as a requirement to define failures for every output FM of the component even when the failure has no viable interpretation within the scope and the viewpoint of the engineering domain at hand. Such hypothetical failures can be removed from the results of DSFM analysis through relatively trivial post-processing.

However, this extreme approach may be perceived as impractical in the industrial context and a compromise between the composability of the model and its complexity may need to be found. This trade-off can be informed (at least in part) by the traditional Common Cause Analysis which, when based on sound engineering judgement and experience, can identify the *likely* key threats posed by other domains of the platform. CCA can thus be used to verify whether the DSFM is likely to provide sufficient interfaces for composition with other models and to control the risks of modelling roll-backs.

The second potential threat to DSFM composability and the appropriateness of soft interfaces is posed by over-constrained failure state transition conditions. As was stated in the previous chapter, the transition *guards must not prevent a failure from being triggered* even if, given the state of component and input FMs, it will have no persistent and tangible effect on the component's behaviour. Instead, in such cases, transitions to the same state should be declared explicitly. Failures should be prohibited by guards only in circumstances where the physical phenomena they model cannot occur. Whilst it is important for the model analysis, this principle is even more critical for the composability of DSFMs since seemingly inconsequential component failures may have a significant effect in *other* DSFMs.

4.4.3 The Granularity and Scope of Model Architectures

Whilst it is clearly necessary to take action to ensure the consistency and adequacy of the DSFM interfaces (whether in terms of FMs and their classes or in terms of component failures) in order to facilitate definition of composable DSFMs, such action is not in itself sufficient – certain qualities of the architecture and the scope of the models may render them non-composable.

The two key principles that need to be followed in establishing the architecture of a DSFM are as follows:

i. *Functionally passive components (including conceptual connectors) must be modelled explicitly;*
ii. *DSFM components which are introduced solely for the purpose of model analysis and which represent the context of the engineering domain must be identified clearly.*

The first principle has already been mentioned in the previous chapter. Functionally passive components – such as wires, pipework or conceptual software 'dataflow' connectors – may themselves fail and, thus, affect the failure logic of the engineering domain. Whilst in the context of design and assessment of 'monolithic' systems, omission of these components from the failure logic models can sometimes be justified (for a failure of a functionally passive component a failure of an active component with an identical effect can frequently be found), the justification is unlikely to hold in the context of safety-critical platform development and DSFM composition. Components which appear passive in one domain may rely on complex and functionally active mechanisms in other engineering domains. For example, apparently trivial dataflows between software components of an aircraft system (e.g. the WBS) can be supported by complex mechanisms in engineering domains concerned with the on-board computation and communication infrastructure (e.g. Integrated Modular Avionics). Ability to compose respective DSFMs in an efficient and clear fashion is clearly dependent on the presence of appropriate components (or groups of components) in all relevant DSFMs.

Finally, functionally passive components of the systems (e.g. the pipework and wiring of the hydraulic and electrical power generation and distribution systems respectively) are frequently most extensive in 'geometrical' and installation models of the platform and are therefore most exposed to the threats arising from geometrical engineering domains. As a result, unless such components are explicitly modelled (along with their failures) the DSFM composition may yield unduly optimistic results.

The second principle listed above recognises that the analysis of DSFMs often requires introduction of components that emulate domain's environment. These components are typically not only abstract representations of the other domains but are also *simplified* representations. For example, the WBS DSFM included two pumps as explicit basic components, even though the system is unlikely to generate its own hydraulic power but will rather depend on aircraft's hydraulic power generation and distribution system. Modelling the pumps was nevertheless advantageous, since it enabled consideration of the likely effects of WBS leaks on the failure logic of the system. However, this representation is significantly simplified and once the DSFM for the hydraulic system is available and composed with the WBS DSFM the WBS pumps components should be removed.

In general, DSFMs and their associated documentation should identify the 'core' part of the model, which represents the engineering domain concerned and is reusable in the context of other models, as well as that part of the model which forms a simplification 'wrapper' aimed at enabling early analysis.

4.5. Instantiation in AltaRica

This section shows how DSFMs specified in the AltaRica OCAS can be composed. The composition of models is performed over three concepts:

- DSFMs which are individual models and are specified in AltaRica as shown in the previous chapter.
- Translation components which convert failure modes between vocabularies of classes and provide a facility for structuring relationships between the DSFMs. Again, these are essentially FLM components and are specified according to the schema of the previous chapter.
- External causes of DSFM failures which require new constructs of the specification language.

This section focuses exclusively on the last concept. The specification of external causes in terms of failures is discussed first followed by failure mode causes. Both scenarios rely on the same language construct – synchronisation.

4.5.1 AltaRica OCAS Synchronisations

The description of AltaRica in the previous chapter showed how components interact in terms of flow interfaces (which were used to specify input and output FMs) connected by flows or complex components assertions (which were used to model FM Flows).

However, the AltaRica language supports another type of component communication – *synchronisations* – which establishes relationships between components' *events*. The OCAS dialect of the language supports three types of synchronisations:

- Synchronization (which in this thesis is referred to a "strong synchronisation" to avoid confusion)
- CCF[36]
- Diffusion.

Declared at the level of any complex component (in OCAS terms – either an equipment or a system), synchronisations essentially declare a new implicit event. This event (i.e. the synchronisation itself) may be associated with a probability law. Formally, synchronisation establishes a relationship between this event and one or more events of basic components (or other, lower-level, synchronisations). The relationship between the events is dependent on the type of synchronisation.

Strong synchronisations force all synchronised events to occur at the same time (and do not allow any single or a subset of events to occur independently). This synchronisation essentially substitutes all synchronised events, and imposes, where specified, its own probability law instead of the laws of constituent events. The synchronisation can only be triggered when the guards associated with all of the synchronised events evaluate to *true*.

In contrast, *CCF* – or "weak synchronisation" – does not substitute the synchronised events but rather establishes a new external common cause. Whenever the CCF is triggered it triggers all synchronised events whose guards happen to evaluate to true at the time. However, CCF itself is not predicated on any guards and can be triggered at any time (in accordance with its probabilistic

[36] The abbreviation stands for Common Cause Failure (Whilst AltaRica is a general specification language, the Cecilia OCAS suite was developed by Dassault Aviation specifically for the purpose of model-based safety assessment). Despite its name, as shown below, CCF synchronisations are not suitable for modelling relationships between different DSFMs

characterisation). Subsequently, events synchronised by the CCF can still be triggered independently, in accordance with their own probability laws. This will not affect either peer synchronised events or the CCF itself.

Finally, *Diffusion* is another form of weak synchronisation, in that it can be triggered at any time regardless of the guards of synchronised events. As with CCF, once it is triggered, diffusion will 'force' all (if any) synchronised events whose guards are true to 'fire'. However, unlike CCF (but similar to the strong synchronisation), diffusion 'hides' synchronised events in that, once they have been synchronised, these events cannot be triggered individually but rather only through the synchronisation.

In terms of a need to specify the external cause(s) of failures, the CCF synchronisation may at first appear suitable (since it would still allow failures to occur independently). However, this synchronisation establishes a *new* and unguarded 'cause' event and cannot be linked to the event that is already declared in another component (e.g. within another DSFM). At the same time, strong synchronisation between cause and effect failures is clearly unsuitable since it would only allow both failures to occur simultaneously.

4.5.2 Emulating Dependent Weak Synchronisation

Ideally, to implement a causal relationship between two failures a new type of synchronisation is necessary which will link synchronisation to an *existing* cause (declared as an event in some component) and will inherit its probability law.

In the absence of such a synchronisation in OCAS, the relationship has to be emulated through other language constructs both on the "cause" and "effect" sides.

Firstly, for each failure that has an external cause, a new event must be declared in the same component. Every transition equation that refers to the failure must be replicated with the failure replaced by the newly-declared event. Composition of hypothetical "Leaks" and "Pressure" DSFMs (see Figure 47 above) is used to illustrate this. Figure 48 shows a transformation of the specification of Green Meter Valve of the "Pressure DSFM".

Secondly, for each failure that is a *cause* of another failure (in another DSFM) a new Boolean state variable (with initial value *false*) and a new event (assigned an instantaneous – *Dirac(0)* – law) must be declared. All state transition equations referring to the original failure must be modified to assign a newly defined state variable a value of *true* (in addition to any original state transitions). Also, a new state transition must be added to the component characterisation; the guard of this transition is the new state variable being *true*, the trigger is the newly declared event

and the effect is assignment of *false* to the state variable. The effect of this transformation is that whenever a 'cause' failure takes place in addition to its original effect a corresponding instantaneous event is immediately issued. Figure 49 shows the transformation of the Blue Meter Valve specification in the "Leaks DSFM".

```
node GreenMeterValve
 flow
  HydIn : WBS_PressureFMs : in ;
  CTRL : WBS_ControlFMs : in ;
  HydOut : WBS_PressureFMs : out ;
 state
  FailSt : {OK,StuckOpen,StuckClosed} ;
 event
  SpringFailure, JamClosed, JamOpen, MotorFailure, Rupture
;
 init
  FailSt := OK ;
 trans
  // Failures & Failure State Transitions:
  FailSt != StuckOpen |- MotorFailure
                                  -> FailSt1 := StuckClosed;
  FailSt != StuckClosed |- SpringFailure
                                  -> FailSt1 := StuckOpe
  true |- JamOpen -> FailSt1 := StuckOpen;
  true |- JamClosed -> FailSt1 := StuckClosed;
 assert
  // Failure Mode propagation conditions
```

```
node GreenMeterValve
 flow
  HydIn : WBS_PressureFMs : in ;
  CTRL : WBS_ControlFMs : in ;
  HydOut : WBS_PressureFMs : out ;
 state
  FailSt : {OK,StuckOpen,StuckClosed} ;
 event
  SpringFailure, JamClosed, JamOpen, MotorFailure, Rupture,
  ExternalJamClosed ;
 init
  FailSt := OK ;
 trans
  // Failures & Failure State Transitions:
  FailSt != StuckOpen |- MotorFailure
                                  -> FailSt1 := StuckClosed;
  FailSt != StuckClosed |- SpringFailure
                                  -> FailSt1 := StuckOpen;
  true |- JamOpen -> FailSt1 := StuckOpen;
  true |- JamClosed -> FailSt1 := StuckClosed;

  // Duplicated transition (external cause):
  true |- ExternalJamClosed -> FailSt1 := StuckClosed;
assert
  // Failure Mode propagation conditions
```

Figure 48 - Transformation of the Target ('Effect') Component: Valve in "Pressure DSFM"

```
node GreenMeterValve
 flow
  LeakIn : {ok,Leak} : in ;
  LeakOut : {ok,Leak} : out ;
 state
  FailSt : {OK,Leaking} ;
 event
  Rupture ;
 init
  FailSt := OK ;
 trans
  // Failure & Failure State Transition
  FailSt = OK |- Rupture -> FailSt := Leaking;
assert
  // Failure Propagation
  LeakOut = ( case {
              LeakIn = ok
              and FailSt = OK : Leak,

              else ok})
 extern
  law <event Rupture> = exponential(2.5e-3) ;
edon
```

```
node GreenMeterValve
 flow
  LeakIn : {ok,Leak} : in ;
  LeakOut : {ok,Leak} : out ;
 state
  FailSt : {OK,Leaking} ;
  BroadcastPending : bool ;
 event
  Rupture, BroadcastRupture ;
 init
  FailSt := OK ;
  BroadcastPending := false ;
 trans
  // Failure & Failure State Transition
  FailSt = OK |- Rupture -> FailSt := Leaking, BroadcastPending := true;

  // Broadcasting failure:
  BroadcastPending |- BroadcastRupture -> BroadcastPending := false;
assert
  // Failure Propagation
  LeakOut = ( case {
              LeakIn = ok
              and FailSt = OK : Leak,

              else ok})
 extern
  law <event Rupture> = exponential(2.5e-3) ;
  law <event BroadcastRupture> = Dirac(0) ;
edon
```

Figure 49 - Transformation of the Source ('Cause') Component: Valve in "Leaks DSFM"

Finally, the newly-created events on both the cause and effect sides of the relationship can be strongly synchronised; the synchronisation should be assigned the *Dirac(0)* law. The overall effect is that whenever a 'cause failure' occurs – the effect on the target DSFM will be identical to that of the 'effect failure'. Figure 50 shows the synchronisation specification in the Cecilia OCAS graphical interface.

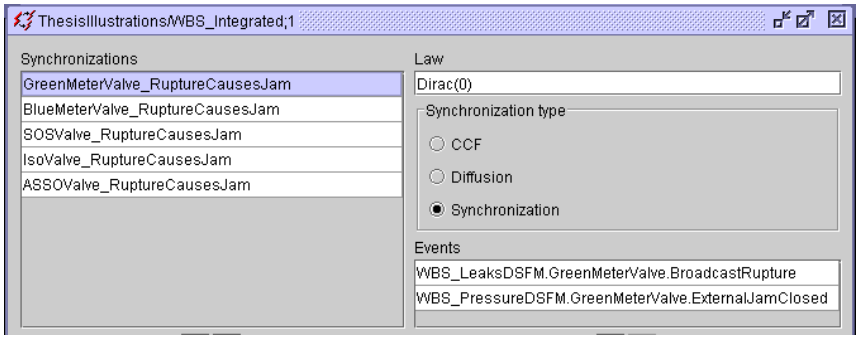


Figure 50 - Synchronisations Between "Leaks" and "Pressure" DSFMs: Green Meter Valves

This specification approach highlights the limitation of the AltaRica OCAS language as a specification notation for failure logic models, and can be said to significantly undermine the elegance of the composition approach presented in the previous section. However, the necessary transformations of models are trivial and are clearly automatable. The author is working with Dassault Aviation in order to extend the Cecilia OCAS suite either directly with the new type of synchronisation or with functionality to enable the automation of the 'emulation' approach described above.

4.5.3 Instantaneous Events: Flow-to-Event Conversion

Having established an approach for specifying the causal relationships between failures (that is AltaRica events), it is necessary to outline how failure mode causes of failures can be similarly specified.

In AltaRica, events can only be caused by higher-level synchronisations. As was described above, in case of a strong synchronisation (or dependent weak synchronisation proposed) dependency on other events can be established; however, there is no direct construct that can establish causal links from flow to an event. The solution relies on the 'conversion' of flows into events which clearly reduces the problem to the one addressed above.

Conversion is achieved through the notion of instantaneous events used in the previous chapter for void triggers of state transitions and in the specification solution above to broadcast failure occurrence. Since instantaneous events – identified by the *Dirac(0)* law – 'fire' immediately upon the guard's becoming *true* and since guards may contain a predicate over the component's input

flow, specification of a flow value detector which emits events whenever a flow takes particular value is in fact trivial. Figure 51 shows the specification of such detector for a Boolean flow and implements both rising-edge and falling-edge detection (signified by the emission of *BecameTrue* and *BecameFalse* events respectively). In the context of DSFM integration such detectors are a particular example of a virtual translation component and reside in the translation layer of the failure logic model.

```
node FlowDetector
  flow
    FlowIn : bool : in ;
  state
    Detected : bool ;
  event
    BecameFalse, BecameTrue ;
  init
    Detected := false ;
  trans
    // Rising Edge Detection:
    FlowIn and not Detected |- BecameTrue -> Detected := true;
    // Falling Edge Detection:
    Detected and not FlowIn |- BecameFalse -> Detected := false;
  extern
    law <event BecameFalse> = Dirac(0) ;
    law <event BecameTrue> = Dirac(0) ;
edon
```

Figure 51 - Boolean Flow Detector Component

FM detection can also be incorporated into more complex components (e.g. to detect composite conditions over FM flows). However, when implemented as standalone components, detectors have an exceptionally simple and regular structure and their construction can in future be automated by the GUI of the modelling tool.

Once the flow-to-event conversion is achieved, the emitted event can be synchronised with a newly constructed event in the target component as shown in previous section.

4.6 Case Study: Common Computational Platform

This section demonstrates a non-intrusive composition of Domain-Specific Failure Logic Models (DSFMs) through an example of the integration of the WBS DSFM with a model of a hypothetical common aircraft computation and communications infrastructure.

In terms of the WBS DSFM a model constructed in the previous chapter is re-used with relatively trivial modifications to the complex BSCU component. Namely, functionally-passive data communication paths are added as explicit components (see Figure 52 and Figure 53). Furthermore, minor modifications are made to the COM and MON components to compensate for the lack of appropriate form of synchronisation in AltaRica (as discussed in Section 4.5. above).

These modifications are described in Appendix C (section C1). For the purpose of the discussion here it is necessary to stress that neither set of modifications compromises the principles of composition of independently defined failure logic models: the former modification merely enforces good modelling practice that was advocated throughout this and previous chapter, whilst the later modification is algorithmic and is only necessary due to the limitation of the chosen specification language.

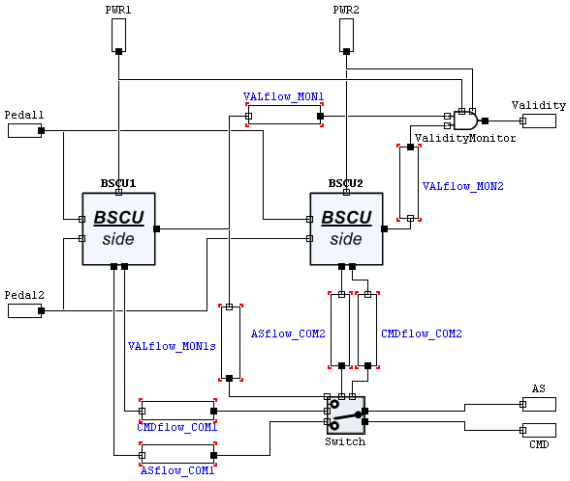


Figure 52 - Revised Model Architecture of BSCU

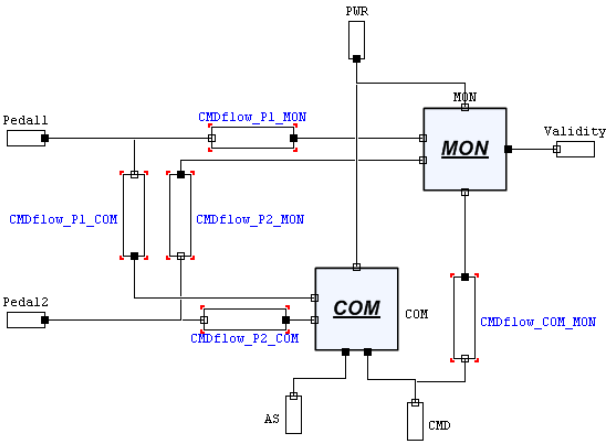


Figure 53 - Revised Model Architecture of a BSCU Channel

The computation and communications infrastructure is akin to the civil aviation Integrated Modular Avionics (IMA), consistent with the ARINC 653 specification [8], which utilises an Avionics Full-Duplex Switched Ethernet (AFDX) data network (consistent with the ARINC 664 specification [7]). Following the civil aviation convention, the infrastructure (for simplicity referred to as “IMA”) is viewed as an aircraft system in its own right. Since the purpose of the case study is to demonstrate and evaluate the *composition* approach presented in this chapter, the model of the infrastructure is simplified.

In terms of the framework of engineering and allocation domains presented in this chapter, the engineering domains of the WBS and IMA are semantically dissimilar as they are concerned with orthogonal decomposition hierarchies (conceptual software components versus a physical network). The prevailing relationship between the failure logic models of these domains is concerned with situations in which IMA output failure modes cause seemingly spontaneous failures of BSCU components in the WBS DSFM.

The integration between IMA and WBS DSFMs is achieved through a translation layer which captures the view of WBS’s BSCU components from the perspective of the IMA. The layer allows us to ‘bridge the gap’ between semantics of two models. The translation components capture the effect of output failure modes of the IMA components on individual partitions and virtual links which, in turn, correspond uniquely to the BSCU components in WBS DSFM. This

overall approach to integration provides a means for rationalising and structuring otherwise unmanageably numerous dependencies between the DSFMs.

4.6.1 Overview of the Computation Infrastructure

This section presents a brief overview of the IMA architecture assumed for the purpose of the case study and the corresponding Domain Specific Failure Logic Model (DSFM). The overview is kept to the minimum necessary for describing the approach to model composition; however, a more detailed description is provided in Sections C2 and C3 of Appendix C for the architecture and the DSFM descriptions respectively.

4.6.1.1 Architecture of the Infrastructure System

The architecture of the IMA (Figure 54) comprises four Core Processing & Input/Output Modules (CPIOMs) connected to the AFDX network via dedicated End Nodes (ENs). An additional EN connects the network to a Line Replaceable Unit (LRU) of the braking system. Whilst the LRU, which hosts the Switch and the Validity Monitor of the BSCU, is considered to be part of the WBS, its End Node is considered to be part of the IMA system. End Nodes are essentially network input/output cards that are physically located inside- and powered by- their respective modules. With exception of the LRU's EN, the system is organised in the standard two side arrangement – each side powered from a separate source.

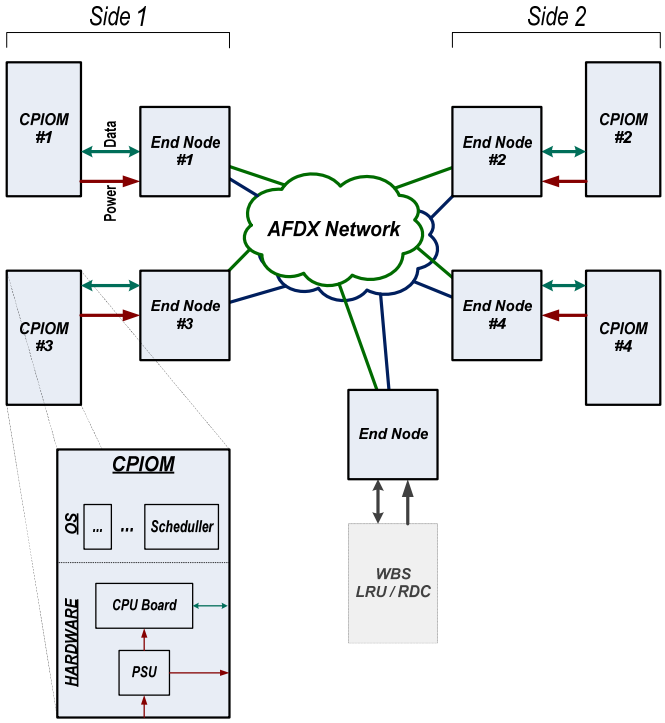


Figure 54 - Simple IMA Architecture

The AFDX network, which connects all the CPIOMs and the LRU, consists of two redundant and identical switched networks – Network A and Network B – each consisting of a pair of switches and a number of "twisted pair" cables (Figure 55). Each switch has four ports that can be connected to either ENs or other switches. Consequently, each switch supports twelve unique communication paths (from each port to each of the three other ports). Since each EN is connected to both networks, it supports four unique paths: two – *from* the CPIOM (or LRU) to each network and a further two – from each network *to* the CPIOM (or LRU).

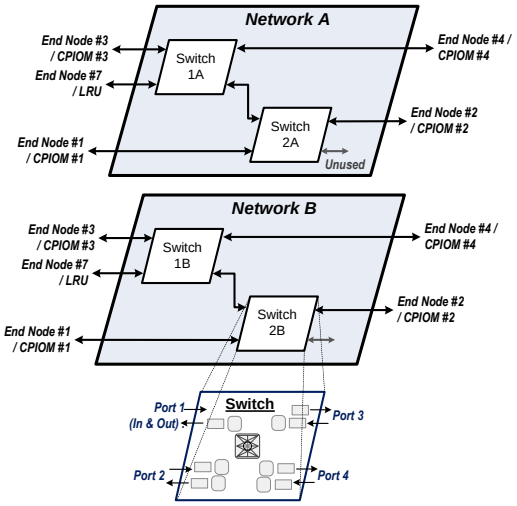


Figure 55 - Redundant AFDX Networks

4.6.1.2 DSFM of the Infrastructure System

The overall architecture of the IMA failure logic model is shown in Figure 56. Internal failure mode flows of the IMA DSFM are predominantly related to the *physical interactions* between the IMA components.

However, all major components (CPIOMs, ENs and Switches) also interact with the application software yielding a set of 'external' failure modes. In this case study the approach proposed by Conmy [34-36] is adopted in a simplified form to rationalise these interactions and failure modes. Conmy proposes six high-level IMA functions:

1) Provision of secure and timely data flow to and from applications and input/output devices
2) Controlled access to processing facilities
3) Provision of secure data storage and memory management
4) Provision of consistent execution state
5) Provision of health monitoring and failure management
6) General provision of computing capability

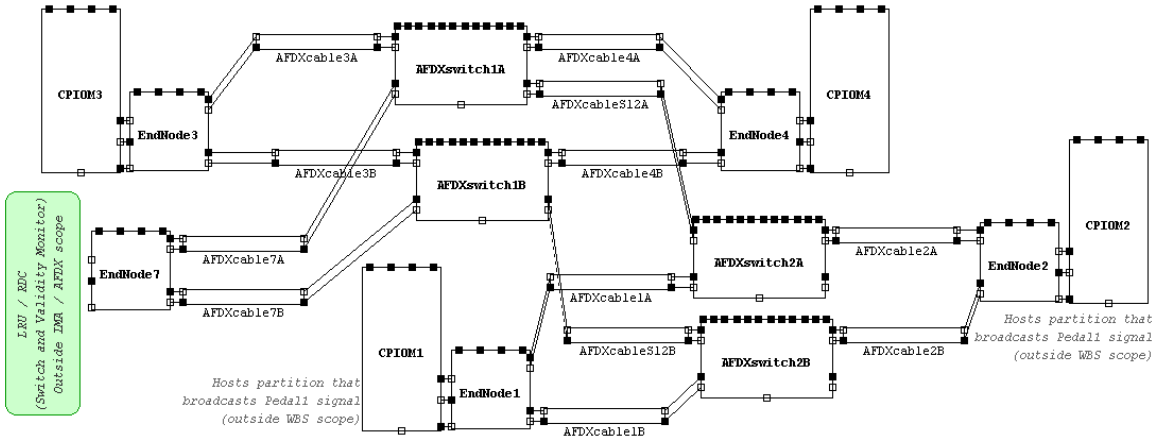


Figure 56 - Architecture of IMA DSFM (Power FM Flows not Shown)

Each of Conmy's functions can be considered as an interaction initiated by the IMA system (or more, specifically, by each CPIOM) and can be associated with the set of failure modes. Since the goal of this case study is not the construction of accurate IMA system failure logic model, but rather the demonstration and evaluation of the DSFM composition approach, Conmy's approach is simplified by omitting functions (4) and (5) from consideration. The latter is a simplification of the IMA system for the purpose of the case study, whereas, omission of the "provision of consistent execution state" function (4) can be further justified by the focus of the failure logic models on the run-time behaviour of the system.

Each of the remaining four functions is associated with a simplified set of failure modes (predominantly identified by application of the *Omission* and *Value* keywords) and modelled as an enumerated output flow of *CPIOM* components in the AltaRica model (so that each *CPIOM* has four enumerated output flows). Table 6 lists such *CPIOM* outputs, their enumeration symbols along with the brief description of the failure modes they denote.

Of course the BSCU (and other IMA subscribers) can be also affected by the malfunctions of network components (switches and end nodes); consequently, these components also have external output failure modes. It is assumed that they interact with subscribers in terms of a single function – provision of network infrastructure. This function, that can be seen as amalgamation of Conmy's functions (1) and (3), is associated with four failure mode classes: *TotalLoss*, *PartialLoss*, *AddressingCorruption* and *DataCorruption*. However, these failure modes are associated with each *path* through a network component. So, for example, the AltaRica characterisation of an end node has four enumerated output flows (with the above enumeration symbols): two for the propagation of data from the CPIOM to the redundant AFDX networks and two for receiving data (namely, *NetworkInfrastructureOutA*, *NetworkInfrastructureOutB*, *NetworkInfrastructureInA* and

NetworkInfrastructureInB respectively); similarly, the network switches have 12 sets of network infrastructure output FMs (and, thus, 12 enumerated output flows[37]).

Table 6 - Output Flows of CPIOM Components

AltaRica Output Flow	Conmy's IMA function	Enumeration Symbol	FM description
Communications	1	TotalLoss	CPIOM fails to provide support to any of the communication channels it is responsible for
		Partial Loss	CPIOM fails to provide support to one or more (but not necessarily all) communication channels it is responsible for
		AddressingFailure	CPIOM directs data to the wrong software component (partition) or output port (this may affect one or more channels)
		OK	Failure Mode Privative (function is provided as intended)
Computing	6	Incorrect	CPIOM performs computation required by one or more partitions incorrectly
		Stuck	CPIOM halts computation of one or more partitions or processes (e.g. due to corruption of process execution state or CPU failure)
		OK	FM privative
DataIntegrity	3	TotalLoss	Data stored or transmitted by, to or from all partitions supported by the CPIOM is lost/ unusable
		PartialLoss	As above but for a subset of partitions
		Corruption	Data is corrupted (affects one or more partitions)
		OK	FM privative
Scheduling	2	TotalLoss	CPIOM fails to schedule any of its partitions appropriately (including providing too little execution time)
		PartialLoss	As above for subset of partitions
		PrioritiesLost	CPIOM incapable of scheduling one or more processes as intended (e.g. due to corruption or loss of priorities data)
		OK	FM privative

4.6.2 Partition and VL Allocation

For the purpose of the case study, it is assumed that the Command and Monitor components of each BSCU channel are implemented as separate IMA partitions. Furthermore, each BSCU command component (partition) contains two processes for the calculation of braking (*CMD*) and anti-skid (*AS*) signals. Whilst partitions rely on a fixed periodic scheduling (to guarantee temporal partitioning), the exact scheduling principle for the processes is assumed to be unknown at the point of the development process reflected by the case study.

The case study further assumes that both partitions of BSCU1 are allocated to the CPIOM 3 and both partitions of BSCU2 – to CPIOM 4. The BSCU's Validity Monitor and Switch are implemented in hardware and contained within a dedicated LRU connected to the IMA's End Node 7.

[37] These are all called *Infrastructure<X>to<Y>*, where *<X>* is an incoming port of the switch and *<Y>* - outgoing (e.g. "*Infrastructure 3to2*").

Whilst the design of the sources of the electronic pedal inputs falls outside the scope of the WBS engineering domain, their 'location' in the IMA is significant for the safety assessment. It is assumed that the partitions that produce Pedal 1 and Pedal 2 data are allocated to CPIOM 1 and CPIOM 2 respectively.

Finally, it should be noted that the architecture of the hypothetical simplified IMA system is such that partition allocation to the CPIOMs uniquely determines the allocation of data communications channels to IMA switches and their ports. For example, the Pedal 1 data channel to COM2 partition is allocated to input port #2 and output port #1 of switch 2A(B) as well as to input port #4 and output port #3 of switch 1A(B).

4.6.3 Integration Layer and DSFM Composition

The DSFMs subjected to composition are expected to be provided as separate AltaRica OCAS equipment nodes (i.e. complex components) and, typically, frozen for further editing (to enforce non-intrusiveness of composition). However, in this case study, whilst the WBS DSFM is indeed 'wrapped-into' equipment, for clarity of illustration, the IMA DSFM is kept at the OCAS system level (i.e. the same model decomposition level as the translation layer described in this section).

As stated in the earlier sections of this chapter, the translation layer captures the causal relationships between various IMA output failure modes and appropriate failures of the BSCU components in the WBS DSFM as well as structuring these dependencies in – as far as practicable – a reviewable and coherent fashion. The layer therefore consists of a number of virtual translation components (some complex and some basic), flows between the IMA model and these components as well as strong instantaneous synchronisations between events in the virtual components and events in the WBS DSFM.

As far as the architecture of the layer is concerned it can be best described as the view of the WBS BSCU from the IMA perspective. Namely the IMA 'sees' *MON* and *COM* software components of the two BSCU channels as four *partitions*; similarly all BSCU dataflow components are 'seen' as *virtual links*[38]. The resultant high-level components of the translation layer (along with the graphical annotations which contextualise them) are shown in Figure 57.

[38] In this model "Virtual Link" denotes a communication between two partitions regardless of whether these partitions are located on the same or different CPIOMs (and thus of whether they do or do not rely on the AFDX network). "Path" and "channel" (used interchangeably) refers to a single physical propagation route through the redundant AFDX networks and/or the CPIOMs as well as coherent portions thereof (such as "path through an end node")

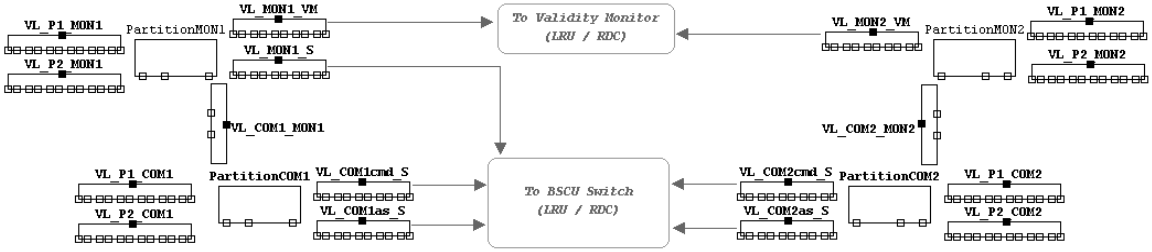


Figure 57 - Architecture of the Failure Logic Model's Translation Layer

The internal structure of these components along with relevant synchronisations and flows is described in the following sub-sections.

4.6.3.1 Software Components: Partitions and Processes

Seen by the IMA as partitions, the *COM* and *MON* software components of the BSCU are sensitive to three out of four groups of their CPIOM's external output failure modes: *Scheduling*, *DataIntegrity* and *Computing*. Consequently each partition component has three enumerated input flows linked to the output flows of the corresponding *CPIOM* (Figure 58).

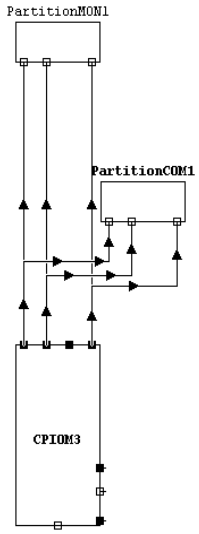


Figure 58 - FM Flows Between the CPIOM and Allocated Partitions (Side 1 Only)

Whilst the interfaces of all of a partition's components are identical, the 'internal' failure logic is to some degree dependent on the details of the corresponding BSCU components. Figure 59 shows a complete characterisation of the (identical) *MON* components' partitions (i.e. virtual components *PartitionMON1* and *PartitionMON2* in Figure 57 above). In general, a partition's exposure to an input Failure Mode results in an apparently spontaneous failure of the corresponding *MON* component of the BSCU. To model this dependence the partition component upon receiving a non-privative value on the input flow 'fires' an instantaneous event (and enters into a corresponding failure state to avoid repetitive events); this event is 'picked up' by a global

synchronisation which forces the corresponding failure in the WBS model to be fired (if permitted by its guard). For example, when the monitor partition is exposed to *TotalFailure* of *Scheduling* the instantaneous *BroadcastUnschedulled* event is issued and the partition enters a *ProcessUnschedulled* state. This event is synchronised through the *MON1_Unschedulled* or *MON2_Unschedulled* strong instantaneous synchronisation (see Figure 60) with the *ExternalProcessTerminated* event in the appropriate BSCU's *MON* component (see Section C1 in Appendix C where this event has been defined to mimic the original *ProcessTerminated* failure).

```
node IMA_homogeneousPartition
  flow
    Scheduling : {ok,TotalLoss,PartialLoss,PrioritiesLost} : in ;
    Computing : {ok,Incorrect,Stuck} : in ;
    DataIntegrity : {ok,TotalLoss,PartialLoss,Corruption} : in ;
  state
    ProcessStuck : bool ;
    ProcessDataLost : bool ;
    ProcessDataCorrupted : bool ;
    ProcessUnschedulled : bool ;
    SchedulingAffected : bool ;
    ComputingAffected : bool ;
    IntegrityAffected : bool ;
  event
    BroadcastStuck, BroadcastDataLost, BroadcastDataCorrupted, BroadcastUnschedulled,
    CONDITION_UnaffectedBySchedulingFailure, CONDITION_UnaffectedByComputingFailure,
    CONDITION_UnaffectedByDataIntegrityFailure ;
  init
    ProcessStuck := false ;
    ProcessDataLost := false ;
    ProcessDataCorrupted := false ;
    ProcessUnschedulled := false ;
    SchedulingAffected := true ;
    ComputingAffected := true ;
    IntegrityAffected := true ;
  trans
    // Conditional (Normal) events
    // Specifying which partial failures this process is (un)affected by
    SchedulingAffected |- CONDITION_UnaffectedBySchedulingFailure -> SchedulingAffected := false;
    ComputingAffected |- CONDITION_UnaffectedByComputingFailure -> ComputingAffected := false;
    IntegrityAffected |- CONDITION_UnaffectedByDataIntegrityFailure -> IntegrityAffected := false;

    // Broadcasting the effect of IMA platform FMs on the partition/process
    not ProcessStuck and (Computing = Stuck) and ComputingAffected |- BroadcastStuck -> ProcessStuck :=
true;
    not ProcessDataLost and ( (DataIntegrity = TotalLoss) or
                ((DataIntegrity = PartialLoss) and IntegrityAffected)
              ) |- BroadcastDataLost -> ProcessDataLost := true;
    not ProcessDataCorrupted and (DataIntegrity = Corruption)
                and IntegrityAffected |- BroadcastDataCorrupted -> ProcessDataCorrupted := true;
    not ProcessUnschedulled and ( (Scheduling = TotalLoss) or
                 ((Scheduling = PartialLoss) and SchedulingAffected)
              ) |- BroadcastUnschedulled -> ProcessUnschedulled := true;
  extern
    law <event BroadcastStuck> = Dirac(0) ;
    law <event BroadcastDataLost> = Dirac(0) ;
    law <event BroadcastDataCorrupted> = Dirac(0) ;
    law <event BroadcastUnschedulled> = Dirac(0) ;
edon
```

Figure 59 - AltaRica Characterisation of Homogeneous Partition Translation Component

However, many of the CPIOM failure modes (in fact all of the failure modes except *TotalLoss* of each function) do not affect all partitions. To model situations when the *MON* partition is and is

not affected a set of *normal events* and *states* is used (these states are modelled through three Boolean variables - *SchedulingAffected*, *ComputingAffected* and *IntegrityAffected* -for each of the three partition's input flows respectively). Consequently, some of the instantaneous *broadcast* events are guarded by a *conjunction* of FM assertion (flow value condition) and normal state assertion (state variable value condition).

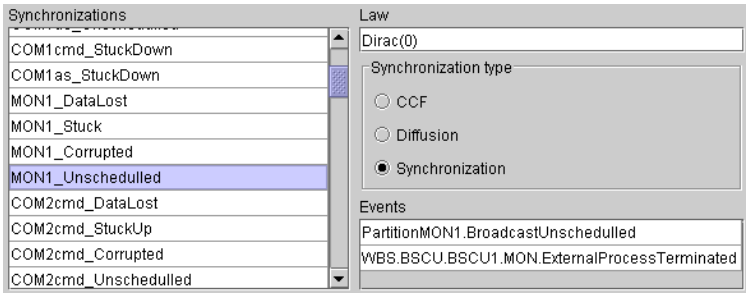


Figure 60 - Synchronisations Between Translation and WBS Components

The characterisation of the *COM* components' partitions is slightly more complex, since the BSCU's COM components contain two separate processes – for calculation of the braking command (*CMD*) and anti-skid modulation (*AS*) respectively. The partition is modelled as a complex component (Figure 61) with two basic components for each process and a common *selector component* (*PartitionSelector*) which contains conditioning events' logic and determines – whenever there is a choice – whether the partition is affected by the CPIOM's FMs (analogous with the conditioning events in *MON* partitions described above). If the selector component 'determines' that the partition is unaffected, it replaces the FM enumeration symbol by the *ok* privative (so that virtual process components simply do not see the inconsequential FMs). As with the *MON* partitions, the two main process components (*CMD* and *AS*) issue broadcast events whenever they are exposed to the appropriate FM. These components also contain further conditioning events for selecting, whenever applicable, whether an individual process is affected by the CPIOM failure mode which affects the partition.

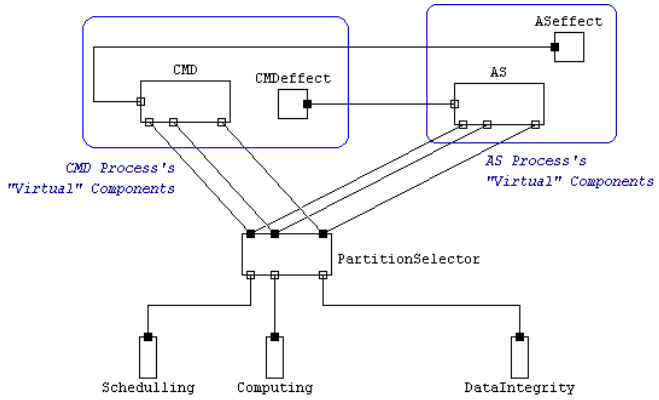


Figure 61 - Internal Structure of the COM Partition Translation Component

Whilst IMA guarantees by design that partitions cannot affect one other's execution time or data, the same protection is not afforded to the processes which reside within the same partition. For example, a *ProcessStuck* failure of the *COM's CMD* process may lead to it *monopolising* CPU time allocated to the COM partition and, thus may have the same effect on the 'co-residing' *AS* process as a *TotalLoss* of *Scheduling* by CPIOM[39]. To capture this dependency, the *COM* partition component contains two simple components *CMDeffect* and *ASeffect*. These components react to the *CMDprocessStuck* and *ASprocessStuck* failures of the appropriate BSCU's *COM* component (through global synchronisations) by exhibiting the *MonopolisesCPU* failure mode. The main virtual process components (i.e. *AS* and *CMD*) are sensitive to this FM and 'raise' a *BroadcastUnschedulled* event in response. This aspect of the failure logic of *COM* components partitions is significant as it demonstrates the *bi-directional dependency* between DSFMs. Such dependencies, where the initiating event is located in the same engineering domain as its effect(s) but the unintended interaction is not 'visible' until both domains are considered, may be overlooked by the traditional common cause analysis[40]. In contrast under the approach advocated by this thesis the logic of this propagation is captured explicitly by the integrated DSFMs and is therefore reviewable.

4.6.3.2 Software Communications: Virtual Links

BSCU dataflow components are seen from the IMA perspective as Virtual Links and can be classified into two types:

(1) **VLs internal to the CPIOM** (e.g. those corresponding to *CMDdataflow*s between the *COM* and *MON* modules of the same BSCU side)

(2) **VLs between partitions residing on different CPIOMs** which rely on the AFDX network functionality (e.g. VLs corresponding to the *pedal positions dataflows* to *COM* and *MON* modules of the BSCU)

Of course the type of VL is determined by a partition's allocation to the CPIOMs rather than by any details of the WBS DSFM.

All 'virtual' VL components are modelled as AltaRica OCAS equipment with interfaces and internal structure determined by the type of the VL above. Figure 62 shows the structure of the first type of VL; it contains two input flows corresponding to the *DataIntegrity* and *Communications* output flows (i.e. FM groups) of a CPIOM as well as two basic components – *Module* and *DSFMlink*. The former component determines the status of the VL, based on the CPIOM failure modes and, if applicable, an internal conditioning event (similar to that of partition components in

[39] Note that, since the WBS description did not include a description of the process scheduling schema, the 'worst case' of co-operative scheduling is assumed for the purpose of the DSFM integration.

[40] See section 4.7 below

the previous section); the result of the consolidation is presented on the enumerated *Status* output flow over the *ok*, *Lost*, *Misrouted* and *Corrupted* symbols (which are self-explanatory failure modes).

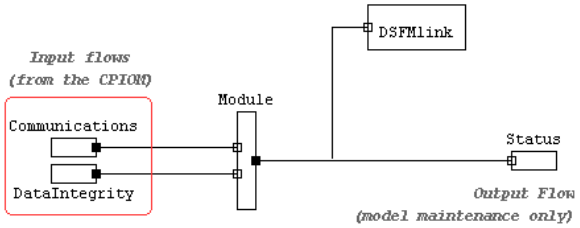


Figure 62 - Structure of a Simple VL Translation Component (CPIOM-Internal VLs)

The *DSFMlink* component is a simple *flow-to-event converter* (see section 4.5.3) which issues appropriate instantaneous events (which are, in turn, synchronised with appropriate failures of the BSCU dataflow components).

The structure of the virtual links that are supported by the AFDX network is more complex (Figure 63), as they rely on two CPIOMs, two End Nodes and one or more pairs of switches. The internal structure of VLs also reflects *partial redundancy* of communications in that for the Virtual Link to fail due to the End Node or Switches failure mode paths through *both of the AFDX networks* must be affected. The redundancy is only partial because the failure modes of *either CPIOM* that supports the VL on sending and receiving sides are single points of failure. Furthermore, undetectable *corruption of data by either AFDX network* will potentially result in corruption of the VL.

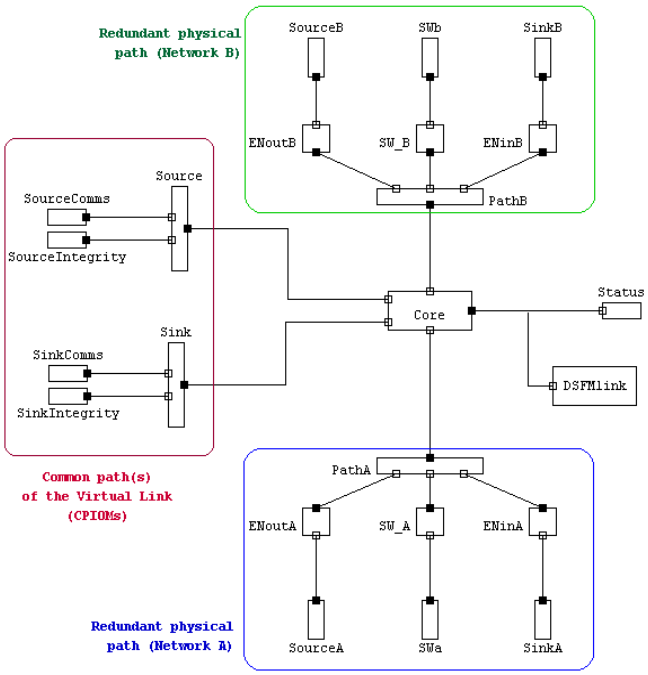


Figure 63 - Structure of a General VL Translation Component (VLs Across Different CPIOMs)

In Figure 63 the failure modes of CPIOMs are consolidated by *Source* and *Sink* components which are identical to the *Module* component of an internal VL presented earlier. *ENout_A*, *SW_A*, *ENinA*, *ENout_B*, *SW_B* and *ENinB* are identical selector components (which propagate *NetworkInfrastructure* FMs subject to conditioning events). *PathA* and *PathB* are simple stateless consolidation components which determine the overall *status of each redundant path*; similarly, *Core* consolidates the *overall status* of the VL from statuses of two common segments and two redundant paths. Finally *DSFMlink* is a flow-to-event converter / event broadcaster component as before.

Note that the highly regular structure of the VL components allows for the automated generation of VLs that are supported by *more than one* pair of network switches by trivial extrapolation of the structure presented above. Furthermore, the *regularity of the flow interface* of these translation components allows the automatic generation of – otherwise unmanageably numerous – flows between VLs and IMA components as well as synchronisations between events in VLs' *DSFMlink* components and failures of appropriate BSCU dataflow components. Figure 64 shows the flows between IMA DSFM and the translation layer components (VLs and Partitions) which were automatically generated from Partition and VL allocation spreadsheets in Microsoft Excel (the 70 necessary synchronisations between translation layer and WBS DSFM were similarly generated automatically).

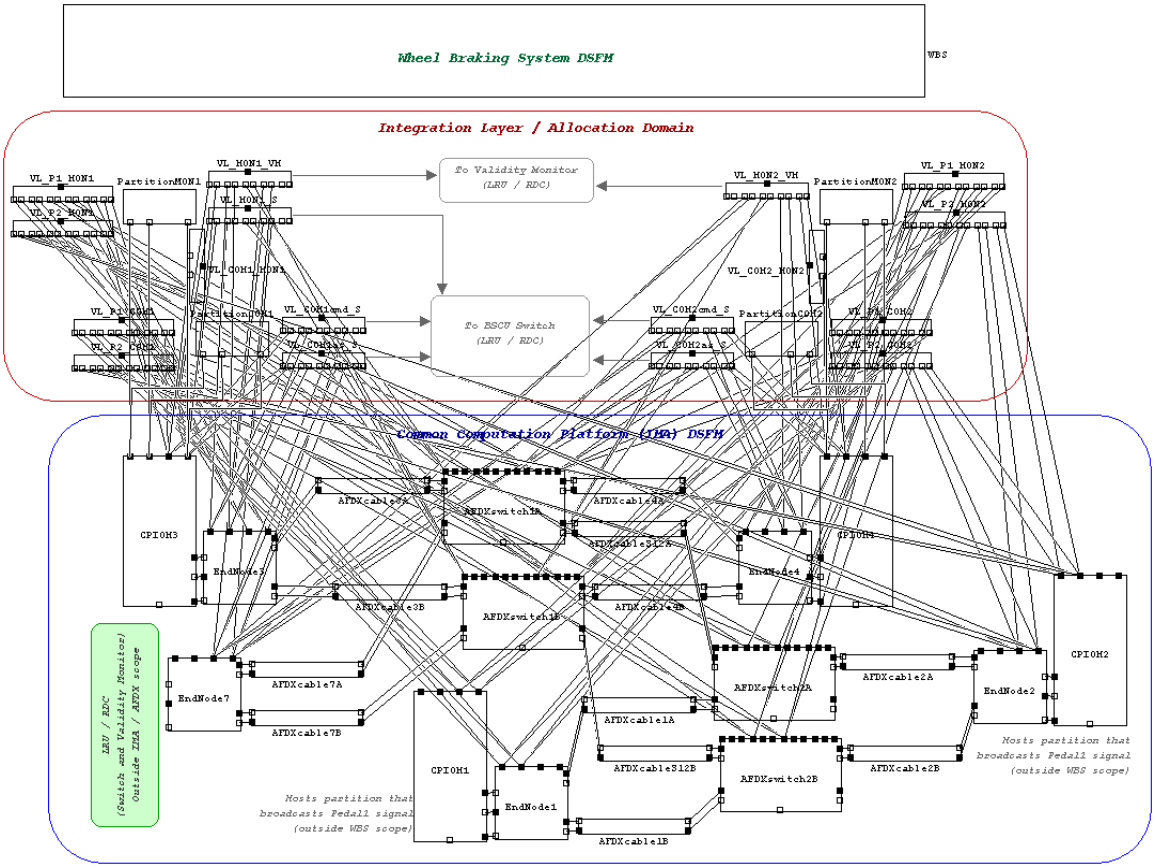


Figure 64 - Composed Failure Logic Model (Automatically Generated Flows)

As well as liberating safety engineers from the repetitive, mundane and, thus, error-prone aspects of DSFM integration, such an automation allows for the rapid adaptation of the overall failure logic model to different alternative allocations and for the 'porting' of the WBS DSFM (along with the translation layer) to different models' IMA DSFMs in a time-efficient manner. It is important to stress that the generation procedure can be easily adapted to any computer-readable format of partition and VL allocations.

4.6.4 Model Analysis

The integrated Failure Logic Model can be simulated and analysed by the Cecilia OCAS as a single whole. Such analysis for example reveals a new single point of failure – *CPIOM3.CPUboard.PartialFailure* – for the *inadvertent application of brakes* failure condition of the WBS. The CPIOM failure causes a *Stuck* failure mode of the *Computing* function of the *CPIOM*; this propagates through the translation layer to simultaneously cause *CMDprocessStuck* and *ASprocessStuck* failures of the *BSCU1 COM* component and a *ProcessStuck* failure of the side's *MON* component leading to both commission of the braking command and an inability of the monitor to report corruption of the primary controller.

The analysis results are not presented here due to their size: analysis of a single WBS failure condition of *inadvertent braking* yields 62 minimal cut sets containing two events and 1270 – for 3 events. Whilst these results are apparently unmanageable, they are not intended to be reviewed directly. Instead, the analysis results should be post-processed and considered incrementally with respect to the results generated from a single WBS DSFM. This process 'filters out' most of the cut sets only showing those which demonstrate new features of failure behaviour not previously seen in a single-DSFM analysis (thus dramatically reducing the number of cut sets that require consideration).

The filtering process is based on two concepts defined by the author: a refinement relation between sets of minimal cut sets and a mapping between failures (i.e. elements of MCSes) which establishes equivalence between failures for the purpose of comparison. For example, for the purpose of comparison all failures of BSCU1 COM component are mapped to all failures of CPIOM3 since it is expected that a failure of the COM can be caused by a failure of the corresponding CPIOM and safety analysts will typically not be interested in an expansion of the results they have previously seen. By contrast, the comparison process will highlight *CPIOM3.CPUboard.PartialFailure* as a new and previously unconsidered behaviour since it invalidates the previously assumed independence between two COM failures (provided that the analysis of the WBS DSFM contains at least one MCS containing both of these failures).

The comparison process is fully automated whilst the mapping definition is facilitated by a graphical “Mappings Management Tool”; the programs were written by members of the MISSA project [6] based on specifications and pseudo-code provided by the author.

To conclude the case study description, it is important to reiterate that the IMA DSFM used in the case study is intentionally simplified. An interesting extension to the case study therefore could be application of a more principled and systematic approach to IMA assessment and construction of a more detailed model. In particular, the LISA method [123] and Conmy’s approach could be used to refine the failure logic characterisations of the CPIOMs and to provide these complex components with a more meaningful structure in terms of the key hardware and OS components of the platform. A particularly interesting approach would be to consider a module’s hardware and the operating system as two separate domains associated with separate DSFMs. The composition approach presented in this chapter could then be used to compose the overall failure logic model of the CPIOMs which, in turn, can be further composed with ‘subscriber’ systems (as shown in this section).

A similar ‘layered’ approach could be applied to the AFDX network where the OSI Reference Model [72], or similar, could be used, to guide the decomposition of the network into engineering domains.

4.7 Relationship to Common Cause Analysis

As mentioned in Chapter 2, civil aviation safety assessment guidance [140, 139] mandates that Common Cause Analyses (CCA) be performed at all levels of design process (forming a thread of activities run in parallel to the ‘core’ of FHA, PSSA and SSA). This section describes how the composition of failure logic models as described in this chapter relates to the existing practice of CCA.

CCA is not a single activity but rather denotes a broad (and, to some extent, eclectic) range of individual analysis tasks whose goal is to identify common dependencies of the different systems, sub-systems or equipment which undermine independence assumptions implied by the system-wise decompositional core safety assessment process. CCA is typically divided into three broad subgroups of analyses:

- *Common Mode Analysis* (CMA) which focuses on the common dependencies of seemingly independent components that arise from the design and maintenance processes;
- *Zonal Safety Analysis* (ZSA) which addresses potential unintended interactions between different systems, subsystems and equipment due to commonalities of their relationship with aircraft structure (e.g. installation and spatial orientation);

- *Particular Risk Analysis* (PRA) which investigates the impact of known threats associated either with the platform environment (e.g. lightning) or with the implementation technology of the systems (e.g. hydraulic fluid leaks, electric short circuits and aircraft tyre burst events).

The goal of the CCA is to identify conditions which undermine the assumed independence of failures in the safety assessment and to assess the impact of these conditions. In failure logic modelling terms, CCA identifies and records external causes of seemingly spontaneous and independent failures of DSFM components. However, when integrated with the primary safety models (whether Fault Trees, Markov Chains or failure logic models) these external causes are typically assumed to be themselves spontaneous and elementary; effectively they are failures external to the DSFM (see Figure 65).

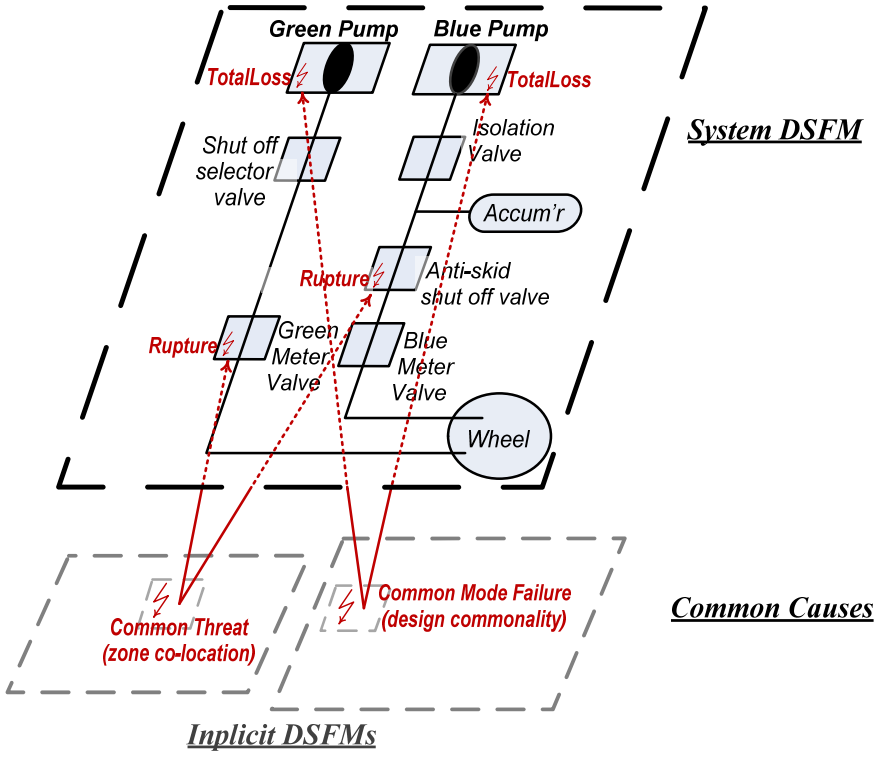


Figure 65 - CCA in DSFM Context

Clearly, the identification and representation of common cause events in failure logic models can be seen as a special case of composition of two DSFMs related to different semantic levels of the platform. The only two differences are that:

a) One of the DSFMs (the common cause DSFM) is exceptionally simple: it consists of a collection of unconnected components (or even just a single component) with failures being predicated on guards that always evaluate to true, and

b) All of the causal relationships between DSFMs are in the same direction: in particular no failures in the common cause DSFM can have external causes in the same DSFM as their effects.

Nevertheless, in its traditional form, common cause analysis can still add value alongside the safety assessment based on failure logic modelling. Simplification of the 'common cause DSFM' into a collection of independent failures enables engineers responsible for a particular engineering domain to consider relationships with other domains before the respective detailed DSFMs become available for integration. This facilitates more accurate safety assessment at earlier stages of design and provides an opportunity to assess whether the DSFM is likely to be composable. Furthermore, for some domains this simplification may be sufficiently accurate even at the later stages of the safety assessment. Finally, the failure logic modelling approach may be inappropriate for some of the engineering domains (e.g. domains that analyse continuous structures and the spread of physical phenomena – such as mechanical stress – and consequently cannot be accurately represented as collections of discrete components and connectors); CCA provides a means of integrating analyses of such domains with DSFMs of other domains through a 'proxy' of common cause events.

One example of CCA being integrated with model-based system safety assessment is analysis of the impact of particular risks carried out by the CCA theme of ISAAC project [5, 9]. In case studies carried out by the project, aircraft equipment installation models in CATIA[41] were used to identify components lying on particular trajectories of the products (shrapnel) of an uncontained engine disc or tyre burst. In safety assessment models (captured in SCADE, StateMate or AltaRica OCAS) each trajectory was represented as a common cause event; the effect of each such event was the failures (total loss) of all equipment identified for the respective trajectory.

The goal of the ISAAC work, however, was not to extend or improve safety assessment methodology but rather to investigate whether current processes can be 'migrated' into a model-based framework. Whilst it appears that, for the engine disc burst scenario, defining an explicit DSFM for engine events and/or trajectories would not add value to the analysis (and common cause events provided an effective proxy for reflecting *results* of geometrical analysis), the same does not necessarily hold for the tyre burst case. The likelihood of tyre burst can be increased by certain failures of the aircraft wheel braking system (e.g. failures that result in overly hard braking). Furthermore, in the context of such failures the independence of tyre burst events from different aircraft gear can no longer be assumed. The ISAAC approach to CCA does not permit

[41] 3D modelling suite developed and marketed by Dassault Systèmes (France): http://www.3ds.com/products/catia/

the capture of potential dependencies between different tyre burst events, or their individual dependence on events in the “functional world”.

Furthermore, the ISAAC approach to CCA is not a restricted instantiation of the approach presented in this thesis. Under the former, common cause events are considered to be part of the *same* safety model as their effect. In the approach presented here, however, a subtly different view is advocated: common causes (even when not arising from complex interactions captured in DSFMs) should be considered as external *to the DSFM* of their effects. This separation highlights the difference between the analyses and the engineering information used for DSFM definition on the one hand and those used for integration of DSFMs and formalisation of common cause analyses on the other. This, in turn, indicates that different aspects of composed failure logic models are the responsibility of different engineering organisations and/or should be validated through different processes.

Finally, it is interesting to note that the view of platform (aircraft) decomposition implicitly adopted by the ISAAC CCA theme is consistent with the domain-based framework described in this chapter. ISAAC CCA divides the aircraft into two broad engineering domains: “geometric” and “functional”. The scopes of these domains overlap, since geometric models contain representations of functional components. The allocation domain is established through equipment naming conventions and unique referencing; in terms of Common Cause Analysis specifically the allocation domain is supported by the mapping manager (developed by ONERA) which translates the geometry domain’s “hit lists” (lists of equipment affected by a particular trajectory) into sets of component failures.

4.8 Model Complexity & Limitations of the Analysis Tool

Whilst the approach to composition presented in this chapter significantly extends all current failure logic modelling techniques as well as the common cause analysis approach developed by the ISAAC project, it highlights a number of challenges in multi-system safety assessment of platforms. The case study presented in section 4.6 has also highlighted some limitations to the approach implementation in AltaRica OCAS.

The challenges and limitations chiefly relate to the analysis complexity of the resultant composed model. For instance the composed case study model can be analysed by the Cecilia OCAS Sequence Generator, to the depth of three events, in approximately 20 hours. Analysis to the depth of four events would sometimes result in memory overflow. Whilst generation of minimal cut sets to the maximum cardinality of three is typically considered pragmatically acceptable, it is important to stress that the current analysis tool does not achieve this within the above periods of

time. Developed as Dassault Aviation's 'in-house' tool and tailored towards features used by the engineers under the (non-FLM) modelling style adopted by the company, Cecilia OCAS counts temporal events towards the size of the Minimal Cut Set. As these events have been extensively used by the author to represent "normal" and "conditioning" events, search to the depth of three events therefore does not guarantee identification of all minimal cut sets of cardinality three (the normal events essentially can 'push' failures out of the cut sets). Sequence generation that guarantees such completeness is, at present, impracticable.

This problem however relates specifically to the analysis tool used. A new sequence generator 'plug-in' is being currently developed by the author's collaborators at Fondazione Bruno Kessler (FBK) to address this limitation.

Further, whilst the analysis time – even to the depth of three events – may appear excessive it can be largely attributed to two factors:

(i) Usage of the 'computationally expensive' modelling constructs to emulate directed weak synchronisation – a fundamental limitation of the chosen implementation language

(ii) A 'brute force', exhaustive simulation, approach to model analysis that underlies existing tools – a feature of the specific analysis tool used by the author

The first factor is specific to the choice of implementation language and its impact can be reduced in other languages whose constructs align better with the structure of the failure logic metamodel presented in this thesis. The second factor is being addressed by the development of the new analysis tool, mentioned above, that is based on NuSMV model checker and the associated NuSMV-SA analysis platform. The tool is expected to optimise the internal model representation and significantly reduce analysis time.

Nevertheless, it is unrealistic to expect that the time required for the analysis of multi-system models (composed from individual DSFMs), will ever be insignificant or that it will be comparable to the analysis of fault trees with the similar number of events. Finally, the time-complexity of the automated analysis also has to be compared not only to the time of analysis of fault trees but also to the efforts of the engineers necessary for re-modelling and integrating of the trees. The efforts necessary for the remodelling task are minimised under the approach presented in this chapter and it has been demonstrated in the course of the case study that significant proportion of the DSFM integration work can be automated.

4.9 Conclusions

This chapter has addressed the issue of composing Failure Logic Models for realistic, large-scale, safety-critical platforms. It was argued that in such a context the assumptions that underlie the composability claims of existing failure logic modelling methods (such as HiP-HOPS) are unlikely to hold. In particular it may be unrealistic to expect models to identify all external dependencies in terms of input failure modes when platform decomposition does not follow a traditional containment hierarchy model. Furthermore, even within the 'pure' containment hierarchy model, it may not be possible to identify Failure Mode interfaces at early stages of design and assessment.

The author has presented a flexible approach to rationalising the decomposition of the platform which combines the principles of containment hierarchy model with the concept of views from the discipline of software architectures and has defined the concepts of Engineering and Allocation Domains. Different patterns of relationships between scopes and viewpoints of engineering domains have been used to identify three key archetypes of allocation domains.

It was shown that whilst, for some of these archetypes, composition of domain-specific failure logic models (DSFMs) may rely on 'hard' interfaces provided by failure modes (albeit that it may require translation between FM class vocabularies used in different models), for others these will not provide sufficiently rich interfaces for composition. The author has demonstrated that in the latter cases failures and failure modes of one DSFM may be manifested as seemingly spontaneous failures in another model. The FLMM has been extended to allow such composition. Based on this, the approach to structuring relationships between different models (based on notions of 'virtual' translation components and a translation layer) has been described and demonstrated using a case study to illustrate its effectiveness.

Principles for the definition of composable DSFMs have been presented and the role of traditional common cause analysis techniques in verifying the composability of the models has been discussed. However, it has been shown that the DSFM composition approach significantly extends traditional CCA and facilitates the representation of circular dependencies between domains that cannot be easily addressed through traditional analysis methods.

Chapter 5: Multi-Mode and Reconfigurable Systems

5.1 Introduction

As was explained in Chapter 3 the key characteristic of failure logic modelling approaches is the requirement to describe the behaviour of the system (and its components) in terms of deviations from the design intent. Overall, with only one exception, discussed separately, all failure logic modelling techniques implicitly assume the following characteristics of the system intent:

a) The intent is coherent and consistent
b) System and component intent is inherently acceptably safe
c) The intent of components can be characterised locally (i.e. in terms of relationship between components inputs, internal abstract state and outputs)
d) The intent of the system (and, by implication, the intents of complex and basic components) is unique and unchanging.

This chapter demonstrates that the last two assumptions do not hold for a large class of industrial systems that can be operated in different modes and/or include functionality for failure detection and reconfiguration (including the controlled degradation of overall functionality). The chapter presents an extension to the FLM Framework that is necessary and sufficient for modelling multi-mode systems and demonstrates how extended metamodel can be implemented in the AltaRica language. Finally, the issues of the compositionality (or, more accurately, the composability) and the reuse of failure logic models is discussed. Highlighted by the 'problem' of modes, these issues are nevertheless present even in static models. In general, characterisation of the component in failure logic models is highly dependent on component's context; this means that in the general case component characterisations are neither reusable nor composable.

5.1.1 Illustration of the Problem Addressed by the Chapter

The problem of non-local dependencies between components in failure logic models has been briefly illustrated in Section 3.8 of the Thesis. Here the problem is illustrated again in more detail on the Wheel Braking System considered in Chapter 3 and in the particular context of system modes.

Analysis of that model for the system-level failure condition of "*Omission of Braking*" will yield, among others, the following two minimal cut sets:

{ BlueMeterValve.JamClosed, BSCU.BSCU1.MON.ProcessTerminated }

{ BlueMeterValve.JamClosed, BSCU.BSCU2.MON.ProcessTerminated }

The presence of the failure of the blue meter valve is obvious – it disables one of the redundant braking channels. The presence of failures of single monitors, however, can be considered as inaccurate: failure of a monitor should not (and does not) lead to loss of the other ("green") braking channel. The problem can be traced to the characterisation of the BSCU *Validity Monitor* (the component that consolidates validity outputs of the two BSCU 'sides' to obtain the overall validity of the control unit). In terms of failure logic characterisation, the *Validity Monitor* currently propagates *FalsePos* FMs from either side as a *FalsePos* FM of the entire BSCU. This *is* accurate when the other side of the BSCU (say, *BSCU1*) fails detectably: in this case the monitor of the failed side will correctly report failure and, thus, an incorrect failure report by the other side (the *FalsePos* FM of *BSCU2*) will result in the whole control unit declaring itself as failed too soon. This declaration will of course result in the incorrect shut-down of the green hydraulic channel (the *Omission* FM). In contrast, if the command module of the *BSCU1* remains failure-free *FalsePos* FM of the *BSCU1* will have no immediate effect on the correct operation of the control unit as a whole and should not be propagated by the *Validity Monitor*.

The problem is that, at the level of the BSCU *Validity Monitor*, the two scenarios are indistinguishable – in both cases *BSCU1.MON* generates no output FM whereas *BSCU2.MON* generates the *FalsePos*. The only difference in the failure logic domain is the failure state and output FM of the *Command* channel of *BSCU1* which is not 'visible' to the *Validity Monitor* component and, thus, cannot be referred to in its failure logic characterisation.

Under this restriction imposed by the present Failure Logic Metamodel (and shared by essentially all current failure logic modelling techniques) the only possible modelling solution is to capture failure logic of the *Validity Monitor* pessimistically, yielding the inaccuracy noted before.

To enable more accurate modelling the metamodel must be extended to allow capturing the fact that at any point of time the BSCU as a whole can be in different modes depending of whether and which of its channels are no longer trustworthy. Section 5.3.2, below, return to this problem to demonstrate the modelling solution in the context of metamodel extensions introduced in the present chapter.

5.2 System Modes

Many industrial systems are designed to perform multiple alternative and mutually exclusive functions throughout their operational life. Examples of such systems can be found in the process industry where plants can be configured to produce different chemicals in different modes of operation or where maintenance and cleaning modes are often defined. Similarly, in the aerospace

domain most aircraft systems operate differently in different phases of flight (such as taxi, take-off, climb, cruise, descent and landing). In such systems, behaviour of components which is intentional and desirable in one mode of operation may be unintended and even unsafe in another mode.

> ***Example:*** *The most famous example of such unsafe behaviour is the Therac-25 radiography machine [90]. The machine was designed to operate in two distinct modes (determined by the operator). Errors in the controller logic allowed in some cases for the system hardware (that determined intensity of the treatment) to be operated in different mode from that of the controller (that determined the duration of treatment). Whilst the behaviour of the hardware had been consistent with its configuration (and its implicit 'local' mode of operation) it was inconsistent with the mode of controller as well as the mode required by the operator. The result was patients' overexposure to radiation (which in some cases has been determined to be the primary cause of death).*

The types of modes that can be found in industrial systems include:

- *Alternative modes of operation*: the system or the equipment can be operated in a number of alternative modes, selected by the operator as appropriate. The sequencing of modes is relatively unrestricted. Examples of these modes may include process plant and Therac-25 modes of operation described above.
- *Phases of operation*: the system operates through a predefined sequence of operational modes. In normal circumstances the operator has a relatively limited influence on the current mode (although they may sometimes influence the duration of each mode and/or have a choice between some predefined alternative modes). Almost all aircraft systems are operated in a phased manner.
- *Reconfiguration and failure mitigation modes*: some system designs have sufficient redundancy provision to be able to perform required functions even in presence of some (diagnosed) failures. Upon detection of significant failures the system will reconfigure to deliver functionality through alternative means. The switch-over between the failure reconfiguration modes may range from appearing seamless to the operator (e.g. many modern aircraft systems) to being facilitated by the operator.

In practice, the three classes of modes above are not mutually exclusive. For example, upon detection of failure some of the modes of operation may be disabled, phase sequence can be simplified (e.g. restricting choice of available alternatives or inhibiting certain non-essential phases altogether) or a special shut-down phase sequence can be initiated.

Returning to failure logic modelling, multi-mode systems clearly pose a significant challenge to the naïve methodology presented in Chapter 3. The challenge is two-fold:

- Multi-modal systems are associated with a number of alternative intents. This means that there is no single reference of 'nominal behaviour' against which failure modes (i.e. deviations) can be defined.
- As transition between system modes may be determined by system inputs, global events or failures of individual components, the correct mode cannot necessarily be established by failure logic characterisations of basic components based on the information available to that component (i.e. component inputs and states). As a result, basic components are incapable of interpreting input failure modes and "ok" privatives since those have ambiguous semantics unless a particular system mode is somehow indicated to the component.

5.2.1 FLMM Extension

To address these challenges the Failure Logic Metamodel must be expanded firstly to enable representation of the mode and transition between modes at the level of any complex component (including the system), and secondly to ensure that all constituent components have access to the mode of any 'ancestor' and can refer to it in their propagation equations and state entry logics (if basic components).

Informally, as discussed above, a mode is an indicator of a current intent of a system (or a complex component) and the corresponding intents of all constituent basic components. In other words, modes can be seen as a facility of decomposing an otherwise complex system intent into more simple 'regions' (predicated on some conditions) which yield unique intents of system components (and thus provide unique interpretations to the input and output failure modes of the components).

Since modes are typically persistent and can be predicated on other modes[42] they can naturally be represented in the FLMM as a state of a complex component. Therefore, the concept of 'mode' bears some similarity with the various concepts of a state (e.g. failure state or failure handling state) of a basic component introduced in Chapter 3. As with the states of basic components, modes are grouped into state-spaces; each mode also has to be associated with the "transition" (Figure 66).

[42] For instance, in case of sequences of phases where each phase is predicated, among other conditions, on the preceding phase of the system.

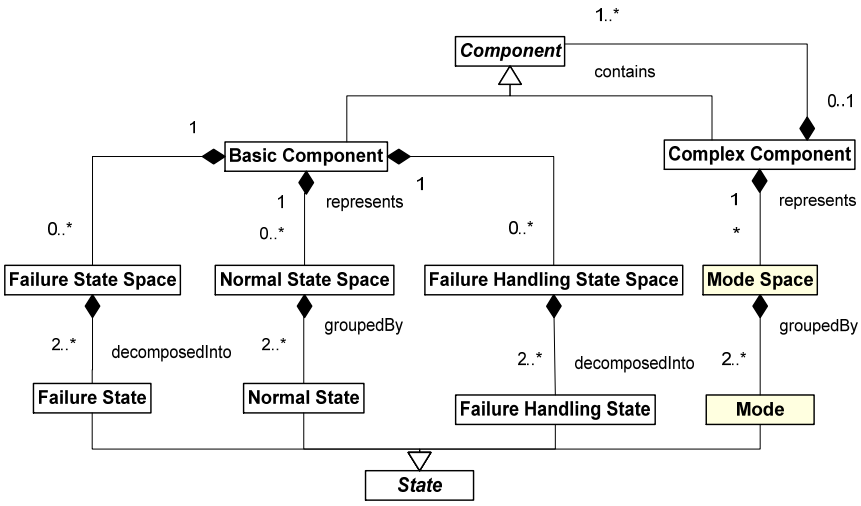


Figure 66- Revised FLMM: Modes of Complex Components

Containing a guard and a trigger, mode transitions can refer to other modes of the complex component, the component's input failure modes and any failures or normal events of the constituent sub-components. The latter means that failures and normal events of basic components are visible to all of a component's 'ancestors' up to the level of the system[43].

However, it is important to note that whilst transitions into the reconfiguration and failure handling modes of a complex component are often fully determined by the preceding mode of the component and events local to sub-components, the same does not hold for phases and alternate modes of operation. Transitions between these modes are apparently non-deterministic from the perspective of the failure logic of the system[44]. In order to resolve mode non-determinism it is therefore necessary to allow complex components to contain normal events of their own. Typically defined at the top level of system, these normal events are only used for transitions between modes defined at the level of the same complex component. This requires a further straightforward modification of the FLMM (Figure 67).

In terms of semantics, the modes of complex components are also similar to the states of basic components: both are abstract representations of some conditions with the granularity established specifically for the purpose of the model and based solely on the engineer's judgement. It is important to stress, however, that for modes the extent of the abstraction from physical states observed in the system can be even greater than for failure and normal states. Reconfiguration of the actual system from one mode to another is often achieved in a distributed manner with a

[43] Note that this is consistent with the discussion in the previous chapter that has established that components events form part not only of a component interface but also of the interface of sub-systems and systems they are contained within. Further, note that the states of basic components – being a modelling convention and an abstraction – remain fully hidden within component's boundaries.

[44] For example, whether or not the Therac-25 machine should have been operated in photon or electron therapy mode cannot be decided based on failures of the machine's components or deviations of user inputs.

number of controllers reconfiguring hardware (e.g. electrical switches or hydraulic selector valves) based on the locally-defined rules and locally observed conditions. The notion of "mode" in such context can be seen as an emergent feature of the system (with respect to the logic of the distributed controllers). Under the FLM Framework it is the responsibility of the safety engineer to reconcile behaviour of individual controllers and to rationalise it into a set of 'holistic' modes. Consequently, sufficiently complex systems may yield a number of possible alternative (and correct) mode characterisations.

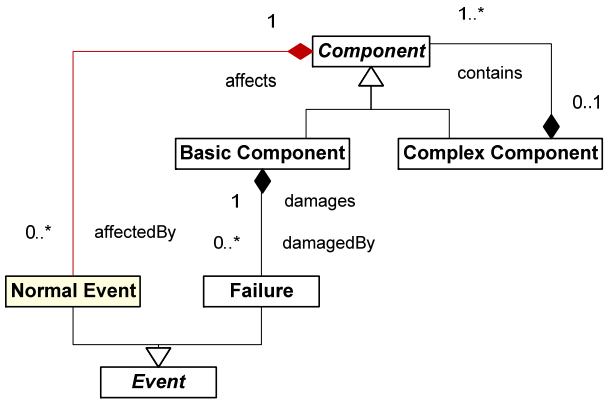


Figure 67 - Revised FLMM: Normal Events of Complex Components

At the same time, the engineering semantics of the modes is also significantly different from that of basic components' states: unlike the latter, modes neither represent a physical state of the component nor determine components' behaviour; instead modes capture (or indicate) what the behaviour *should be*.

To summarise, this thesis defines a complex component mode (or just 'mode') as follows:

Complex Component Mode:

An abstract persistent condition of a complex component that:

- *Determines the intended function of the complex component and the intended behaviour of all sub-components it contains*
- *Associates all failure mode and privative flows within the complex component with unique interpretation*
- *Is wholly determined by the previous mode of the component, modes of the higher-level complex components (the component belongs to) as well as failures and normal events of the sub-components (that the component contains)*

The complete revised Failure Logic Metamodel is shown in Figure 68. The model is supplemented with a number of constraints guarding the 'visibility rules' (discussed throughout this section) of various events and states:

(i) Transition triggers associated with non-mode states of any (basic) component can only refer to the events declared within the same component (or, alternatively, can be left "void")

(ii) Transition triggers associated with modes of any (complex) component can only refer to the normal events declared within the same complex component or to failures and normal events declared within basic components that are themselves declared within the domain of the complex component (or, alternatively, can be left "void")

(iii) State Proposition declared within a component (as part of either a transition guard or a propagation condition) can only refer to the state of the same component or the state (by implication – mode) of its ancestors

(iv) FM Proposition declared within a component (as part of either a transition guard or a propagation condition) can only refer to the Input FM of the same component.

The extended metamodel shows that the three broad classes of modes, identified in the beginning of this section, are fundamentally similar. Each different class is merely characterised by particular *prevailing* patterns in the respective mode-space:

- *Alternative modes of operations* are typically associated with transitions which are not predicated on another mode and are usually triggered by normal events declared at the level of the same complex component (typically – a system);
- *Phases of operation* transitions are typically guarded by a small number of modes, typically, with only one –or a very small number of– transition(s) leaving every mode (i.e. showing nearly a 'pipeline pattern' of mode transitions). The transitions are typically triggered by normal events declared either at the level of the same complex component (e.g. representing abstraction of time) or at the level of constituent components (e.g. representing exhaustion of some finite resource such as a fuel tank, a hydraulic accumulator or an electrical battery).
- *Reconfiguration and failure mitigation modes* are typically triggered by failures of basic components or by input failure modes (via a void trigger). The models of these modes and their transitions tend to form directed acyclic graphs (DAGs).

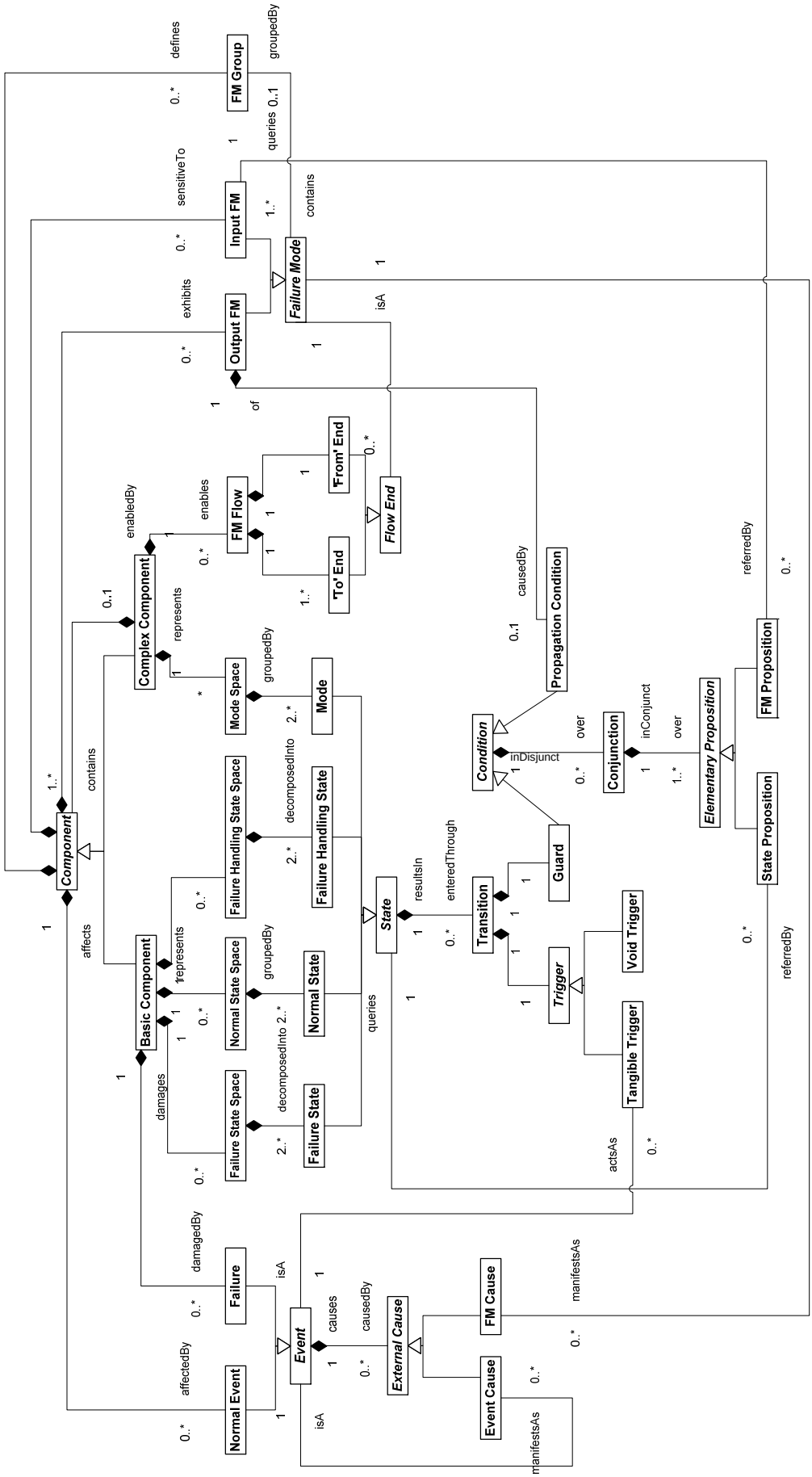


Figure 68 - Complete Revised Failure Logic Metamodel (FLMM)

5.2.2 Modes as Component Context

Complex components' modes can be seen to provide basic components with (a reference to) the necessary contextual information that influences a component's failure behaviour. This section considers various implications of such a contextual dependency on the properties of basic components, the concept of modes and even properties of the failure logic modelling approach as a whole.

5.2.2.1 Component Failures and Dynamic Exposure Intervals

The influence of the system mode on the behaviour of basic components is not limited to the interpretation of input failure modes (or the lack thereof) and, thus, propagation conditions of the component that have been discussed so far. The expanded FLMM permits component state transitions to be similarly context-dependent.

> ***Example:*** *certain failures of components may be more or less likely depending on how a component is being used at the time: electrical fuses, circuit breakers and hydraulic valves used in 'stand-by' redundant channels of the system may be more likely to fail in modes when the respective lines are energised. The Failure Logic Metamodel permits us to reflect this aspect of the component failure logic by predicating failure state transitions on the mode of the system. In other modes, failure state transitions of 'cold' spares may be prohibited; for 'warm' spares only transitions triggered by 'special' – less likely – failures may be permitted. It is important to note, however, that spares may also be energised inadvertently in modes when they are not intended to be 'active'. Consequently, transition guards would rarely be predicated on a mode alone and would typically contain a disjunction over modes and commission failure modes of the activation signal.*

Considering the reverse of a typical cold spare scenario leads to an important observation about the nature of modes in failure logic models. Consider a situation where a certain failure is only possible in a certain mode of operation when the component is *not* actively utilised by the system. Whilst such a failure may not have an immediate effect on the component's interface, its local effect will be 'stored' in a failure state of the component. The latter may cause an output FM of a component once the system moves to another mode where the component is active (even though the original failure may be impossible in this mode). This simple hypothetical example shows that, whilst modes partition the intent of the system, they cannot be generally seen as partitioning the failure logic model itself into a number of independent more simple 'mode-free' models:

Failure behaviour of a system in one mode may be influenced by any permanent damage sustained in preceding modes.

At the same time, *the concept of modes enables us to capture dynamic and conditional exposure intervals of individual failures* (associated with a particular probability distribution function) more accurately in the FLM Framework. Of course, in some circumstances conditional exposure intervals could be captured even without using modes by including input FMs in the guards of failure state transitions (see Chapter 3). However, this only allows limited exposure to failure to circumstances when the component is exposed to an input deviation (such as inadvertent activation of a component). In the absence of input failure modes, there is no mechanism for segregating situations when component is active and when it isn't. This yields the next observation about modes:

Whilst modes are necessary to enable the correct interpretation of input failure modes by individual components, they are also essential for interpreting the absence of input FMs.

5.2.2.2 Reusability of Component Characterisations

For multi-modal systems, the context dependency of components clearly limits the reusability of component models (characterisations). Indeed, for a system consisting of *n* redundant and identical cold standby subsystems ("channels") and, thus, potentially operated in *n* modes, models of seemingly identical components in different channels will not be identical.

It is important to stress that the problem of context dependency of components in failure logic models is not specific to multi-modal systems. Multi-modal systems merely *highlight* this problem since in these systems the context is dynamic and thus needs to be explicitly represented. For fully static systems the context still exists, it is still non-local and models of basic components are still sensitive to it. However, static context remains implicit in failure logic models and, thus, the dependency is somewhat less obvious. Nevertheless it can be illustrated with the WBS example used in Chapter 3.

The system contains two redundant hydraulic channels each containing one meter valve. The design and functionality of these two valves are nearly identical[45]. In these circumstances a natural expectation is that the same failure characterisation could be reused for both components. In practice, however, the characterisation of the two valves in the failure logic model is significantly different (Figure 69). The blue valve would generate a commission of braking as a result of inadvertent command with no need for any hydraulic input failure modes. In contrast, the shut-off selector valve of the green channel 'protects' the green meter valve from being exposed

[45] Trivial modifications to system description (such as stipulating electronic rather than mechanical link between cockpit pedal position and blue meter valve) will remove any differences whilst maintaining the failure characterisation of the components.

to a commission of control: this meter valve will only exhibit a commission failure mode if it is simultaneously exposed to failure modes on both control and hydraulic inputs.

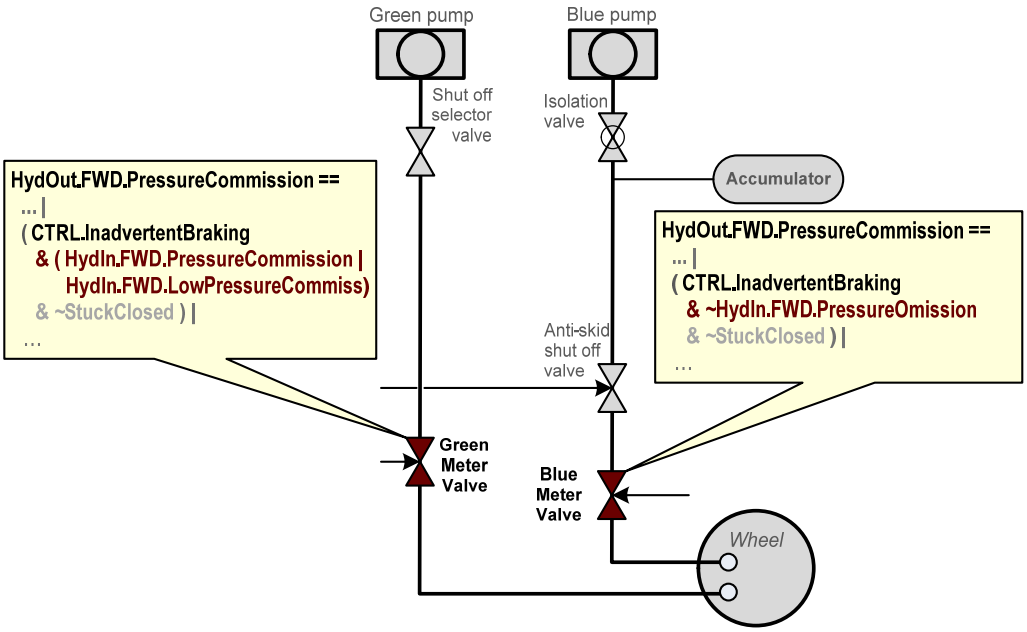


Figure 69 - Difference Between Failure Logic of Green and Blue Meter Valves

Furthermore, if the design of the blue channel were to be modified and a single meter valve was to be replaced with two identical valves connected in series and driven by the same mechanical pedal position, the characterisations of these two valves would not be identical: the downstream valve will be protected by the upstream 'sibling' and so will *only* exhibit a commission failure mode if it is exposed to a commission of hydraulic pressure (along with suffering from an internal failure or being further exposed to control failure mode); the upstream valve would *not* benefit from such protection and could exhibit commission failure mode as a result of internal malfunction alone. It is important to stress that this scenario will hold in purely 'static' (mode free) models, and regardless of the specific modelling notations or techniques. The scenario demonstrates a significant limitation to the validity of any claims that (in the context of failure logic modelling approaches) "reuse of 'component safety analyses' across applications becomes possible" [162]. In fact, *component characterisations can only be reused when (sufficiently) identical components are employed in (sufficiently) identical context*; regardless of whether that context is static or dynamic.

Based on experience in the case studies, section 5.5.3 suggests that a partial solution to the reusability of failure characterisations may be possible within the boundaries of individual systems (or, possibly, in product families). However, as far as the author knows, no work on such extension to any failure logic modelling technique has been carried out to date and the proposal remains not fully validated.

5.2.2.3 Context Boundaries and Dysfunctional Modes

Previous discussions have demonstrated that modes are sometimes needed to provide basic components with contextual information that is necessary for the interpretation of failure modes and, often more importantly, the absence of any deviations from the intent. At the same time the FLMM has stipulated that modes are declared at the level of particular complex components which limits their domain (boundaries of visibility). So what happens when a failure mode flow crosses the boundary of a mode domain that essentially determines its semantics?

In general, two solutions can be taken to ensuring that FMs can still be interpreted outside the 'source' complex component:

- The context (i.e. component mode) can somehow be communicated to the environment
- A complex component's output failure modes (and privatives) can be translated to the failure modes with identical semantics in the new context.

The former approach clearly requires us to allow components in failure logic models to communicate through means other than failure modes; it widens the component interface and weakens information hiding in the model. Consequently, this thesis favours the latter approach which is consistent with discussions on model composition in Chapter 4.

To demonstrate both the problem and modelling solution in concrete terms, a new (fourth) class of system modes – *dysfunctional modes* – is discussed. Dysfunctional modes identify circumstances when the design of the system (or complex component) *deliberately* does not comply with the full intent (in a wider context). In general two sub-classes of such modes can be identified:

- *Design limitations:* whereby in some well-defined circumstances incorrect operation of the system is deemed acceptable in order to reduce complexity or improve performance
- *Degraded modes:* when following a number of failures some of the less essential functionality of the system is 'sacrificed' in order to maintain more critical aspects of system operation and/or to reduce further threats to the limited remaining resource.

In practice, dysfunctional modes are almost always a result of engineering trade-offs. Indeed, whilst system safety is one of the key design drivers it is not the only driver. In practical designs it is often necessary to trade safety characteristics off against other quality attributes of the system (such as weight, availability, modifiability or supportability). Even *within* the safety attribute itself, trade-offs are often necessary between the availability of a safety-critical function and other considerations that may impact safety of the system (such as the complexity of the system that may adversely affect the engineers' ability to validate and verify design correctness).

In terms of failure logic models, dysfunctional modes are typically triggered by failures of basic components (indicating a need for 'shedding' non-essential functionality or exposing a limitation of a failure detection or monitoring scheme) or normal events declared at the level of a complex component (exposing a limitation of design in some circumstances)

> ***Example:*** *An aircraft fuel system may be required to provide annunciation of abnormal fuel distribution or leaks. This functionality, however, is unreliable (and may be inhibited) when the aircraft performs certain manoeuvres (e.g. steep climb or descent) which, whilst 'normal', are judged to be sufficiently infrequent and not long-lasting. Similarly, certain cockpit announcements and warnings are typically intentionally inhibited during the last stages of landing or after a certain speed is reached in take-off.*

As was mentioned above, when failure mode flows cross the boundaries of domains of dysfunctional modes they may need to be translated into appropriate FMs in the new context. The principled approach to such translation – that is consistent with the approach to multi-system composition described in the previous chapter – requires the introduction of a dedicated virtual "translator component". However, since this approach increases the number of model components[46] it can sometimes be perceived as 'cluttering' the failure logic models. In a practical setting, it may sometimes be possible to 'absorb' the logic of the translation component in the failure logic characterisation of the components closest to the interface of the mode domain.

The model of the Braking System Control Unit (BSCU) of the WBS, used in previous chapters, can be used to illustrate the solution. The BSCU contains two redundant sides (each consisting of the Command and Monitor modules) as well as a Validity Monitor and a Switch (see Figure 70). In all previously presented models it was assumed that the Switch component cannot fail. This assumption is now revoked.

In particular it is now assumed that the switch can suffer from two hypothetical failures: one – *ProcessStuck* – ultimately results in the switch's becoming insensitive to the validity input; the other – *ProcessTerminated* – prevents the switch from propagating any braking or anti-skid commands (i.e. resulting in *LackOfBraking* and *AntiSkidOmiss* output failure modes over *CMD* and *AS* outputs of the switch respectively).

The switch failures clearly affect the 'trustworthiness' of the command and anti-skid outputs of the BSCU as a whole. Furthermore, in this scenario the Validity Monitor will clearly be unable to detect and duly report that the main outputs are invalid. In other words, failures of the switch expose fundamental limitations of the BSCU design in general and the specification of the

[46] And, in particular, the number of model components that cannot be traced to any elements of system design.

monitoring schema in particular. This limitation cannot be attributed to any individual basic component – it is a limitation of the design of the complex BSCU component as a whole.

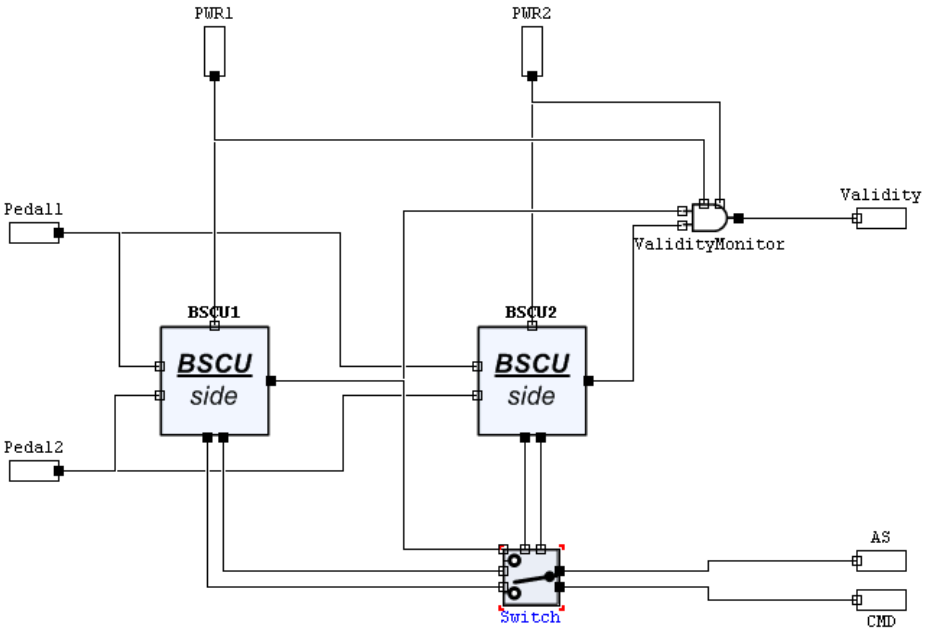


Figure 70 - BSCU Architecture

Generally, there are two approaches that could be taken to correctly reflect the limitation in the failure logic model of the BSCU. On the one hand, it can be observed that the limitation effectively establishes a dependency between the Switch and Validity Monitor components in terms of an assumption that the switch will never fail. This assumption can be considered to be part of the intent and, thus, a deviation from the assumption (i.e. failure of the switch) can be captured in terms of FM flow between the two components. However, this FM flow will have an unintuitive interpretation in terms of BSCU design (since the two components do not interact in any tangible fashion) and, more importantly, this solution is not scalable.

On the other hand, the more principled solution (described above) is to recognise that the known limitation of the monitoring schema means that BSCU can be operated in two scenarios (i.e. modes): normal operations (when validity monitoring is effective) and a dysfunctional mode (when the design limitation is exposed and the monitoring is ineffective). Further, the conditions upon which the BSCU moves from *Effective* to *Ineffective* mode are known[47] – failures of the Switch component (Figure 71).

[47] It is interesting to note that the knowledge of these conditions along with clear understanding of limitations (and, presumably, justification of why the limitation is acceptable) is the difference between dysfunctional modes and systematic failures of complex components. Indeed, the same approach could be used for modelling hypothetical systematic failures of complex components (i.e. design errors not attributable to individual basic components).

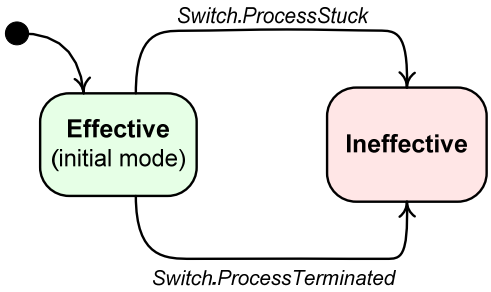


Figure 71 - BSCU Modes and Transitions

These two 'monitoring modes' allow us to model the contextual dependency between (the semantics of) validity monitor output failure modes and failures of the validity switch without establishing an explicit flow between two components.

The reinterpretation of the failure modes associated with the BSCU validity output is performed through a dedicated virtual translator component inserted between the Validity Monitor and the BSCU interface (Figure 72). The translator component propagates both FMs (*FalsePos* and *FalseNeg*) unchanged in both modes; however, in the *Ineffective* mode the absence of *FalsePos* is translated into a *FalseNeg* output FM (Figure 73).

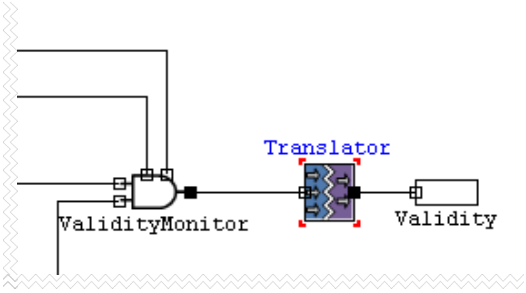


Figure 72 - Revised Failure Logic Model Architecture of BSCU (partial)

Translator (virtual) [in BSCU]			
Input Failure Modes			
Failure Mode (ID)		**FM Class**	**FM Group**
In.FalsePos		Omission	In
In.FalseNeg		Commission	In
Output Failure Modes & Propagation Conditions			
Failure Mode (ID)	**FM Class**	**FM Group**	**Propagation Condition**
FalsePos	Omission	Validity	In.FalsePos
FalseNeg	Commission	Validity	*BSCU.Ineffective* **&** ~In.FalsePos | In.FalseNeg

Figure 73 - Characterisation of the Virtual Translator Component

5.3 Implementation of Modes in AltaRica/OCAS

Originally the selection of the model specification language was influenced, among other factors, by the claims of the compositionality of the various failure logic methods (e.g. see [162] for HiP-HOPS). As a result, the selected language (AltaRica Dataflow/OCAS) is strictly compositional, in that complex components ("equipment") may only contain sub-components, assertions over interfaces (input and output flows) of the complex components (and sub-components) as well as synchronisations of sub-components' events. In particular, *equipment nodes may contain neither states nor events of their own*. Whilst it is still possible to implement the Failure Logic Metamodel in AltaRica, these language restrictions mean that the implementation has to rely on more complex language constructs.

Essentially, for each complex component with modes it is necessary to construct one or more components to hold mode model(s). Furthermore, in order to make failures and normal events 'visible' to the mode observer and, in reverse, to make modes visible to the basic components a number of synchronisations have to be explicitly established. Finally, the code of individual basic components that are either sensitive to modes or influence mode transitions also needs to be modified.

Overall, whilst modes introduce significant additional complexity to the models and the task of codifying the models, the process is systematic and algorithmic with the overall procedure outlined below.

5.3.1 Procedure for Encoding Modes in AltaRica

The procedure for encoding modes in AltaRica OCAS consists of three steps:

(1) The mode model of the components is constructed in a dedicated (virtual) mode observer component.
(2) Models of all basic components that trigger mode transitions are modified and synchronisations are established between the basic components and mode observer.
(3) Models of all basic components that are sensitive to modes are modified and appropriate synchronisations are established between these components and the mode observer.

In the following description it is assumed that, before the procedure commences, models of basic components already include all failure and normal state transitions. In practice, the codification process is likely to be iterative.

Step 1: Construction of the mode observer

For each mode-space in each Complex Component a special virtual basic component[48] is declared (the "Mode Observer"). The component has two state variables:

- One (typically called "*Mode*") defined over an enumerated type with each enumeration symbol representing a distinct mode (i.e. an implementation of the State metaclass that belongs to the *Modes* metaclass in the FLMM).
- The other – typically called *BroadcastPending* – defined as a Boolean with initial value *false*.

For each input FM of the complex component that features in the mode transitions guards, an input port of the observer is declared (and linked to the appropriate input of the complex component).

For each void trigger of a mode transition a separate event is declared in the observer and assigned the *Dirac(0)* law.

For each non-void *mode trigger* an event is defined within the observer (typically named after the component and event – Failure or Normal – that the trigger should be associated with) and for each mode itself a further instantaneous event is defined (typically called "*Broadcast<XXX>*", where *<XXX>* is replaced by a mode identifier).

The mode observer has no assertions (i.e. no *assert* clause) but has two sets of transitions. First, mode transitions are constructed as appropriate with reference to the observer inputs, void triggers and mode triggers defined above. Each mode transition not only assigns an appropriate value to *Mode* variable but also 'turns' *BroadcastPending* to *true.*

Second, for each value of *Mode* a transition is defined. The transition is predicated on this mode and *BroadcastPending* variable being *true* and is triggered by the corresponding 'broadcast event'; the effect of the transition is the assignment of *false* to the *BroadcastPending* variable.

Step 2: 'Binding' of mode triggers

At the level of the complex component which contains a mode observer, a synchronisation is defined for every non-void (i.e. non-instantaneous) trigger event of the *Mode Observer*. For triggers that relate to the normal events of the complex component itself, the synchronisation is assigned a weak hiding type ("diffusion"). For triggers that relate to events of basic components the synchronisation is assigned a strong type ("synchronization") and an instantaneous – *Dirac(0)* – law. Initially each synchronisation contains only the appropriate (trigger) event of the mode observer.

[48] In a sense that it does not model any component that can be identified in system design.

Next, each component that contains a failure or a normal event that should trigger the mode change (as well as the local state change of the component) is augmented by:

(i) Defining a new Boolean state variable – *ModePending* (with initial value *false*)

(ii) Defining a dedicated instantaneous event for every failure or normal event that should trigger a mode change (typically called "*Trigger*<*XXX*>", where <*XXX*> is replaced by the identifier of the original event.

(iii) Augmenting every existing component state transition that is triggered by such event so that (in addition to the original effect) it assigns the *ModePending* variable a value of *true*

(iv) Declaring a new transition from every state entered through the above transition. This transition is triggered by the appropriate new trigger event (defined in the second step above) and is predicated on both the component state and the *ModePending* being *true*; the transition doesn't affect the original component state itself but changes *ModePending* to *false*.

The result of this change is that whenever a failure or a normal event, *XXX*, which should trigger a mode transition, 'fires' – a corresponding new *Trigger*<*XXX*> event immediately takes place.

Finally, the newly defined basic components' trigger events are included in the appropriate strong synchronisations defined in the beginning of this step.

At the end of this step the mode observer 'sees' all necessary failures and normal events and is capable of executing mode transitions appropriately. However, the mode remains, so far, 'invisible' *to* the basic components.

Step 3: Local modes broadcast

Similarly to the previous step, at the level of the complex component that contains a mode observer a strong instantaneous synchronisation is defined for every broadcast event of the *Mode Observer*. Each synchronisation initially only contains the respective observer's event.

Every basic component whose behaviour is dependent on the mode of the higher-level components must be changed as following:

(i) A new state variable is defined (typically called "*Mode*") with exactly the same enumerated type as the mode variable of the applicable mode observer. For components that are sensitive to more than one orthogonal group of modes – more than one variable may need to be defined.

(ii) For each possible value of this mode (i.e. each enumeration symbol of its enumerated type) a new event is defined (typically called "*Broadcast*<*XXX*>" where <*XXX*> is the value of the mode variable).

(iii) For each such new event a transition of mode should be defined with the guard being simply "*true*" and the effect of the transition being an assignment of the respective value to the *Mode* variable.

Finally, each newly constructed event is added to the appropriate synchronisation of the complex component (defined at the beginning of this step).

At the end of this step each 'mode-aware' component includes a 'replica' of the mode model of the observer. This replica, on the one hand, is accessible and can be used in the behaviour specification of the component; on the other hand it is fully synchronised with the observer and, thus, reflects the mode of the 'ancestor' complex component at any point of time.

5.3.2 Illustration

In this section the previous example of (dysfunctional) BSCU monitoring modes is used to illustrate implementation of modes in the AltaRica OCAS.

First, the mode observer is constructed (Figure 74). As the BSCU mode model contains only two states – *Effective* and *Ineffective* – the observer's *Mode* variable is defined over two enumeration symbols and only one broadcast event (*BroadcastIneffective*) is necessary. The *Ineffective* state can be entered through two transitions – associated with different failures of the *Switch* component – yielding two trigger events (*TriggerSwitchStuck* and *TriggerSwitchTerminated*).

```
node MonitoringModeObserver
  state
    Mode : {Effective,Ineffective} ;
    BroadcastPending : bool ;
  event
    TriggerSwitchStuck, TriggerSwitchTerminated, BroadcastIneffective ;
  init
    Mode := Effective ;
    BroadcastPending := false ;
trans
    // *** Mode transitions ***
    // Mode change in response to triggers
    // Note: that each transition sets BroadcastPending flag
    Mode = Effective |- TriggerSwitchStuck -> Mode := Ineffective, BroadcastPending := true;
    Mode = Effective |- TriggerSwitchTerminated -> Mode := Ineffective, BroadcastPending := true;
    // *** Mode broadcasts ***
    // A broadcast consists of "firing" the instantaneous event
    // After the broadcast the flag is reset to "false"
    BroadcastPending and Mode = Ineffective |- BroadcastIneffective -> BroadcastPending := false;
  extern
    law <event BroadcastIneffective> = Dirac(0) ;
edon
```

Figure 74 - BSCU MonitoringModeObserver in AltaRica OCAS

To communicate failures of the *Switch* to the mode observer the characterisation of the former is modified (Figure 75). Upon failure, the *Switch* 'memorises' that an event needs to be 'sent' to the mode observer by assigning the *ModePending* variable a value of *true*. Consequently, whenever the component is in the *ModePending* state and its *FailSt* is not equal to "*OK*" an appropriate *trigger event* is 'fired' immediately.

```
node Switch
  state
    FailSt : {OK,Stuck1,Lost} ;
    ModePending : bool ;
  event
    ProcessTerminated, ProcessStuck,
    TriggerTerminated, TriggerStuck ;
  init
    FailSt := OK ;
    ModePending := false ;
  trans
    // *** Failure state entry logic ***
    // Note "queing" of trigger commonications through ModePending variable
    FailSt = OK |- ProcessStuck -> FailSt := Stuck1, ModePending := true;
    FailSt = OK |- ProcessTerminated -> FailSt := Lost, ModePending := true;
    // *** Mode trigger broadcast logic ***
    ModePending and FailSt = Stuck1 |- TriggerStuck -> ModePending := false;
    ModePending and FailSt = Lost |- TriggerTerminated -> ModePending := false;
  assert
    // *** Propagation conditions ***
  extern
    law <event TriggerTerminated> = Dirac(0) ;
    law <event TriggerStuck> = Dirac(0) ;
edon
```

Figure 75 - Revised AltaRica OCAS Specification of the Switch Component

```
node Translator
  flow
    In : WBS_ValidityFMs : in ;
    Out : WBS_ValidityFMs : out ;
  state
    Mode : {Effective,Ineffective} ;
  event
    BroadcastIneffective ;
  init
    Mode := Effective ;
  trans
    // "Reading" mode broadcast
    true |- BroadcastIneffective -> Mode := Ineffective;
  assert
    // Reinterpretation of Validity Monitor's
    // failure modes from BSCU context (In flow)
    // to the WBS context (Out flow)
    Out = ( case  {      Mode = Ineffective and In = OK          : FalseNeg,
                  else In})
edon
```

Figure 76 - AltaRica Characterisation of the Virtual Translator Component

Within the BSCU only one basic component is sensitive to the dysfunctional *Ineffective* mode – the *Translator*. Within this component the modes of the *MonitoringModeObserver* are replicated with a single transition triggered by a dedicated *BroadcastIneffective* event (Figure 76 above)

Finally, three synchronisations are defined at the level of the *BSCU*: two for communications of *trigger* events from *Switch* to *MonitoringModeObserver* and one for the mode *broadcast* (from the observer to the translator component). Figure 77 shows all three synchronisations with the mode broadcast shown in detail.

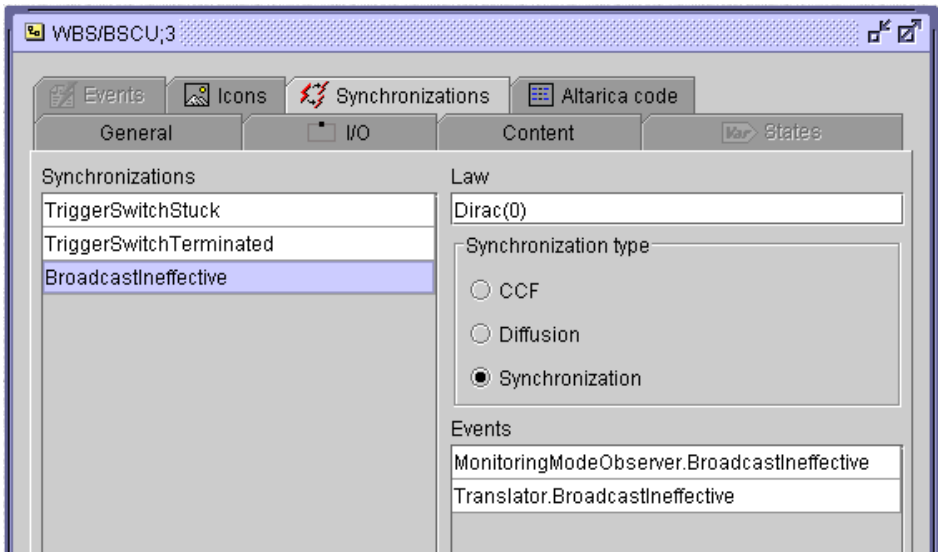


Figure 77 - BSCU Synchronisations

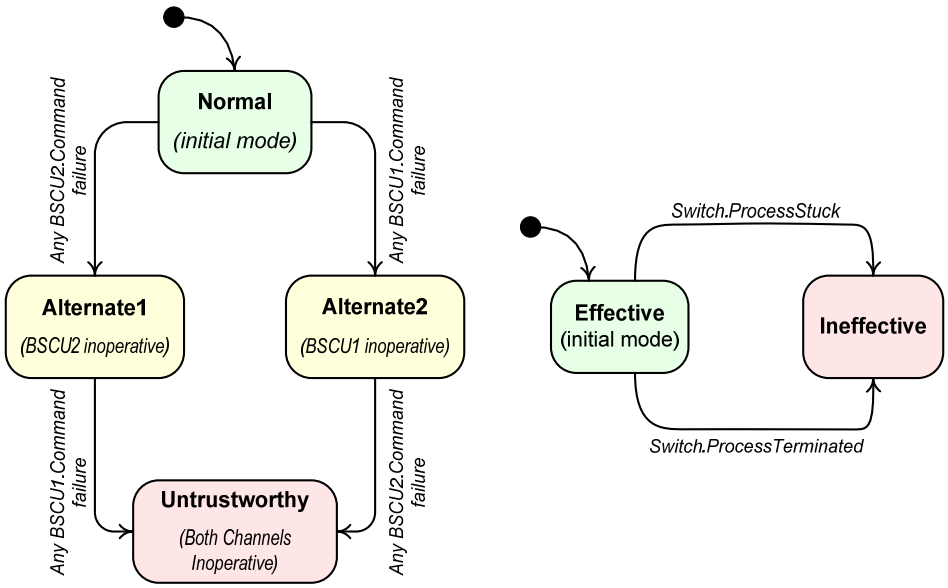


Figure 78 - Revised Architecture of BSCU's Failure Logic Model

Before concluding the illustration it is interesting to reiterate, that, even disregarding Switch failures and the dysfunctional mode of the BSCU, the model of the BSCU presented in Chapter 3 is inaccurate. The problem, previously described in Section 5.1.1, is that without reference to the

real status of two command modules the failure logic of the Validity Monitor can only be characterised pessimistically. In order to improve accuracy of the failure logic model of the BSCU, the new mode model of the unit (orthogonal to the dysfunctional mode model discussed above) has to be constructed to capture the status of both redundant channels at any point in time (Figure 78, above). The complete revised model is included in Appendix B, Section B3.

5.4 Implications for the Modelling Process

Throughout the discussion presented in preceding sections it was demonstrated that in general failure logic models cannot be considered to be compositional:

- The failure behaviour of complex components and systems cannot be fully attributed to the basic components they contain (since modes and some normal events are part of a non-decomposable characterisations of the complex components).
- Individual components cannot be constructed in isolation and then 'brought together' to form a model (since their characterisations are context-dependent and require us to consider components intent in the wider context of the system).
- Even in the context of simple and intuitive component intents and in the absence of dynamic reconfiguration and modes, the degree of effort required for integration of basic components' characterisations into a single model is greatly dependent on whether a consistent interface between components has been established.

As a result, the key challenge to the failure logic modelling *process* is to reduce the impact of non-compositionality of models on the cost and efficiency of the overall safety assessment.

In order to preserve *some* composability of the failure logic models and the compositionality of the modelling process, the models have to be constructed in two analytical stages: a holistic architectural stage and (relatively) compositional detailed component modelling stage. In practice, the two stages are iterative with the detailed modelling stage being capable of identifying necessary revisions in the architecture.

5.4.1 Establishing Model Architecture

The informal goal of the architectural stage is to set the overall modelling principles and conventions for the model. This consists of:

- the identification of components, their hierarchies and intents (based on system design description);
- the elicitation of components failure modes and dependencies (i.e. FM flows);
- the construction of preliminary mode models of all complex components.

The underlying system assessment process is creative and relies on engineering judgement as well as on a thorough understanding of the design proposal. We speculate that architectural analysis can benefit from being conducted by small teams of engineers that include both safety and reliability engineers on the one hand and design and domain experts on the other. Necessary inputs for the process are design descriptions (and, where available, models), lists of known low-level threats (e.g. short circuits for electrical systems or leaks for hydraulic systems) and design issues (e.g. known limitations of sensors and/or failure diagnostic mechanisms) for the system and a list of high-level safety concerns (such as failure conditions or hazards identified during the FHA).

The process generally proceeds in a top down manner: first considering the level of the system, and then decomposing any complex components into sub-components until the level of basic components is reached.

In order to identify dependencies between the components, safety engineers consider both intended and unintended paths that are established by the design as well as intentional and unintentional interactions over these paths [54]. Furthermore interactions in terms of flows of energy, material and information (including control) should be considered [68, 112].

Unintentional interactions (over intended and unintended paths) are typically captured as specialised commission failure modes (e.g. short circuits, leaks etc.); however, safety analysts also have to consider whether further refinement of such generic failure modes is necessary (e.g. specialising the general “leak” FM into more detailed sub-types such as “slow leak” and “fast leak” failure modes) in the context of particular system, high-level safety concerns and low-level threats. By contrast, intentional interactions typically yield a significantly larger number of failure modes as more subtle deviations from intent need to be considered. To aid the identification of such failure modes a set of guidewords (such as HAZOP [85] and SHARD [123] guidewords) is applied to each interaction in order to find viable interpretations.

Identification of the modes of complex components is similarly guideword-based whereby the four classes of modes discussed throughout this chapter are used as cues. Where detailed reconfiguration rules implemented by controllers are known, these can be used as a ‘starting point’ for the mode model. For each identified mode each component that falls within the mode’s domain is briefly considered and its intent (for the given mode) is clarified. The result of the process is a preliminary mode model that identifies all modes and possible transitions between the modes. However, the details of the transitions’ guards and triggers are typically not specified at this stage (as they are likely to be dependent on the detailed characterisations of basic components); instead, abstract and informal transition conditions are captured and retained until

the detailed failure logic model emerges. Overall, with respect to system modes, the process envisaged here is very similar to Papadopoulos's "Analytical Stage" of the safety analysis process [115].

The process of eliciting a system's and its components' modes and failure modes is highly speculative, creative and judgement-driven. In general, safety engineers can only identify *viable* interpretations of guidewords; whether the resultant modes and interfaces are sufficient or, even, necessary can only be established with certainty during the detailed modelling stage. However, in order to control the modelling process risks proactively and to reduce the incidence of expensive architecture 'roll backs' a partial architecture validation step can be introduced. This step is similar to HAZOP and SHARD and can be seen as a 'lightweight' and scenario-based variant of both FMEA and FTA. During the validation two types of scenarios (or "walk-throughs") are used:

a) *Inductive scenarios:* These start with an initiating event (e.g. a failure) typically related to one of the perceived key low-level threats for the system or its technological domain. The scenario is developed by systematically walking through the architecture to identify how the initiating event affects the system in terms of failure mode propagation and/or change in components states. During scenario development, co-effectors (that affect failure mode propagation) are systematically recorded and different corresponding causal 'branches' are systematically explored. For each branch both the immediate effects (in terms of system-level output failure modes) and ultimate effects (in terms of FMs and any appropriate mode switches) are recorded.

b) *Deductive scenarios:* these are the reverse of the above. The development of a deductive scenario starts with a significant effect condition – typically a system level failure condition (although individual output FMs, 'internal' failure modes, transitions into particular modes as well as any combinations thereof may also be used). The scenario is developed by deductive walkthrough where mode dependencies are considered one flow at the time and the causes of the condition are recursively identified. The process is similar to an informal fault tree analysis applied to the component-and-connector view of the failure logic model architecture.

The primary goal of the above inductive and deductive analyses is to test whether significant system failure behaviour scenarios can be expressed in terms of system modes and of FMs captured by the failure logic model architecture and, thus, to increase confidence in architecture's expressiveness, completeness and correctness. However, when conducted with sufficient rigour and appropriately documented, scenario-based validation can add further value to the safety assessment process:

- By enabling some preliminary safety analysis feedback to the design process (e.g. identification of the key weaknesses and challenges);

- By generating material (i.e. simulation and review 'cases') that can be utilised for simulation- and review- based validation of detailed failure logic models.

It is important to stress that for all but the most trivial systems, definition of the architecture of the failure logic model is a highly iterative process where the 'validation' step highlights deficiencies of the architecture and necessary modifications as well as triggers revision and refinement of both failure modes and system mode models.

5.4.2 Basic Components Characterisation and Model Composition

Once the model architecture is established the detailed characterisations of components can be established in relative isolation. However, in specifying the failure logic of individual basic components safety engineers still need to maintain a system-wide perspective in that the components' intent needs to be considered in a wider context.

The first step at the level of basic components is to establish which of the higher-level modes the component is sensitive to and to establish the intended behaviour of the component in each such mode.

Following this, component characterisations are constructed. The process typically starts with the specification of failures and normal events, followed by the specification of local states (i.e. failure states, normal states and failure handling states) and, eventually, the specification of propagation conditions for all output FMs. Impact of all of the modes the component is sensitive to must be considered at the time of specification of both state transitions (in terms of exposure to individual failures and semantics of input failure modes in transition guards) and failure propagation conditions (in terms of semantics of input and output failure modes as well as all possible interpretations of privatives).

Establishing the failure logic characterisation of the components may, in some cases, identify incompleteness, inaccuracy or an inappropriate level of granularity of the model architecture. Therefore, whilst scenario-based validation of architecture should *minimise* the incidence of 'roll-backs', revisions of the failure logic model's architecture may be triggered by component-level analysis and modelling.

Once component models are complete, the final failure logic models can be assembled. Guided by the architecture, the consolidation of the model proceeds 'bottom-up' through the progressive composition of components' models. At each level of composition, the components interfaces are

connected by FM flows and mode models of complex components are refined. In particular, the abstract transition conditions of dysfunctional and failure mitigation modes are progressively replaced by concrete specifications of guards and triggers in terms of (other) modes, events and FM flows (made available by the component models and their composition respectively). This model consolidation process is broadly similar to the "synthetic step" of analysis proposed by Papadopoulos [115].

Finally, in some cases, analysis of failure logic models of the system or individual complex components can be used to facilitate and inform progressive refinement of the mode models. However, for most non-trivial models the process cannot be fully automated and will often rely on the engineering judgement.

5.5 Case Study: Electrical Power Distribution System

The concept of modes is demonstrated on an Aircraft Electrical Power Distribution System (EPDS). The system closely relates to that deployed on the A320 aircraft, although it has been simplified for the sake of clarity here. This section presents a brief summary of the system, architecture of the failure logic model and components characterisations.

5.5.1 EPDS Overview

The function of the EPDS is to distribute electrical power from three ultimate sources – alternating current generators – to six busbars. Two of the generators (Generator 1 and Generator 2) are identical and are powered by aircraft engines; the third – "emergency" – generator is hydraulically powered.

The six busbars are organised in two groups: three alternating current busbars (AC1, AC2 and AC Essential) and three direct current busbars (DC1, DC2 and DC Essential). AC to DC conversion is achieved by three transformers (Transformer 1, Transformer 2 and Essential Transformer) – each transformer is, in principle, capable of supporting all three DC busbars alone.

The architecture of the system is shown in Figure 79. Traditionally EPDS is divided into three sides: Side 1, Side 2 and "Side Essential" – corresponding to the left, right and centre portions of the diagram. Similarly, it is sometimes useful to consider the EPDS architecture as consisting of DC and AC "sections".

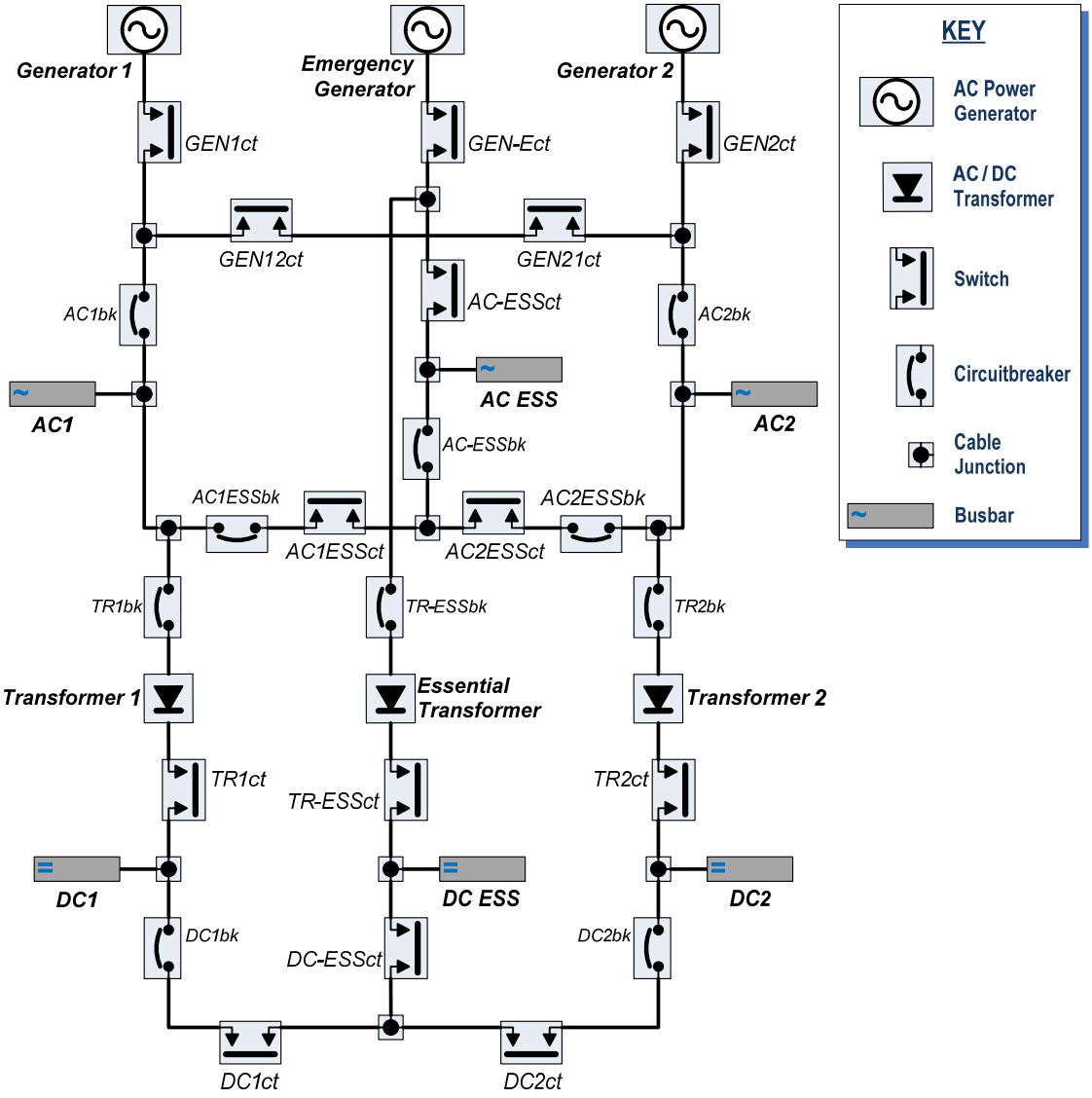


Figure 79 - EPDS Architecture

Each side of the EPDS is connected in a pipe-line fashion and is in principle capable of operating in isolation. However, in order to provide redundancy the sides are connected by three “cross-feed” lines: two in AC section and one in DC section of the system. The flow of the electrical power in the system is controlled via 14 contactors (electrically controlled switches) by four controllers (not shown). Exactly one controller controls each contactor and the configuration is determined by relatively simple rules (Table 7).

Finally, the system contains a number of overcurrent circuit breakers (which are intended to interrupt the circuit in presence of short circuit conditions) as well as a number of functionally passive junctions

Table 7 - EPDS Reconfiguration Rules

Controller ID	Contactor ID	Configuration rules ("closed when…")
GEN	GEN1ct	Generator 1 is operative
	GEN2ct	Generator 2 is operative
	GEN-Ect	AC1 and AC2 are both unable to deliver power
	GEN12ct	Either Generator 1 or Generator 2 is inoperative (but not both)
	GEN21ct	Either Generator 1 or Generator 2 is inoperative (but not both)
AC	AC1ESSct	Generator 1 and AC1 are both operative
	AS2ESSct	Generator 1 or AC1 are not operative while both Generator 2 and AC2 are operative
	AC-ESSct	Both Generator 1 and Generator 2 are inoperative; *or* Transformer 1 or DC1 is not operative and Transformer 2 or DC2 is not operative
TR	TR1ct	Transformer 1 is operative
	TR2ct	Transformer 2 is operative
	TR-ESSct	Essential Transformer is operative
DC	DC1ct	Generator 1 or Generator 2 is operative and DC1 is operative
	DC2ct	Generator 1 or Generator 2 is operative and DC2 is operative and Transformer 1 or DC1 is inoperative; *or* Generator 1 or Generator 2 is operative and DC1 is operative and Transformer 2 or DC2 is inoperative
	DC-ESSct	Generator 1 or Generator 2 is operative and DC1 is operative; *or* Generator 1 or Generator 2 is operative and DC2 is operative and Transformer 1 or DC1 is inoperative

5.5.2 Model Architecture: Key Principles and Assumptions

This section outlines some fundamental principles of the EPDS failure logic model: failure mode types and flows and key failure logic characteristics of the different components.

First, a set of 'basic' failure mode types has to be established. Most of the dependencies between EPDS components are established in terms of electrical power. The flow of electrical power between a source and a consumer establishes dependencies *in both directions* in the failure domain: the consumer's dependency on the source in terms of *voltage* and *current* provision (and associated the failure modes of *Omission* and *Commission*) as well as a reverse dependency in terms of short circuit failure modes. A conceptual power propagation path between two components, thus, yields five failure modes organised in three basic groups.

However, *physical paths* established by EPDS wiring are generally capable of propagating power in both directions; in other words, most physical paths host two conceptual paths described above (and thus can propagate two sets of FMs – one in each direction). The overall failure logic model architecture is shown in Figure 80 where each 'connector line' indicates flows of all five failure modes (associated with the conceptual dependency path).

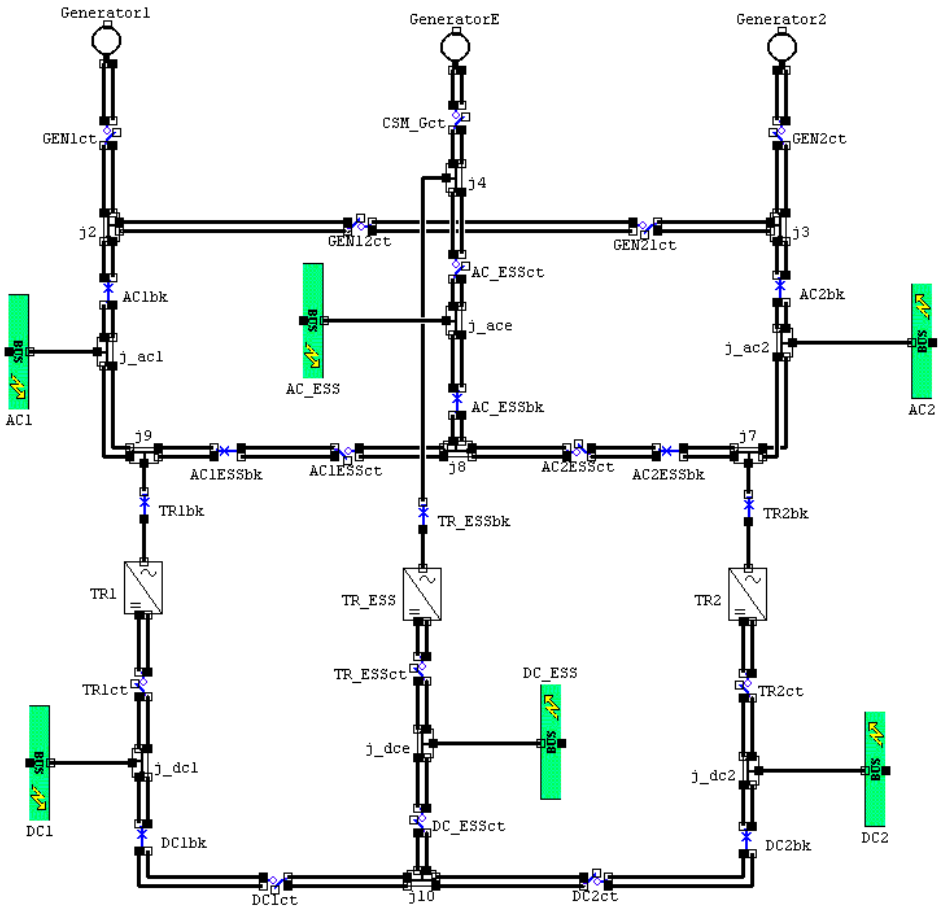


Figure 80 - Architecture of EPDS Failure Logic Model

The most interesting failure mode type in EPDS is a short circuit. This failure mode can be ultimately caused by spontaneous failures of busbars and transformers as well as by inadvertent connections between pairs of live generators or transformers. The immediate effect of this failure mode on *junctions* is that all of the electrical current is drawn towards the source of the short circuit. So a junction powered from one terminal and receiving a short circuit from another terminal will generate an omission of current to the remaining third terminal. The ultimate 'worst case' effect of short circuits is permanent damage to generators (or transformers) leading to a permanent loss of power provision.

Designed to protect the system from short circuits, circuit breakers – when operational – interrupt power flow upon detection of the overcurrent condition. In terms of the failure logic characterisations this means that circuit breakers transform a short circuit input failure mode into a failure handling state ("tripped"). Once tripped, circuit breakers exhibit omissions of current and voltage. For the purpose of this case-study it is assumed that *once tripped, circuit breakers cannot be reset until the end of the flight*. It is also assumed that *an operational circuit breaker will trip before any significant damage is caused to a generator* or a transformer. However, an internal failure (*Stuck*) may prevent a circuit breaker from tripping when necessary.

Turning to current and voltage failure modes, omissions can be caused by:

- Spontaneous failures (*TotalLoss*) of busbars, junctions, transformers and generators
- Transformers and generators being exposed to short circuits (as described above)
- Spontaneous failures of circuit breakers (*OpenSpontan*) and contactors (*FailOpen*)
- Contactors being inadvertently commanded to open by controllers (omission FM)

Furthermore, as was mentioned above, exposure of a junction to a short circuit may result in omission of current (but not voltage) being exhibited on one of the terminals

Commissions of current and voltage can only be caused by spontaneous failures (*FailClosed*) of – or inadvertent commands (commission FMs) to – contactors. Commission of current is modelled predominantly to reflect cases of 'incidental correctness' when busbars are powered when not expected or through an unexpected route. Commission of voltage, however, plays an important part in the correct modelling of situations when live sources of power are inadvertently connected. Operational generators and transformers 'convert' commission of voltage into an output short circuit. The short circuit then propagates through the system allowing any circuit breakers along the way to trip (and thus stop both the short circuit and original commission FMs from propagation) and, potentially, threatening the (other) power source.

Finally, it is important to note that short circuits only propagate towards the source of power. So a junction exposed to a short circuit FM on one terminal will only propagate it to those other terminal(s) that are either exposed to a commission of voltage or are supposed to be powered in a current mode of operation. The mode of operation of this system, therefore, affects the propagation of failure modes in general and the potential damage caused by short circuits in particular.

5.5.3 Failure Logic Model Architecture: Modes

The three EPDS cross-feed lines clearly provide the system with an ability to utilise redundant power distribution paths. As a result, the system may be operated in a large number of alternative modes, each mode establishing an intended behaviour of the system in terms of a unique power distribution path and, thus, an intended configuration of contactors.

In order to elicit modes of the EPDS it is necessary to consider and *rationalise* the reconfiguration rules implemented by controllers (see Table 7 above) and to identify stable configurations. A failure-free EPDS is powered by two non-emergency generators with Generator E disconnected from the network. Each generator powers its own side with the lower AC and DC cross-feed lines configured to power essential busbars from Side 1. Overall, all busbars are intended to be powered. When failures of EPDS components occur, the intent of the system is to reconfigure and

power all busbars through alternative power distribution paths (e.g. with only one non-emergency generator powering the entire system). However, in some circumstances provision of power to all busbars is no longer viable. In these circumstances the network is intended to reconfigure into an "*Emergency Mode*". In this mode the Emergency Generator is connected to the network to power only two essential busbars (in case of *DC_ESS* – through an essential transformer). All other flows of power are disabled and non-essential busbars are left intentionally unpowered. The emergency mode is therefore an example of the *degraded mode* (see Section 5.2.2.3 of Chapter Five). Since busbars lie on the interface of the EPDS and, thus, on the boundary of the domains of the system modes, absence of input failure modes for the non-essential busbars in the Emergency Mode has to be *translated* into omissions of current and voltage output FMs within the busbar characterisation.

Before describing the remaining (non-emergency) modes of the system it is necessary to recognise that reconfiguration rules indicate that the lower AC cross-feed line is only used for powering Side Essential and is never (intentionally) used for powering Side 1 from Side 2 or vice-versa. For the purpose of this case study, it is merely assumed that some compelling justification exists for not utilising all of the opportunities provided by the lower AC cross feed line. However, consulting a more detailed and accurate representation of the system design (e.g. [2]) reveals that the reason for the restriction is that contactors *AC1ESSct* and *AC2ESSct* are in fact a single assembly; the design of the assembly, in the absence of failures, prevents both contactors from being closed simultaneously.

Overall, non-emergency modes can be captured through three 'parallel' models that relate to the ability of generators to power the AC section of the system (*ACG modes*), to the effectiveness of the lower cross-feed line in terms of powering Side Essential from Sides 1 and 2 (*XF modes*) as well as to possible power distribution routes in the DC section of the EPDS (*DC Modes*) – see Figure 81.

The abstract modes' transition conditions are:

- *Alternate ACG* modes are entered whenever the corresponding side of the AC section of the system cannot be powered effectively by its own generator
- *Alternate XF* modes are entered whenever the corresponding (AC) side becomes incapable of providing power effectively to the essential side
- *Alternate1* and *Alternate2 DC* modes are entered whenever the corresponding (DC) side becomes incapable of providing power effectively to the essential side; *AlternateE* mode is entered whenever both non-essential sides become ineffective.
- The *Emergency* mode is entered whenever effective power flow through both AC side 1 and AC side 2 becomes impossible.

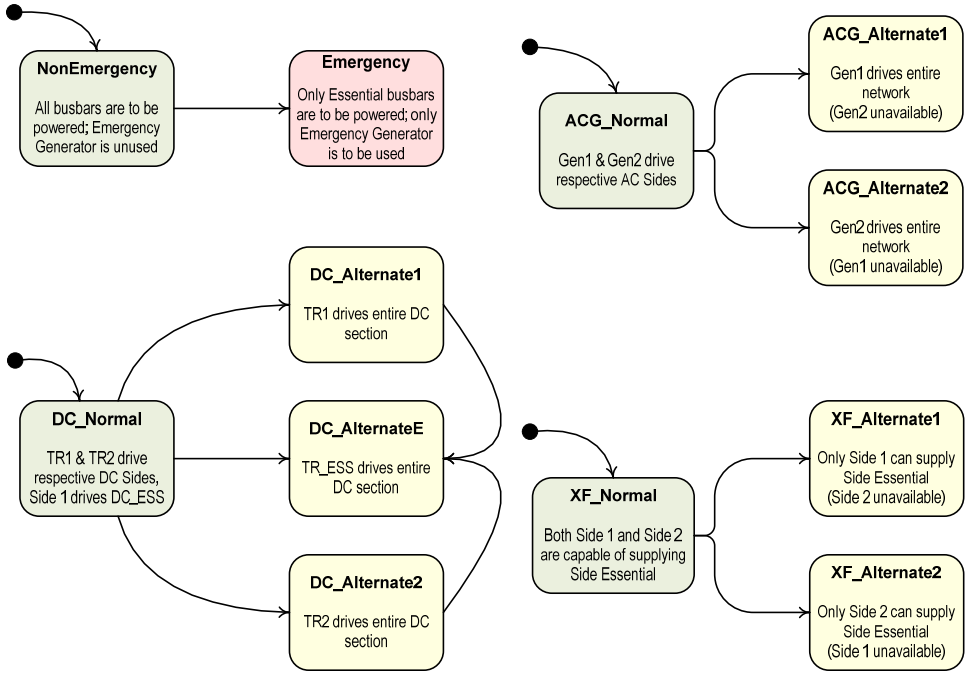


Figure 81 - EPDS Mode Model (partial)

5.5.4 Components Failure Logic Characterisation

Table 8 documents the modes to which some of the EPDS components are sensitive. It is important to note that five components of the EPDS – two essential busbars and all three generators – are not sensitive to any modes (their intent is static throughout operation of the system and regardless of any failures).

Once the significant modes and respective intents for (each) component are clarified, the task of characterising the failure logic of the basic components becomes broadly similar to that in a mode-free context. Figure 82 (presented across two pages below) shows characterisation of the *J2* junction in AltaRica OCAS[49]. The *trans* clause of the characterisation contains sections dedicated to both failure state transitions and (reading of) EPDS mode broadcasts. With respect to the latter section, two groups of modes are being read (in accordance with Table 8): NonEmergency / Emergency and ACG modes; in order to reduce complexity, other 'irrelevant' modes (i.e. DC and XF modes) are not being read.

Finally, the *assert* clause of the characterisation specifies failure mode propagation conditions which refer to EPDS modes whenever appropriate.

[49] Only partial characterisation is shown. For the sake of brevity definitions of I/O flows and some of the propagation conditions are suppressed.

Table 8 - Basic Components' Sensitivity to- and Intent in- Modes

Component(s)	Mode	Intent in the mode
AC1, AC2, DC1, DC2 (non-essential busbars)	*NonEmergency*	Each busbar intended to be powered
	Emergency	Each busbar is intended to be unpowered in this mode. Lack of input FMs is "translated" into omission FM. Exposure to commission of current may lead to "incidental correctness"
J2 (wiring junction)	*Emergency*	No terminals are intended to be powered
	NonEmergency	One terminal of the junction is intended to be powered (depending on the ACG mode – see below)
	ACG_Normal	Only upper terminal of the junction is intended to be powered, other terminals are only powered when exposed to commission FM
	ACG_Alternate1	
	ACG_Alternate2	Only side terminal of the junction is intended to be powered
J3 (wiring junction)	*Emergency*	No terminals are intended to be powered
	NonEmergency	One terminal of the junction is intended to be powered (depending on the ACG mode – see below)
	ACG_Normal	Only upper terminal of the junction is intended to be powered, other terminals are only powered when exposed to commission FM
	ACG_Alternate2	
	ACG_Alternate1	Only side terminal of the junction is intended to be powered
GEN1ct	*Emergency*	Upper terminal is intended to be powered; lower terminal is intended to be unpowered. The contactor is intended to be open. Undue closing of the contactor may result in commission of power exhibited on lower terminal; in absence of other input FMs it will not result in commission of power on upper terminal.
	NonEmergency	Intent is determined by the ACG mode (see below).
	ACG_Normal	Upper terminal is intended to be powered, lower terminal – unpowered, and contactor - closed. Short circuit typically propagates freely in both directions; commission of power typically propagates freely upwards.
	ACG_Alternate1	
	ACG_Alternate2	Both terminals are intended to be powered; contactor is intended to be open. Inadvertent closing of the contactor may result in commission of power in both directions.
AC1ESSbk	*Emergency*	Neither terminal is intended to be powered. Short circuits do not propagate nor cause failure handling mode transition unless the opposite terminal is exposed to the commission of voltage.
	NonEmergency	One of the terminals is expected to be powered (depending on ACG and XF modes – see below)
	XF_Normal	Intent is determined by the ACG mode (see below)
	XF_Alternate1	Left terminal is intended to be powered, right terminal – not.
	XF_Alternate2	Right terminal is intended to be powered, left terminal – not.
	ACG_Normal	If in XF_Normal then intent is identical to that in XF_Alternate1; otherwise these modes have no effect.
	ACG_Alternate1	
	ACG_Alternate2	If in XF_Normal then intent is identical to that in XF_Alternate2; otherwise this mode has no effect.

```
node j2
  state
    FailSt : {OK,Lost} ;
    Mode : {NonEmergency,Emergency} ;
    ModeACG : {Normal,Alternate1,Alternate2} ;
  event
    TotalLoss,
    BroadcastEmergency, BroadcastAlternate1, BroadcastAlternate2 ;
  init
    FailSt := OK ;
    Mode := NonEmergency ;
    ModeACG := Normal ;
    // T-junction component for j2
    // The component has three terminals A, B and C that can be connected to the circuits.
    // Depending on system configuration each terminal can (directly or indirectly) connect to a circuit that
    // produces or consumes the power. Therefore each terminal is associated with two ports (Xin and Xout)
    // to model failure modes associated with electrical power flows into the junction as well as out of the
    // junction (resp.)
    // Note: associated with each power flow are three groups of FMs - Current and Voltage FMs (that flow
    // in the same direction as power) as well as ShortCircuit FM (that flows in the opposite direction)
    // So Ain^Voltage, Ain^Current and Aout^ShortCircuit are all inputs). Voltage and Current FM groups are
    // modelled as enumerated types {ok,Omiss,Commis}; ShortCircuit FM is modelled as Boolean
  trans
    // *** Failure & Failure State Transition***
    FailSt = OK |- TotalLoss -> FailSt := Lost; // Junctions can only fail completely (no flow in any direction)
    // *** "Reading" mode broadcasts ***
    // Emergency Mode:
    true |- BroadcastEmergency -> Mode := Emergency;
    // ACG Modes:
    true |- BroadcastAlternate1 -> ModeACG := Alternate1;
    true |- BroadcastAlternate2 -> ModeACG := Alternate2;
  assert
    // *** Failure Propagation Conditions: Voltage FMs ***
    (Aout^Voltage =  ( case {                                                    // Lower terminal
                        Mode != Emergency
                        and ( (FailSt = Lost) or
                              ( (ModeACG = Normal or ModeACG = Alternate1)
                                and Bin^Voltage = Omiss Cin^Voltage != Commis ) or
                              ( ModeACG = Alternate2
                                and Cin^Voltage = Omiss and Bin^Voltage != Commis )  )          : Omiss,
                        Mode = Emergency and FailSt = OK
                        and (Bin^Voltage = Commis or
                              Cin^Voltage = Commis)                                          : Commis,
                        else ok})) and
    (Bout^Voltage = ( case {                                                     // Upper terminal
                        Mode != Emergency and ModeACG = Alternate2
                        and ( FailSt = Lost or
                              ( Cin^Voltage = Omiss and Ain^Voltage != Commis ) )             : Omiss,
                        ( Mode = Emergency or ModeACG = Normal or ModeACG = Alternate1 )
                        and FailSt != Lost
                        and ( Ain^Voltage = Commis or Cin^Voltage = Commis)                   : Commis,
                        else ok})) and
    // *** Failure Propagation Conditions: Current FMs ***
    // (Broadly similar to voltage FM but possibility of a short circuit
    // drawing current away from a terminal is considered)
    (Aout^Current =  ( case {                                                    // Lower terminal
                        Mode != Emergency
                        and ( (FailSt = Lost) or
                              ( (ModeACG = Normal or ModeACG = Alternate1)
                                and ( Bin^Current = Omiss or
```

```
                    (Cout^ShortCircuit and not Aout^ShortCircuit))
                and ( Cin^Current != Commis or
                    (Bout^ShortCircuit and not Aout^ShortCircuit)) ) or
              ( ModeACG = Alternate2
                and ( Cin^Current = Omiss or
                    (Bout^ShortCircuit and not Aout^ShortCircuit))
                and ( Bin^Current != Commis or
                    (Cout^ShortCircuit and not Aout^ShortCircuit)) ) )          : Omiss,
          Mode = Emergency and FailSt = OK
          and ( ( Bin^Current = Commis
                and (Aout^ShortCircuit or not Cout^ShortCircuit) ) or
              ( Cin^Voltage = Commis
                and (Aout^ShortCircuit or not Bout^ShortCircuit)) )          : Commis,
          else ok})) and
```

```
// *** Failure Propagation Conditions: Short Circuits ***
// - A short circuit on any terminal may propagate only to other terminals
// - A short circuit will not propagate if the junction has failed
// - A short circuit will only propagate to terminals that supply voltage
(Bin^ShortCircuit = ( (Aout^ShortCircuit or Cout^ShortCircuit)        // Upper terminal
          and FailSt = OK
          and ( ( Mode != Emergency and ( ModeACG = Normal or ModeACG = Alternate1 )
                and Bin^Voltage != Omiss ) or
              Bin^Voltage = Commis ) )  )
```

Figure 82 - Partial Characterisation of J2 Junction (AltaRica OCAS)

5.5.5 Refinement of Mode Models

Once characterisations of all of the basic components become available the overall failure logic model of the EPDS is composed. Whilst the component composition part of this task is trivial, the task also requires refinement of the abstract modes models (with respect to the specification of mode transitions).

Considering the ACG *modes* first, informally, the transition from *ACG_Normal* mode to *ACG_Alternate1* or *ACG_Alternate2* modes will take place when *the corresponding side of the AC section (i.e. side 2 and side 1 respectively) of the system cannot be effectively powered by its own generator*. This condition is formalised for *ACG_Alternate2* mode as following:

> *The system is in ACG_Normal mode and either Generator 1 exhibits omission of voltage failure mode or GEN1ct contactor fails open.*

It is important to observe that this transition specification is not identical to the corresponding *GEN* controller rule which merely considers the voltage output of *Generator 1*. This indicates that a limitation of the controller that has to be addressed in the model (see next section). However, in itself, it is a potentially valuable observation from the perspective of the safety assessment of EPDS. It highlights possible improvement to the configuration rules and indicates that justification of the limitation should be sought from the system design engineers.

Table 9 shows a transition specification (in pseudo-code) for all modes identified in Figure 81 above. In the AltaRica OCAS model of the EPDS the resultant mode models are captured in four dedicated mode observers (one for each mode model). Characterisations of some of the basic components (some of the contactors) are modified to allow them to 'communicate' occurrence of relevant failures (e.g. *GEN1ct.FailOpen* in the above case) to the environment. Finally, the events of the mode observers (both transition events and mode broadcasts) are synchronised with the appropriate component's events.

Table 9 - EPDS Mode Transitions Specification

Mode	Refined Transition Specification	Abstract Specification
Emergency	In *NonEmergency* mode both junctions *j7* and *j9* exhibit *omission of current* FM *on essential side* terminals	The mode is entered whenever effective power flows through both AC side 1 and AC side 2 become impossible
ACG_Alternate1	In *ACG_Normal* mode *Generator2* exhibits omission of voltage or *GEN2ct fails open*	Alternate ACG modes are entered whenever the corresponding side of the AC section of the system cannot be effectively powered by its own generator
ACG_Alternate1	In *ACG_Normal* mode *Generator1* exhibits *omission of voltage* or *GEN1ct fails open*	
XF_Alternate1	In *XF_Normal* mode *AC2ESSbk* exhibits *omission of voltage* on essential side terminal or *AC2ESSct fails open*	Alternate XF modes are entered whenever corresponding (AC) side becomes incapable of effectively providing power to the essential side
XF_Alternate2	In *XF_Normal* mode *AC1ESSbk* exhibits *omission of voltage* on essential side terminal or *AC1ESSct fails open*	
DC_Alternate1	In *DC_Normal* mode *DC2bk* exhibits *omission of voltage* FM on the essential side terminal or *DC2ct* fails open	Alternate1 and Alternate2 DC modes are entered whenever corresponding (DC) side becomes incapable of effectively providing power to the essential side
DC_Alternate2	In *DC_Normal* mode *DC1bk* exhibits *omission of voltage* FM on the essential side terminal or *DC1ct fails open*	
DC_AlternateE	In *DC_Normal* mode *simultaneously* (*DC1bk* starts exhibiting *omission of voltage* FM on the essential side terminal or *DC1ct fails open*) **and** (*DC2bk* starts exhibiting *omission of voltage* FM on the essential side terminal or *DC2ct fails open*); **or** In *DC_Alternate1* mode *DC1bk* exhibits *omission of voltage* FM on the essential side terminal or *DC1ct fails open*; **or** In *DC_Alternate2* mode *DC2bk* exhibits *omission of voltage* FM on the essential side terminal or *DC2ct fails open*	AlternateE mode is entered whenever both non-essential sides become ineffective

5.5.6 Characterisation of Controllers

The previous section has highlighted that EPDS mode transitions do not mirror the reconfiguration rules of the controllers. This means that either the mode model has to be modified or the characterisation of the controllers needs to take into account that in some modes due to design limitations controllers are capable of exhibiting failure modes even in the absence of internal failures.

The latter approach is favoured and illustrated on the *GEN* controller of EPDS. The controller commands five contactors based on voltage status at four points in the electrical network (Figure 83, page 202) and on predetermined reconfiguration rules (see Table 7 on page 192 above). For simplicity it is assumed that the controller cannot fail; revoking this assumption, however, is relatively trivial. The controller is clearly sensitive to both *NonEmegrency* / *Emergency* and *ACG* modes. The intent allocated to the controller in each mode is shown in Table 10.

Table 10 - Intended Behaviour of GEN Controller in Various Modes

	Intended Command to Each Contactor				
Mode	**GEN1ct**	**GEN2ct**	**GEN12ct**	**GEN21ct**	**GEN-Ect**
Emergency	Open[50]	Open	Open	Open	Close
NonEmergency	*Depending on ACG mode*				Open
ACG_Normal	Close	Close	Open	Open	*Irrelevant*
ACG_Alternate1	Close	Open	Close	Close	
ACG_Alternate2	Open	Close			

However, the controller is only capable of observing the system status at four points. Therefore the status 'perceived' by the controller may be different from the real status of the system (as reflected in the system modes). For example, whilst the controller is capable of correctly opening *GEN1ct* and closing cross-feed contactors when *ACG_Alternate2* mode is entered due to lack of voltage from *Generator1*, it will fail with respect to its intent (and will exhibit a commission FM on the output to *GEN1ct*) whenever the alternate mode is entered through another cause (e.g. failure of the *GEN1ct*). Furthermore, as the controller logic is determined by the voltage measurement between *GEN1ct* and *Generator1*, it has no way of determining which generator supplies this voltage; as a result, commission of voltage from the upper terminal of *GEN1ct* will also cause a commission FM of the controller.

[50] For control of all contactors (except *GEN-Ect*) in *Emergency* mode the intent associated with the *ACG* mode is irrelevant.

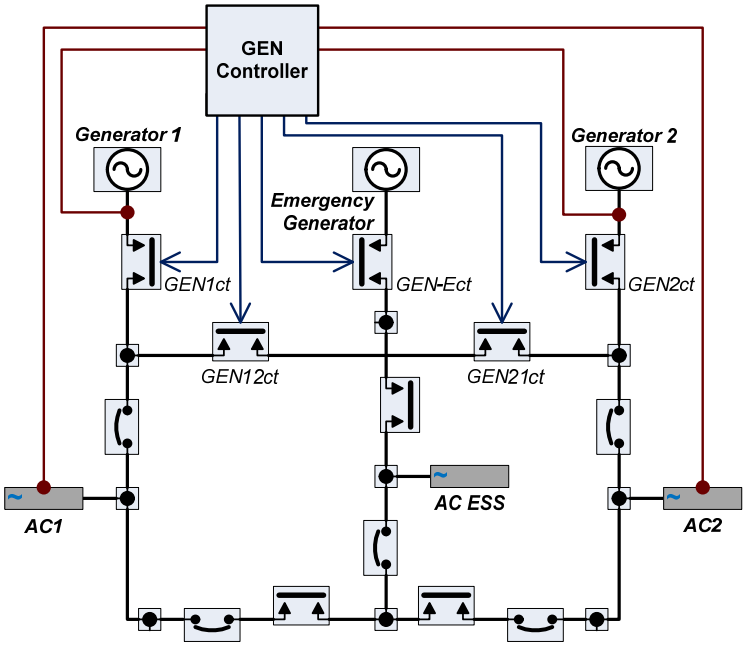


Figure 83 - GEN Controller in EPDS

The same principle applies to the *Emergency* mode and control of the *GEN-Ect* contactor. Finally, the specification of the intent of the *Emergency* mode has stipulated than all contactors which do not lie on the direct path from the *emergency generator* to the *essential busbars* have to be opened (in order to avoid interference with the emergency provision and minimise threat to the 'resource of last resort'). This intent is not fully implemented in the GEN controller and, thus, in some circumstances non-emergency generators' and cross-feed contactors may be inadvertently closed in the emergency mode.

```
node GEN_Controller
    // Controller can be influenced by presence or absence of voltage failure modes at points of observation
    // Note #1:  controller is sensitive to the GEN1ct and GEN1ct failure modes (exhibited on upper terminals)
    // Hence input FM groups include "forward" FMs (e.g. GEN1fw) and "backward" FMs (GEN1bw) representing
    // FM flows from Generators and Contactors respectively.
    // Note #2: Component is only sensitive to voltage failure mode (and not current FMs or ShortCircuits)
    // Note #3: All FM groups (i.e. input voltage FMs and output control FMs) are modelled as a simple
    // enumerated type {ok, Omiss, Commis}
```

```
  assert
    // *** Failure Mode Propagation Conditions ***
    (GEN1ct = ( case {
                    Mode != Emergency and ModeACG != Alternate2
                    and GEN1fw = Omiss and GEN1bw != Commis          : Omiss,
                    (Mode = Emergency or ModeACG = Alternate2)
                    and (GEN1fw != Omiss or GEN1bw = Commis)         : Commis,
                    else ok})) and
    (GEN12ct = ( case {
                    Mode != Emergency
                    and ( ( ModeACG = Alternate1            // Controller doesn't recognise that Side 2 failed
                        and ( (GEN2fw != Omiss or GEN2bw = Commis)
                            and GEN1fw != Omiss and GEN1bw != Commis)) or
                      ( ModeACG = Alternate2                // Controller doesn't recognise that Side 1 failed
                        and ( (GEN1fw != Omiss or GEN1bw = Commis)
                            and GEN2fw != Omiss and GEN2bw != Commis)) or
```

```
                ( ModeACG != Normal              // Controller "believes" that
                  and ( (GEN1fw = Omiss and GEN2fw = Omiss                  // both sides failed
                        and GEN1bw != Commis and GEN2bw != Commis) or       // or
                       ( (GEN1fw != Omiss or GEN1bw = Commis)               // both sides healthy
                        and (GEN2fw != Omiss or GEN2fw = Commis))
              ) )       )                                          : Omiss,
            (Mode = Emergency or ModeACG = Normal)                 // Contactor should be open, and
            and not (GEN1fw = Omiss and GEN2fw = Omiss              // controller doesn't see both
                  and GEN1bw != Commis and GEN2bw != Commis) //      sides as failed, and
            and ( (GEN1fw = Omiss and GEN1bw != Commis) or          // controller sees either side 1
                  (GEN2fw = Omiss and GEN2bw != Commis) )           //       or side 2 as failed
                                                                   : Commis,
            else ok})) and
(GEN21ct = ( case {                  // Identical to GEN12ct above
```

Figure 84 - Characterisation of GEN Controller (AltaRica OCAS)

5.6 Key Findings and Limitations

This section presents an overview of key limitations identified in the course of implementing expanded failure logic modelling methodology (presented in the current chapter) and illustrated by the EPDS case study. The key findings are:

- Complexity of the retrospectively-constructed mode models of the system
- Time complexity of the analysis of models specified in AltaRica OCAS
- The types of loops encountered in the models (and their effect on model analysis)
- The need for constraining the order of transitions with void triggers
- Complexity of component characterisations

5.6.1 Loops

The EPDS architecture clearly contains a number of loops (Figure 85 highlights two such loops). The presence of these loops in the failure logic model of the system threatens our ability to analyse it either through systematic parsing (as in Papadopoulos's approach to fault tree synthesis) or through exhaustive simulation (as implemented by the Cecilia OCAS sequence generator). The majority of the loops are either purely syntactical or allow flows to converge to a correct fix point. In such cases the loop is 'resolved' by injecting a trivial 'instantaneous delay' (implemented in a simple virtual component where the delay is guarded by an instantaneous event). Nevertheless, in some cases more complex *ad hoc* solutions are necessary as loops may, in principle, stabilise in an incorrect state. To illustrate the latter case consider the wiring loop in the AC section of the system formed by two cross-feed lines. For simplicity consider a scenario when in Normal mode of operation, first all of the circuit breakers and then all of the switches in the loop fail in closed position. According to the modelling assumptions, commission of voltage (and current) will be generated in the loop; this commission will be propagated to two non-emergency generators that will 'convert' it to short circuit failure modes. Consider that as a result of such simultaneous

exposure to short circuits both non-emergency generators fail simultaneously. Clearly there is no longer power provision in the system. However, the loop in the EPDS model would stabilise in a wrong ‘equilibrium state’ and will continue exhibiting commission FMs.

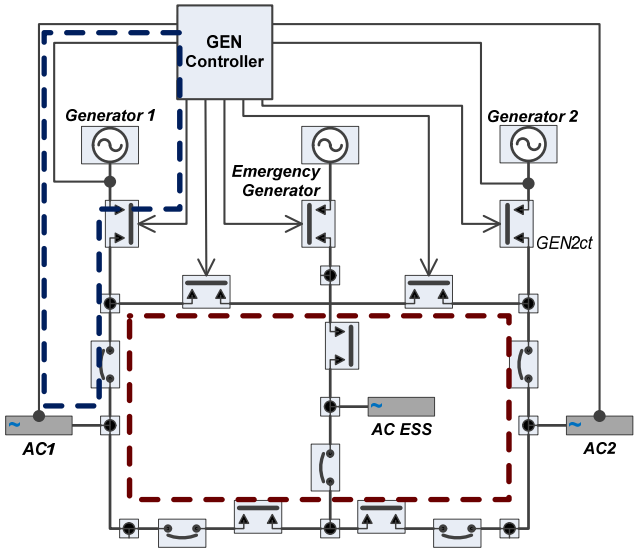


Figure 85 - Some Loops in the AC Section of EPDS

To resolve such loops more complex solutions than a mere unit delay may be necessary (typically requiring instantaneous injection of some behaviour – such as the omission FM – into the loop, effectively to ‘shake it out of’ the wrong equilibrium).

5.6.2 Order of Transitions with Void Triggers

Thirdly, the failure logic model of EPDS contains a large number of state transitions with a void trigger: tripping of the circuit breakers and failures of the generators (and transformers) when exposed to short circuits as well as transitions in ‘unit delay’ components (see above) and mode observers. As a result, in some circumstances, guards of more than one instantaneous transition become *true* at exactly the same point of time. The order in which transitions are executed may affect the resultant state of the model and only some orders may be valid. For example, when a busbar exhibits a short circuit FM, typically at least one circuit breaker and exactly one generator are simultaneously exposed and in principle either one can change the state as a result. However, in practice, the design of these components guarantees that failure-free circuit breakers will trip before the generator sustains any significant damage. In the AltaRica OCAS, this assumption can be captured by assigning instantaneous events an *explicit priority*. This imposes a total order on all of the instantaneous events in the model.

In the EPDS model instantaneous events are assigned to one of the five priority classes as indicated below:

(i) Instantaneous events in unit delay virtual components

(ii) Instantaneous events associated with Failure Handling State transitions of circuit breakers

(iii) Instantaneous events associated with Failure State transitions of Generators and Transformers

(iv) Instantaneous synchronisations associated with the broadcast of modes

(v) Instantaneous synchronisations associated with the broadcast of mode triggers

5.6.3 Complexity of Component Characterisations

The experience with modelling reconfigurable systems under the failure logic modelling approach has uncovered (and the EPDS case study has illustrated) that for such systems models of individual components are significantly more complex that has been previously described in the publications on individual techniques. The source of complexity is two-fold:

- Non-local context-dependency of failure logic of the components increases the number of variables that need to be considered in specifying propagation equations of the output failure modes
- The reconfiguration logic implemented by the controllers may be misaligned with the corresponding mode models

Considering the latter issue first, it is important to note that the design of the EPDS has been considered "as is": failure logic modelling has not actively influenced the design, and the mode model implemented in the failure logic model had to be constructed post-factum. This 'forensic' reconstruction of modes and rationalisation of the configuration rules sometimes yields complex characterisations of the controllers. Other model- based approaches are likely to yield more simple and intuitive models in such 'over the wall' contexts. Therefore, *unless retrospective rationalisation of configuration rules is considered to add significant value to the system assessment,* failure logic modelling may not be the most effective approach for the purely reactive analysis of design proposals that include detailed (and relatively complex and/or distributed) reconfiguration rules.

Returning to the first issue, the mode-dependency of all model components inevitably increases the complexity of component models. However, it can be observed from Table 8 (page 197 above) that, whilst dependencies of different components with the same design (e.g. junctions *J2* and *J3*) on higher-level context (i.e. modes) are different, components may be insensitive to some aspects of the context (e.g. *XF* and *DC* modes for *J2* and *J3*) and their context-dependency may

show *common patterns*. Indeed, junctions *J2* and *J3* have identical sets of four entries in the "intent in the mode" column of the table; the only difference between two junctions is what EPDS modes each of those entries is mapped to. This observation can be generalised as follows:

Failure logic of individual basic components is only dependent on a fixed set of aspects of the high-level context.

It can further be observed that if such a fixed set of important aspects (we call it "local modes") is captured locally (e.g. through a local enumerated variable or a set of Boolean variables), and if the characterisations of all propagation conditions and state transitions are only permitted to refer to higher-level modes by virtue of such local modes then *these characterisations themselves become reusable across similar components*. The only non-reusable part of the component logic is 'isolated' in the mapping between local and high-level modes.

Indeed, each of the junctions in the EPDS model has a subset of four simple local modes characterised by the following intents:

a) Only the upper terminal of the junction is intended to be powered
b) Only the lower terminal of the junction is intended to be powered
c) Only the side terminal of the junction is intended to be powered
d) No terminals are intended to be powered.

The propagation conditions of the junctions are not only simplified by referring to one of the four local modes (instead of the EPDS modes) – they become identical for all of the junctions in the system. Furthermore, the characterisations of the junctions could be reused in a mode-free context by permanently 'enabling' the local mode (appropriate for the component's context).

This solution was used extensively in the complete EPDS model and it was observed that it uses a principle similar to *instantiation* in the object oriented modelling paradigm. The solution allowed us to capture the behaviour of busbars, circuit breakers, contactors, junctions and transformers in five unique partial component models; these 'super characterisations' were reused across the entire failure logic model.

5.6.4 Analysis Complexity

Introduction of the concept of "mode" also had an effect on the duration of the model analysis. The sequence generation applied to the EPDS model presented in the previous section takes about seven minutes to be completed to the cardinality of three events and a couple of hours – to the cardinality of four. Similarly to the previous chapter, this is significantly higher than what would be expected from a modern Fault Tree Analysis tool for the model with the comparable number of

basic events. Whilst some of these overheads must be attributed to the inherent complexity of the failure logic models with modes, a significant proportion can be attributed to the particular characteristics of the AltaRica OCAS language and metamodel implementation schema adopted by the author.

Firstly, AltaRica OCAS is a strictly compositional language that does not permit specification of the state at the level of complex nodes. This means that the modes (which are states of such nodes) have to be emulated by dedicated 'virtual components', which inevitably increases analysis overheads. Secondly, the metamodel implementation schema adopted in this chapter has intentionally favoured the clarity of the mapping between failure logic modelling concepts and specification language constructs over the time efficiency of the model analysis. In particular allowing virtual mode observers to communicate modes to the components through AltaRica flows rather than computationally expensive synchronisations would significantly reduce the analysis time.

Finally, it is important to reiterate that the present analysis tool adopts a brute force approach to sequence generation. It is expected that the new generation of tools, currently under development, will result in a further significant reduction of the analysis time.

5.7 Related Work

Whilst the challenges posed by multi-phase, multi-mode and dynamically reconfigurable systems are well recognised, the author is only aware of a single publication – Papadopoulos in [115] – that systematically addresses this challenge in the context of failure logic modelling approach.

In his approach, tailored to HiP-HOPS, Papadopoulos establishes a hierarchical dynamic model of the system in parallel to the hierarchy of 'static' IF-FMEA tables (Figure 86). The dynamic model is captured through a Statechart-like notation. The dynamic model is hierarchical in two senses:

- The layers of dynamic model mirror component decomposition (and, thus, hierarchy of the IF-FMEA tables)
- Within each layer every mode can be decomposed into a chart of sub-modes

Papadopoulos provides relatively strict rules to define what constitutes a mode and sub-mode; in particular a mode is defined as "*an abstract functional state in which the system maintains a stable functional profile*", and a sub-mode as: "*a distinguishable abstract state within a mode, in which a certain structural configuration of components is employed to deliver the mode functionality*".

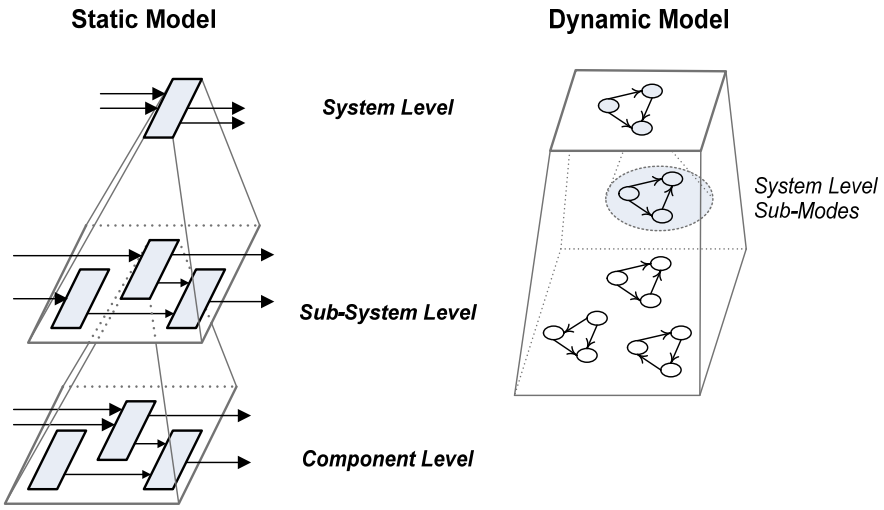


Figure 86 - Static and Dynamic Model Hierarchies in HiP-HOPS

Finally, each mode chart can "communicate" with any other mode chart in the dynamic hierarchy (through a global broadcast) and with the IF-FMEA tables in the static hierarchy. Within the IF-FMEA tables a new column – "Relevant Modes" – is added. Whenever more than one mode is entered into that field, the overall semantics of the field is a disjunction of the entries. The semantics of the entire row of the table is a conjunction of the "Relevant Modes" field and a disjunction of "Input Deviation Logic" and "Component Malfunction Logic" fields respectively.

The HiP-HOPS solution is broadly similar to the one presented in this chapter. Papadopoulos's "modes" broadly map to Alternative Modes of Operation, Phases of Operation and Dysfunctional Modes as discussed in section 5.2, whilst his sub-modes are similar to Reconfiguration and Failure Mitigation Modes. Nevertheless, a number of important differences can be identified:

(1) Whilst this thesis's insistence on a metamodel definition facilitates implementation in a single specification language, HiP-HOPS solution relies on two separate formalisms.

(2) The HiP-HOPS mode charts communicate exclusively through global broadcasts; therefore, whilst for the purpose of editing the mode charts are maintained locally, at the level of complex component, they are effectively *global constructs*. This clearly increases complexity of the HiP-HOPS models compared to models compliant with the FLMM (which allows restricting the domains of the modes to boundaries of the appropriate complex component).

(3) HiP-HOPS mode charts and IF-FMEA tables do not appear to allow definitions of "translation components" on the boundaries of mode domains (indeed, there is no notion of the mode domain in HiP-HOPS). Therefore, composition of two or more independently defined models may require integration of their dynamic models and/or modification of the components' characterisations. This is similar to the solution that was deliberately avoided in section 5.2.2.3 as (further) undermining composability of failure logic models.

(4) It is unclear how a number of orthogonal mode models – such as EPDS's ACG, DC and XF modes – can be specified at the level of a single complex component (or a system) under the HiP-HOPS schema. In particular, the apparent format of IF-FMEA tables does not allow the specification of conjunctions over modes. We can only assume that orthogonal mode charts have to be combined into a single 'cross product chart'. This clearly makes HiP-HOPS mode models suffer from *explicit* 'state explosion': the mode chart of the EPDS would contain 29 modes (versus 12 in the EPDS model presented here)

(5) Despite mode information's being made available to IF-FMEAs, the format of the tables only permits modelling of dynamic mode-dependent exposure intervals of individual failures if, for each failure, the set of modes in which it can occur is identical to the set of modes in which it can result in an output failure mode[51]. One of the implications of this constraint is that any failure of an unused 'spare' is assumed to have a probability of zero (i.e. no 'warm spares' are allowed).

On the other hand, the HiP-HOPS facility for specialisation of modes into sub-modes can clearly be beneficial in terms of the organisation and management of mode models. Indeed, with such facility at hand the mode model of EPDS could be further rationalised as in Figure 87.

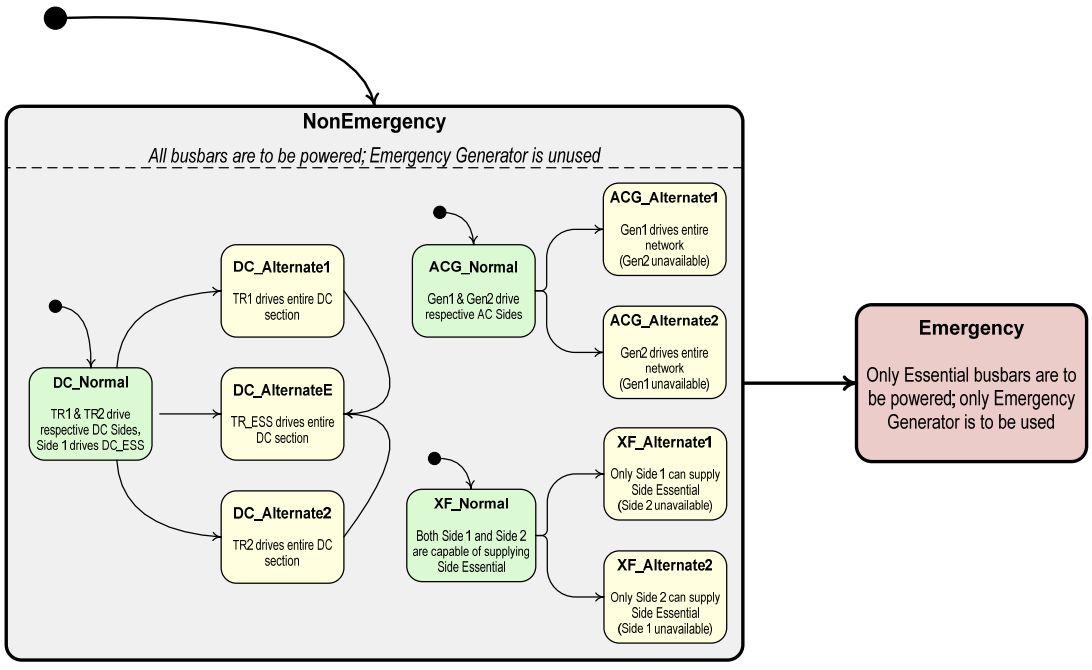


Figure 87 - Hierarchical Representation of EPDS Modes

At the same time, however, we find that the strict rules imposed by Papadopoulos on the engineering semantics of modes and sub-modes sometimes defeat the value added by the hierarchy of modes. In practice, the failure mitigating mode (e.g. alternative configuration) may

[51] "Dynamically extended" IF-FMEA tables are conceptually equivalent to the collection of independent "mode-unaware" tables for each possible mode. (This equivalence is explicitly noted by Papadopoulos)

be persistent across more than one mode or phase of operation. Furthermore, it may *prohibit* further transitions into certain modes or phases. In such systems, without the flexible approach to engineering semantics of modes and sub-modes, the information hiding utility of hierarchical organisation of mode models is lost and hierarchy is reduced to a decorative facility only.

> ***Example:*** *A fuel management system of a modern aircraft is operated in a succession of phases: the Load Alleviation Transfers phase at take-off, the Main Transfers phase during the flight and 'reverse' Load Alleviation Transfers after descend-to-land. In HiP-HOPS each of these phases would be captured as a mode.*
> *Some failures of hydro-mechanical equipment of the system may render some (redundant) fuel galleries inoperative and, thus, require the re-routing of main transfers. Modes yielded by such reconfigurations would be modelled as sub-modes in HiP-HOPS.*
> *However, such alternate configurations may prohibit any further load alleviation transfers in later phases of flight (since these are deemed as non-essential for flight safety). In terms of the HiP-HOPS model this would mean that transition between two modes will be predicated on a sub-mode of the 'source' mode or, in other words, within each mode the initial sub-mode will depend on the exact transition into the mode. Such a model is not hierarchical.*

5.8 Conclusions

In this chapter the issues surrounding multi-modal and dynamically configured systems have been discussed. It was shown that the metamodel presented in Chapter 3 is not sufficiently expressive and does not permit accurate modelling of such systems. By implication, the same criticism applies to all of the failure logic modelling techniques subsumed in that the 'baseline' Failure Logic Metamodel.

Extension to the metamodel has been proposed and its impact on the failure logic modelling process has been analysed. Furthermore, the chapter has demonstrated how the extended metamodel can be implemented in a standard third-party specification language (AltaRica OCAS) even in the context of apparent language limitations. This latter contribution reinforces the principled approach to the FLM Framework definition taken in this thesis, which facilitates separation between conceptual- and implementation notation specific- concerns.

The FLM Framework extension presented in this chapter favours the flexible treatment of different 'types' of system modes such as phases of operation, degraded and failure mitigation modes. It was shown that in practice, not only are these classes of modes broad and not mutually exclusive, but they merely denote particular patterns of mode models. It was further argued that a

more restrictive approach (such as that taken by Papadopoulos), whilst apparently providing structure and guidance to the model construction process, is likely to be inefficient and unintuitive in practice for some systems.

Throughout the discussion in this chapter, it was shown how specific issues of system modes can be generalised into an observation concerning the *context dependency of the component characterisations* in failure logic models. It was shown that, in general, the FLM Framework (regardless of the notation or technique employed) does *not* facilitate the reuse or the 'compositionality' of the safety assessment. Whilst this is a fundamental characteristic of the modelling approach, partial remedies in terms of model construction process and modelling approach have been proposed. The proposed solutions segregate holistic and compositional parts of the process and, similarly, context-dependent and reusable parts of the component characterisations.

The concepts developed and discussed in this chapter have been demonstrated on the practical case study of Aircraft Power Generation and Distribution System (EPDS). The system contains significant redundancy and a large number of possible configurations. Failure logic models were constructed for the EPDS design as provided by author's collaborators; in this context, the modes of the system along with their associated rationale and intent had to be 'forensically' reconstructed. Whilst the modelling process in such a context is labour-intensive and the resultant models are in places complex, it is observed that the task of *systematic rationalisation* of systems reconfiguration rules may bring substantial benefits to the safety assessment processes. Specification of failure logic models results in a qualitatively better understanding of the system design, its limitations and justifications which is likely to add value to the development process as a whole. Further discussion of this issue is provided in the next chapter.

Chapter 6: Evaluation

This chapter presents the evaluation of the thesis contributions in general and, in particular, provides evidence to support the thesis proposition defined in Chapter 1. The first section of the chapter outlines the evaluation strategy. It begins with revisiting the thesis proposition and decomposing it into manageable key hypotheses. An evaluation argument then links each hypothesis to the evidence presented in the thesis. To further aid navigation through the evidence, the evaluation strategy is illustrated by a small number of diagrams presented using the Goal Structuring Notation (GSN) [83, 84] – a graphical notation for representing structured arguments.

Subsequent sections (6.2 through 6.3) present (or, where they have already been presented, summarise) the three main forms of evidence used to support the proposition:

- *Case Studies* (section 6.2) that are used to evaluate technical aspects of the proposition as well as to confirm the overall soundness of the concepts defined in the proposed framework;
- *Metamodel Experiments* (section 6.3) – consisting of the validation of the Failure Logic Metamodel, its instantiation and traces to the concepts of pre-existing safety analysis methods – that are used to confirm the consistency, soundness and completeness of the framework;
- *Peer Reviews* (section 6.5), that are used to further assure the conceptual soundness of the framework as well as to supplement and assure adequacy of the other forms of evidence.

The chapter concludes with the outline of the identified limitations of the failure logic modelling approach that were identified during the research and evaluation. These limitations are common to all failure logic modelling methods and are their identification is a key contribution of the thesis. Wherever applicable, pragmatic mitigations of the effects of these limitations put in place in the course of research are briefly outlined.

6.1 Evaluation Strategy

The proposition defended by this thesis has been defined in Section 1.3 as follows:

> *It is possible to provide a **well-defined, sound** and **pragmatic** framework for the failure logic modelling of realistic systems which tackles the problems of **dynamic reconfiguration** of systems and **composition of interdependent system models**.*

To discharge this proposition it is therefore necessary to show that the proposed FLM Framework (as introduced in Chapter 3, and extended in Chapters 4 and 5) meets each of the criteria highlighted in the proposition above as illustrated in Figure 88.

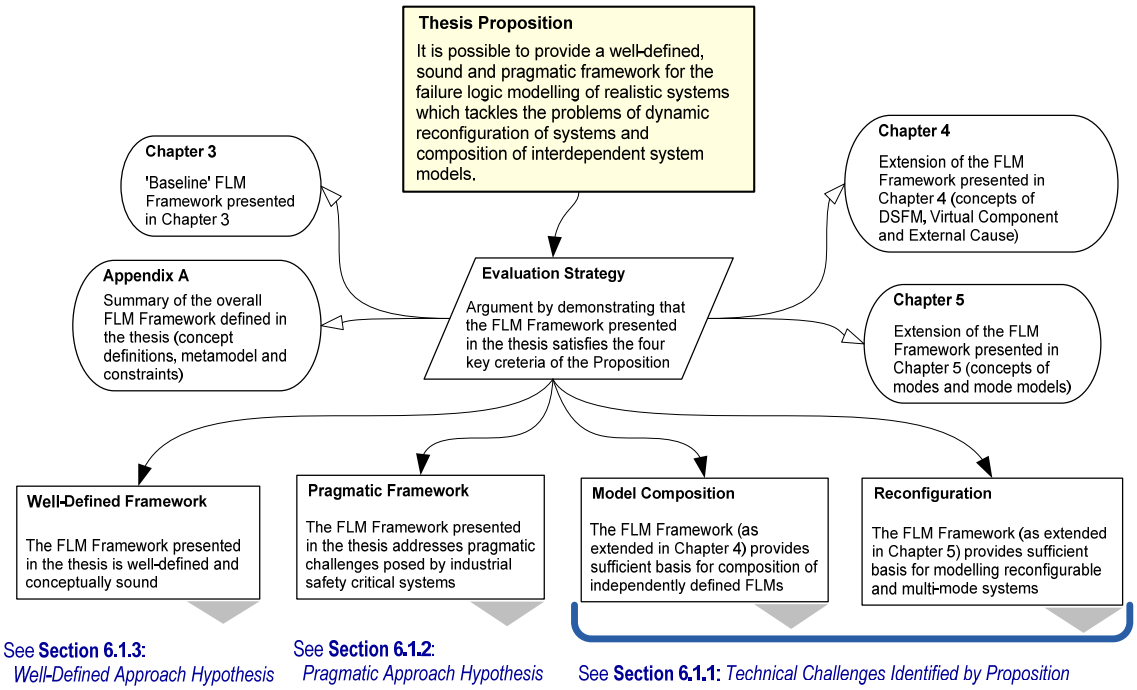


Figure 88 - Evaluation Strategy

Each of the derived hypotheses is addressed in the following sub-sections starting with the last two (addressed together due to their similar technical nature). The technical chapters (i.e. 3, 4 and 5) and Appendix A of the thesis that, respectively, present and summarise the proposed FLM Framework form the context for the evaluation argument (shown in the goal structure of Figure 88 as GSN context elements – the "rounded" rectangles).

6.1.1 Technical Challenges Identified by the Thesis Proposition

The thesis proposition explicitly identifies two challenges posed by large-scale industrial safety-critical platforms: the dynamic reconfiguration of systems and the composition of interdependent system models – described in sections 1.2.1 and 1.2.2 of Chapter 1 respectively. Evaluation of the proposed FLM Framework with respect to these challenges is based primarily on the two case studies introduced in the respective chapters of the thesis. Specifically, the adequacy of the framework with respect to the issues of dynamic reconfiguration is confirmed through the Aircraft Electrical Power Distribution System (EPDS) case study that demonstrated a successful definition of models of such a reconfigurable and multi-mode system. Similarly, the adequacy of the model composition approach was demonstrated through the integration of models of the Wheel Braking System and a Common Aircraft Computation and Communications Platform (for simplicity

referred to as “IMA”). The contribution of these two case studies into evaluation is further summarised in section 6.2 below.

It is important to stress that neither the EPDS nor the IMA case study has been used for development of the FLM Framework itself (i.e. unlike the ARP WBS case study, they are *not formative*). The evaluation based on the case studies has been overseen by the author’s industrial collaborators, providing assurance that the experiments are representative of the challenges found in real industrial systems.

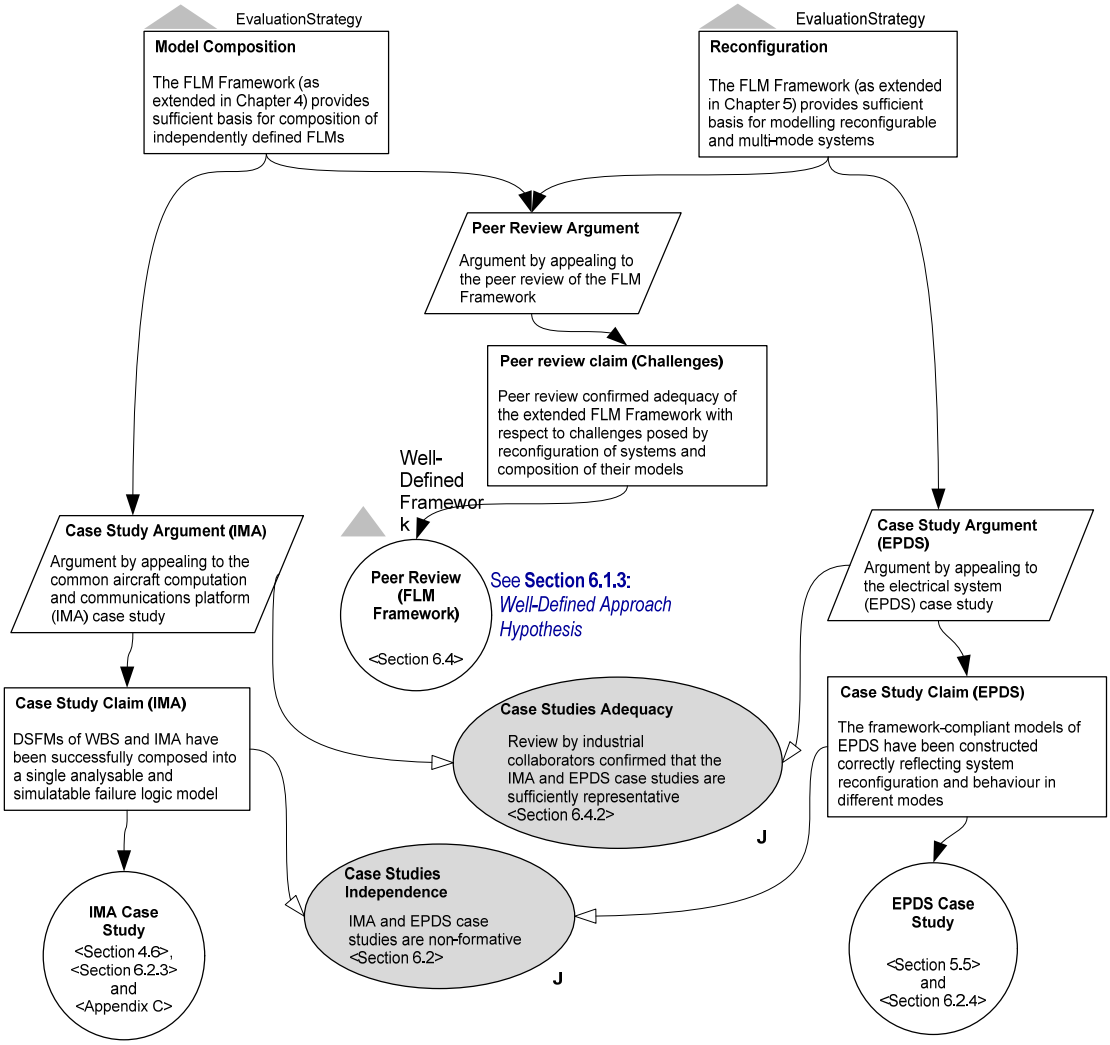


Figure 89 - Technical Challenges Identified by the Proposition (Evaluation Argument)

Finally, to mitigate against any limitations of the coverage of the case studies, the proposed Framework (including the concepts introduced in Chapters 4 and 5) has been directly reviewed by research collaborators in the MISSA and Airbus Dependability Network projects (as discussed in section 6.4 below). Whilst often involving robust discussions, the reviews have confirmed the overall adequacy of the Framework and have identified no significant flaws.

Figure 89 summarises the evaluation strategy with respect to the two technical criteria of the thesis proposition (note that shaded shapes indicate items first presented in this chapter whilst the ellipses and circles of the GSN represent justification of the argument and ultimate evidence it is based upon respectively).

6.1.2 Pragmatic Approach Hypothesis

The development of the evaluation strategy with respect to the criterion of "pragmatism", shown in Figure 90 (page 218), is based on three key branches. Firstly, it is demonstrated that the *FLM Framework adequately addresses key characteristics of realistic industrial safety-critical systems* (beyond the two technical challenges explicitly identified in the previous section). To achieve this, a review of a complete real system – the fuel system of a modern aircraft – has been undertaken in order to identify such characteristics (section 6.2.2). The two non-formative case studies, mentioned in the previous section, have been specifically selected/constructed to provide sufficient coverage of all but one[52] of the identified features (see Table 12 in section 6.2.5). The completeness and adequacy of the set of the identified characteristics and their coverage by the case studies has been confirmed by a review by industrial collaborators (section 6.4 with a summary presented in Table 13 on page 247).

Secondly, the thesis has demonstrated that the proposed Framework is practicable in that the notation-agnostic *FLM Concepts and Metamodel can be adequately and systematically instantiated in a third-party specification language* – the AltaRica OCAS – and analysed (including simulation) by the associated tools. The adequacy of the instantiation (and mapping between the FLM concepts and AltaRica constructs) has been confirmed by research collaborators from ONERA and Dassault Aviation – who have extensive experience with the AltaRica language.

Thirdly, it was demonstrated that the FLM Framework is *systematic* in the sense that it includes *guidance for repeatable construction of the models*. Whilst some of the guidance has been included in this thesis (for example, in sections 4.4 and 5.4), this claim is made predominantly on the basis of a stand-alone "Failure Logic Modelling Handbook" developed by the author and fashioned after the NUREG and NASA Fault Tree Analysis Handbooks [158, 157]. Currently in draft form, the handbook has been reviewed by the industrial and research partners of MISSA project as part of the formal mid-term evaluation cycle and has attracted positive and constructive feedback (see section 6.4.2).

[52] The one "uncovered" characteristic – of the significance of detailed temporal aspects of system failure logic to the safety assessment – has been deemed to have atypically disproportional, if not unique, importance for the fuel system.

Finally, it is important to note that whilst the ability to instantiate the FLM Framework in a specific language is clearly important for the claim that the presented approach is indeed pragmatic, it is also important for the framework to be specified in a notation-agnostic fashion that is not strongly coupled to any specification language (section 1.2.3). This objective is largely achieved 'by construction' and is further addressed in the next section.

6.1.3 Well-Defined and Sound Approach Hypothesis

Discharging the last hypothesis of the Thesis Proposition also relies on three high-level claims (see Figure 91): that the proposed Framework is conceptually sound, that it is internally consistent and that it is not overly constrained by any particular specification language. As was mentioned above the last claim is inherently upheld by the FLM Framework being based on a set of safety engineering concepts and a notation-agnostic metamodel. Given that the Failure Logic Metamodel is formalised in the *Eclipse* and *Epsilon* platforms[53], it was possible to *demonstrate internal consistency* of the metamodel (including its constraints) through automated "validation".

The conceptual soundness of the FLM Framework is arguably the most substantial claim of this part of the evaluation strategy and is demonstrated in a number of diverse ways. First, sections 6.3.2 through 6.3.4 demonstrate that the concepts of the FLMM can be traced to those of fault tree analysis (including its dynamic and non-coherent extensions) and pre-existing failure logic modelling approaches (such as HiP-HOPS and FPTN). Second, it is demonstrated that all of the concepts of the framework have been exercised during the case study (see Table 12 in section 6.2.5) yielding a high degree of confidence that they have valid engineering interpretations. Thirdly, the ability to instantiate the metamodel in the context of a general third-party specification language (discussed in the previous section) demonstrates that the framework is 'complete' (in that it identifies information sufficient for construction of concrete, consistent and analysable models).

Finally, it is important to note that this aspect of evaluation relies most heavily on the peer review of the presented approach. The FLM Framework has been extensively reviewed through both presentations and technical reports by author's industrial and research collaborators in the Airbus Dependability Network and MISSA projects (see section 6.4) as well as by colleagues in the HISE group in the Department of Computer Science at the University of York. These peer reviews confirmed the soundness of the framework as well as consistency of the underlying Failure Logic Metamodel.

[53] See section 6.3 for further description.

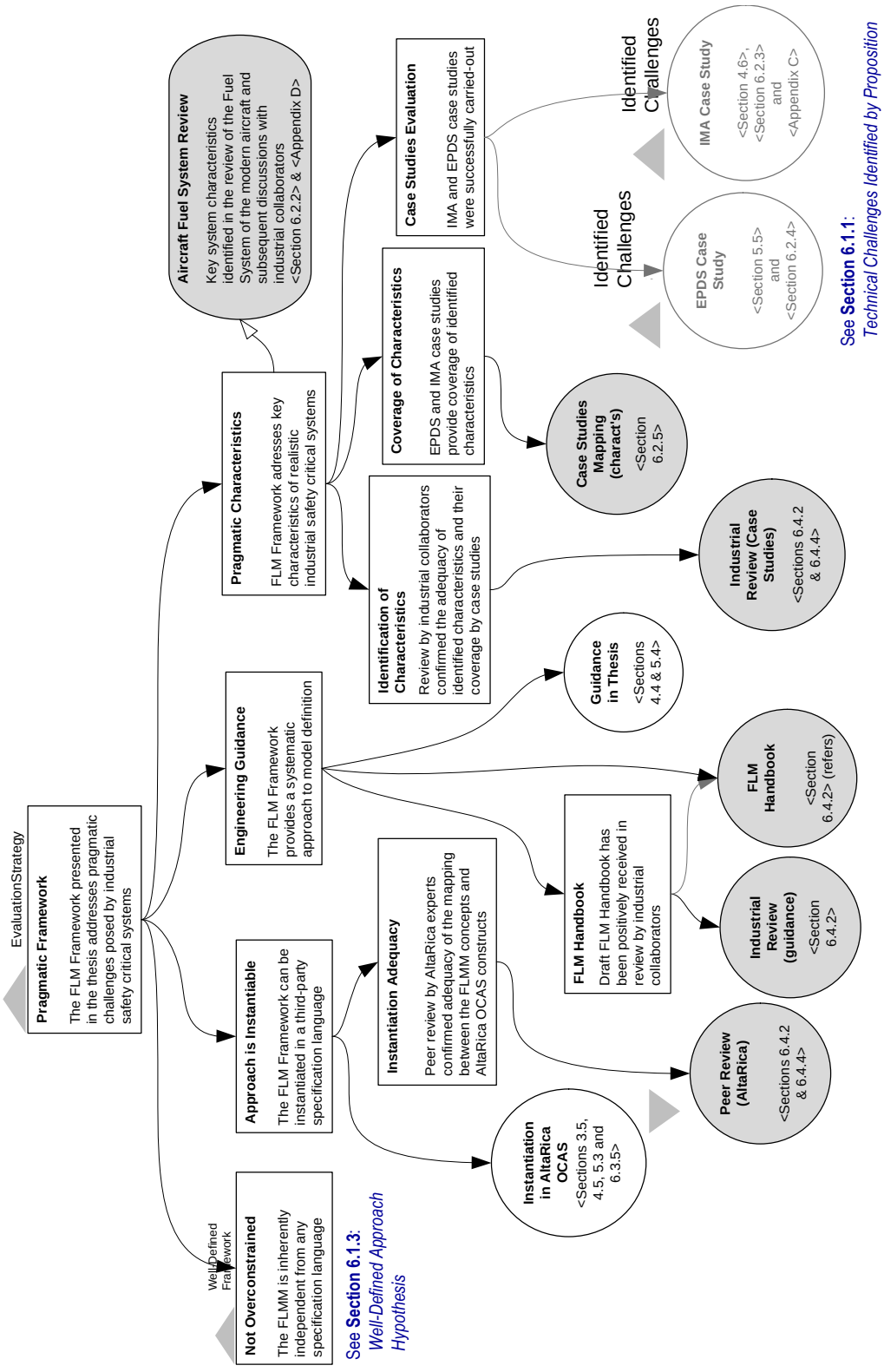


Figure 90 - Pragmatic Approach Hypothesis (Evaluation Argument)

Figure 91 - Well-defined Approach Hypothesis (Evaluation Argument)

6.2 Evaluation through Case Studies

Case Studies were primarily used to evaluate the technical feasibility of the 'baseline' failure logic modelling approach (Chapter 3) as well as the extensions proposed in this thesis for modelling systems with dynamic reconfiguration (Chapter 5) and for the composition of independently-defined failure logic models (Chapter 4). Careful selection of the case studies – based on the review of a real aircraft system (discussed below) – also contributes to assurance that the approach is pragmatic.

Table 11 - Case Studies Coverage (Outline)

Case Study	Technical Thesis Contribution		
	Unifying 'baseline' FLM Framework (Chapter 3)	Composition of independently defined models (Chapter 4)	Modelling reconfigurable systems (Chapter 5)
WBS (ARP 4761)	✓ Formative Evaluation	Small-scale formative evaluation only	✓ Formative Evaluation
EPDS	✓		✓
IMA / WBS (integration)	✓	✓	

Evaluation through case studies was carried out in three steps:

1. At the research development stage the *Aircraft Wheel Braking System* (WBS), as described in ARP 4761[54], was used as a formative case study. This system has emerged as a de-facto benchmark for both model-based safety assessment methods and safety analysis of aircraft systems.
2. Upon completion of the research development phase a single aircraft system – the *Fuel System of a modern aircraft* – was reviewed to identify key characteristics of real industrial safety-critical systems. The fuel system itself has not been modelled (due to the scale and complexity of the system and time-constraints of a doctoral research programme) beyond a number of small-scale partial experiments.
3. Two evaluative (non-formative) case studies have been selected / constructed to provide both the coverage of two challenges – of dynamic reconfiguration and model composition – explicitly identified during the research, on the one hand, and – of the characteristics identified during the Fuel System review (above) on the other. These case studies were concerned with the *Integration of the WBS with a Common Aircraft Computation and*

[54] Note that this was not the simplified system used as an illustrative example in Chapters 3 and 5

Communications Platform (“IMA”) and a hypothetical aircraft *Electrical Power Distribution System* (EPDS).

Overall, Table 11 provides an overview of the coverage of thesis contributions by the three case studies (WBS, EPDS and IMA / WBS integration).

6.2.1 Wheel Braking System

Appendix L of ARP 4761 provides a detailed description of a simplified Wheel Braking System of a hypothetical aircraft. The popularity of this system as a case study renders it a *de facto* benchmark for model-based system safety assessment.

For the purpose of evaluation, the complete description of the system has been further extended in this thesis with causes and effects of hydraulic leaks. The resultant case study is sufficiently complex to demonstrate all relevant concepts of the FLM Framework introduced in Chapter 3 and 5 with the sole exception of *Failure Handling States*.

This case study was also used to assess the *pragmatic importance* of the concept of modes to the failure logic modelling approach. The system has been progressively simplified until it could be described without utilising concepts introduced in Chapter 5 of the thesis (this corresponds to the expressive power of existing methods such as FPTN, FPTC and most variants of HiP-HOPS). The resultant system was grossly inadequate: a selector valve between the blue and green hydraulic channels had to be removed, the BSCU had to be limited to a single channel and the functionally-passive connector between BSCU `CMD` output and Green Meter Valve had to be assumed as not being susceptible to any failures.

It is important to note that the WBS case study has been used throughout the research for the *formative evaluation* of the proposals. Because of this, further case studies were conducted *a posteriori* to ensure acceptable degree of independence of evaluation activities from those of research development.

6.2.2 Aircraft Fuel System

Whilst the pragmatic focus of the thesis proposition requires case studies to be closely linked to real industrial systems, the complete modelling and analysis of even one such system requires resources beyond those available in the course of doctoral studies. To resolve this trade-off a two-stage approach to the evaluation of the non-formative case study has been taken. In the first stage (described in this section) a complete aircraft system has been reviewed to identify key general features of modern industrial safety-critical systems. In the second stage (described in the

following two sections) smaller-scale case studies were selected (in case of EPDS) or constructed (in the case of the WBS / IMA integration) to provide adequate coverage of the identified features. As outlined in the Evaluation Strategy and described in Section 6.4 below, the integrity of the process and adequate coverage of the pragmatic challenges by the case studies have both been verified through peer reviews by industrial collaborators.

The reviewed system was the Fuel System of a modern aircraft. The author was provided with access to draft design documents by Airbus Operations on the company's site in Filton, UK during a five week visit. The review identified the following five key characteristics of the fuel system:

(1) Complex mode logic
(2) Intentional architectural limitations
(3) Circular dependencies and loops
(4) Complexity of scale and design decomposition
(5) Time-dependency and reliance on consumable resource

The first four characteristics have been considered as being typical for a large number of aircraft systems and are briefly summarised below. The fifth characteristic, however, was judged – in consultation with industrial collaborators – to be unique in its prominence to the fuel system and was not addressed by the case studies. The fuel system and all five identified characteristics are described in more detail in Appendix D.

Complex mode logic

The fuel system is typically operated through a number of phases and is highly reconfigurable (e.g. upon failure). Whilst it may at first appear that the failure-handling modes of the system (termed "workarounds") can be seen as sub-modes of phases of operation, a closer examination of the mode model reveals that this is not the case. In particular, switch-over between the phases may be affected by the active workarounds. For example, certain workarounds (entered upon detection of equipment failures) may inhibit the non-essential phases of operation (such as load alleviation fuel transfers prior to landing). Similarly, many workarounds affect the manner of execution (the exact physical paths used by different types of fuel transfers) of more than one phase of operation. This means that a structured mode model advocated by Papadopoulos in [115] is not applicable to this system: with direct transitions between sub-modes of different modes, the hierarchy of mode models is likely to be a purely decorative facility (since high-level modes will not have a unique initial sub-mode). The complexity of the mode model of the fuel system is further increased by the presence of some orthogonal sub-models (some workarounds are not mutually exclusive and "side in control" modes of the fuel management sub-system are largely independent from the workarounds of the liquid-mechanical architecture).

Intentional Architectural Limitations

In addition to the modes outlined above the fuel system can be operated in a *manual mode* with pilots controlling transfers directly through the cockpit interface. The manual mode can be initiated by the pilots at any time regardless of the system status; in addition, it is entered upon FQMS detecting an unhandled combination of two or more failures. What makes this manual mode remarkable is that the pilots do not have full control of the system. Certain transfers (or, to be precise, certain transfer *paths*) are intentionally made unavailable in manual mode. Thus, even in circumstances when a healthy transfer path exists in principle and can be utilised in automated operations, effective manual operations may be impossible. This 'intentional sacrifice' of some of the functionality cannot be justified in terms of the system model. Instead, the justification comes from a perceived trade-off between fuel transfer availability in rare circumstances (when manual mode is required) and the need to minimise the likelihood of pilot error (that is increased by complex control tasks).

Complexity of Scale and Design Decomposition

As was indicated above the fuel system is divided into two sub-systems: the Liquid-Mechanical Subsystem and the Fuel Quantity Management Sub-system (FQMS) – designed in relative isolation by different engineering organisations. Simplified partial failure logic models of the two subsystems have been constructed during the review process. These experiments have indicated that at *early* stages of the development two *sub-systems can be meaningfully analysed in their own right*. For example, the analysis of the liquid-mechanical architecture can identify failures that may lead to fuel becoming 'unusable' (either by being isolated in reserve tanks or through leaks). Analysis of the FQMS could identify failure scenarios that lead to the management subsystem inadvertently commanding transfers, erroneously commanding transfers on compromised paths or failing to command transfers when needed. The models of the sub-systems, however, could not be effectively directly composed through input and output FM flows. Instead the composition would require definition of complex virtual components to explicitly represent the concept of "fuel transfer"[55]. This finding highlighted the utility of virtual components in the integration of independently-defined large-scale models.

In addition to the composition of sub-systems models, other likely model integration scenarios were identified during the review process. Firstly, the FQMS contains a number of software partitions implemented on the IMA. The complete safety assessment should clearly take into account the effects of IMA failures on the ability to continuously provide fuel to aircraft engines. Secondly, review of the fuel system documentation has uncovered a case when the same set of components (the liquid-mechanical architecture) is considered by *separate safety analyses* from

[55] This is broadly similar to the "virtual link" components that were later used for composition of WBS and IMA (see Section 4.6.4)

two different viewpoints: the ability to support effective transfers and the loss of fuel through leaks. This is a clear case of two engineering domains being defined over the same scope but from different viewpoints (see Table 5 in section 4.3.1, page 129). This observation has provided further evidence of the necessity of a *flexible* conceptual framework for rationalisation of platform (de)composition (as advocated in Chapter 4).

Circular Dependencies and Loops

Whilst the problem of loops in failure logic models is not new, review of the fuel system along with preliminary partial experiments have provided a new insight into this issue. Fuel system models contained two types of loops: model loops due to control feedback (e.g. closed-loop control schemas) and strong circular dependencies of abstract models. Control feedback loops do not always result in circular dependencies in failure logic models. Furthermore, when such dependencies are established they can be relatively easily resolved in the context of 'forward search' model analysis. The issue of strong circular dependencies is, however, significantly more complex. The nature of these dependencies is described in more detail in Appendix D. Broadly speaking they are analogous to situations when zealous application of fault tree construction rules would result in a non-terminating construction process.

In failure logic models these circular dependencies cannot be trivially addressed by the introduction of an infinitely short delay or the utilisation of a fixed point search algorithm – as the analysis may stabilise in an *incorrect* equilibrium. Instead such dependencies can only be correctly resolved by enriching the models with a more complex loop resolution behaviour. Whilst having no direct engineering interpretation, this injected behaviour can be informally seen as an attempt to 'shake' the model out of a false local equilibrium.

6.2.3 Integrated Modular Avionics

This case study, concerned with integration of the DSFMs of the Wheel Braking System and the shared aircraft communications and computation platform, was designed specifically to evaluate the proposed approach to model composition in contexts other than the classical containment hierarchy view of decomposition of large scale engineering artefacts ("platforms"). As such the case study addresses one of the key characteristics of realistic systems – complexity of scale and design decomposition – identified in both the Thesis Proposition and the review of the Fuel System.

The case study was constructed based on the following three objectives:

(1) To evaluate feasibility of the composition of two DSFMs that cannot be reasonably expected to explicitly identify cross-model dependencies in terms of FM flows;

(2) To evaluate the feasibility of the composition of two DSFMs where semantic / conceptual heterogeneity between typical views of the systems renders the allocation domain[56] non-trivial (that is where effective and clear composition of failure logic models is likely to require the introduction of "virtual components");
(3) To evaluate, as far as practicable, the applicability of the composition approach to scenario(s) where causal dependencies between models do not have identical direction (and, thus, fall beyond the expressive power of typical Common Cause Analyses).

It is important to note that evaluation of the applicability of the FLM Framework to the safety assessment of Integrated Modular Avionics was *not* the objective of the case study. Consequently, it was decided to significantly simplify the model of the infrastructure to the bare minimum necessary for evaluating feasibility of model composition. The IMA model was constructed to be *broadly* in line with the ARINC 653 specification [7], the safety assessment performed by Conmy [34, 36] and descriptions of various aspects of the system found in the public domain [1, 4].

This integration scenario naturally addresses the first two objectives of the case study:

- It is unrealistic, especially at the earlier – conceptual – stages of design, to expect safety engineers working with the software engineers responsible for 'subscriber' systems (e.g. WBS) to explicitly identify various dependencies of the software architecture on the computation platform.
- The dissimilarity of typical viewpoints adopted in the design (and analysis) of subscriber systems and the platform itself requires the introduction of virtual components to model concepts of Partitions and Virtual Links explicitly.

To address the third objective within the same case study a design assumption has been introduced stating that:

a) Anti-skid and braking command calculations performed by the *COM* components of BSCU channels are implemented as two pseudo-parallel software processes in the WBS;
b) The scheduling of processes in the same partition utilises a co-operative (rather than a fixed-time) schema.

The assumption enables a 'covert' causal dependency path between otherwise apparently independent failures of the *COM* component; the path is only revealed in integration of the WBS and Infrastructure failure logic models.

Overall, the case study has demonstrated the technical feasibility of the proposed approach to the composition of failure logic models. The composed models could be both analysed and simulated.

[56] See Section 4.3, Chapter 4

In terms of the composition process, the notion of “virtual components” has allowed capturing cross-model dependencies in a clear and structured fashion. Further, the introduction of these components provided a means for addressing the issues of allocation (mapping) of partitions and virtual links and the nature (i.e. logical structure) of these dependencies in relative isolation. For example, reallocation of the software components and communication links to different physical IMA equipment does not affect the structure of the virtual components; similarly, changes to the platform’s communication protocol would not affect the dependencies between the WBS failure logic model and the virtual VL components. Finally, the highly regular structure of interface and internal logic of the virtual components (VLs and partitions) allowed for the composition of failure logic models to be performed semi-automatically, demonstrating the feasibility of fully automated integration of models based on allocation databases.

Whilst successful overall, the case study has identified some significant limitations at the level of the specification language and analysis tool. Namely, the lack of weak-directed synchronisation in AltaRica required models to be post-processed prior to composition. Also, it was discovered that the Cecilia OCAS sequence generator does not currently treat ‘temporal’ *Dirac(x)* events in a consistent fashion. This temporal law has been extensively used in the case study to mark Normal Events (used in both the IMA DSFM and the “translation layer” to resolve non-deterministic effects of failures and failure modes). Whilst the latter problem does not affect model simulation, it does require the sequence generator to be set to perform search to a very large cardinality (which is prohibitively time consuming).

6.2.4 Aircraft Electrical Power Distribution System

The final case study undertaken as part of the research reported in this thesis is the Aircraft Electrical Power Distribution System (EPDS). The main objective of the case study was to evaluate the technical feasibility and adequacy of the FLM Framework extensions introduced in Chapter 5 of the thesis. In addition to this, the case study was selected specifically to address the following characteristics identified in the review of the fuel system:

- Complex mode logic containing a number of orthogonal mode models;
- Intentional architectural limitations;
- Circular dependencies and loops.

The system description has been developed as part of the MISSA project by ONERA and Airbus Operations (France). Although the system is simplified, the author had no part in making the simplification assumptions, thus, ensuring a high degree of independence between the evaluation and the research development.

From the basis of the system description, a ‘hierarchy’ of models have been constructed in an *incremental* fashion. The initial failure logic model (EPDS #1) included the electrical network only (i.e. without its associated controllers and modes of operation) and was concerned with the propagation of electrical power (and associated FMs); this system did not include the causes or effects of short circuits allowing a simple global vocabulary of failure mode classes (i.e. omission and commission of electrical power). The next model of the hierarchy (EPDS #2) has revoked the ‘freedom from short circuits assumption’. In addition to introducing circuit breaker components and the short-circuit failure modes, it was necessary to refine the power FMs into two groups of FM classes – concerned with provision of electrical current and voltage – in order to model short circuit propagation and effects accurately.

It is important to note that the first two models of the hierarchy included a number of loops. Most notably, the two cross-feed lines between the two main sides (see Figure 92 below) of the system create strong dependencies between junction components which could not be resolved by the propagation delay approach. In particular, such a naïve approach results in a system reaching a wrong equilibrium after the failure of all generators (that clearly results in a total loss of electrical power provision) since the *absence* of the failure mode (modelled by the `OK` privative) gets ‘locked’ inside the loop. This loop had to be resolved through injection of a transient omission FM (in both directions) upon failure of any generator.

Whilst the first two models of the EPDS hierarchy do not allow for the analysis of the system behaviour per se, they can be used to assess the fundamental adequacy of the physical system architecture (before development of the controllers commences).

The models at the subsequent levels of the EPDS hierarchy have progressively enriched the failure logic models with mode models. The first EPDS mode model (applicable to the entire system) included only two modes: the initial Non-Emergency mode (where the system is powered by non-emergency generator(s) and all busbars are expected to be powered) and the Emergency Mode (whereby the system is only powered by the Emergency Generator and only two essential busbars are expected to deliver electrical current). As discussed in Chapter 5, the Emergency Mode is an example of both a *degraded mode* and an *intentional design limitation* since, although the physical architecture of the system often makes it possible to deliver electrical power to non-essential busbars, the design stipulates that – in order to protect the last remaining ‘live’ generator – only the essential equipment should be powered.

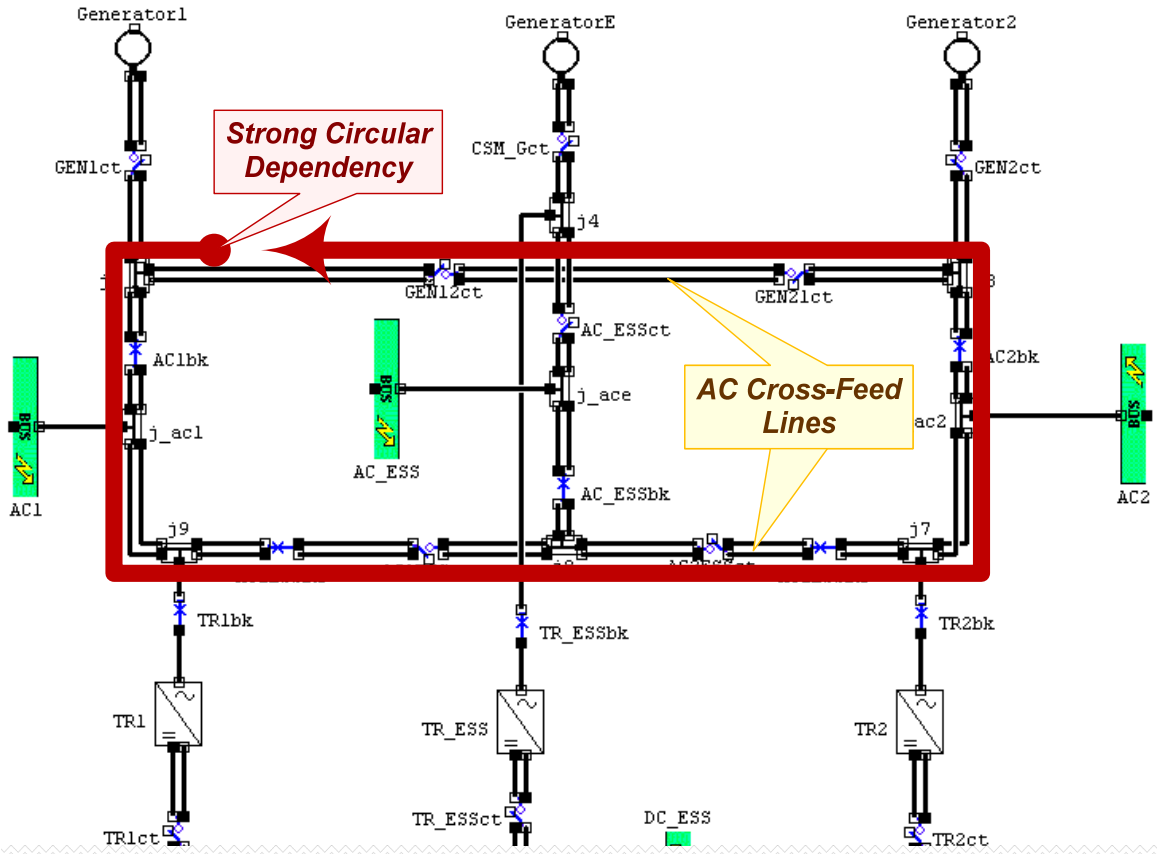


Figure 92 - Strong Circular Dependency in EPDS Models

Further models of the hierarchy have introduced mode models related to the AC generators (ACG), the AC cross-feed (XF) and the DC parts of the system (see Figure 93 below, reproduced from Section 5.5.2) and corresponding controllers. It is important to make two observations here. First, these three groups of modes are largely orthogonal and yield a relatively complex overall mode model. Second, introduction of the controllers increases the number of loops in the EPDS and, in particular, introduces control loops.

The above discussion demonstrates that the EPDS case studies have addressed all three outstanding system characteristics identified in the review of the aircraft fuel system:

- Complex mode logic – through the presence of three orthogonal groups of system modes along with the overarching Emergency / Non-Emergency mode model;
- Intentional architectural limitations – through the presence of an Emergency Mode, where provision of power to non-essential busbars is intentionally sacrificed;
- Circular dependencies and loops – through the presence of strong circular dependencies in the abstract / early EPDS models as well as feed-back loops of more detailed models.

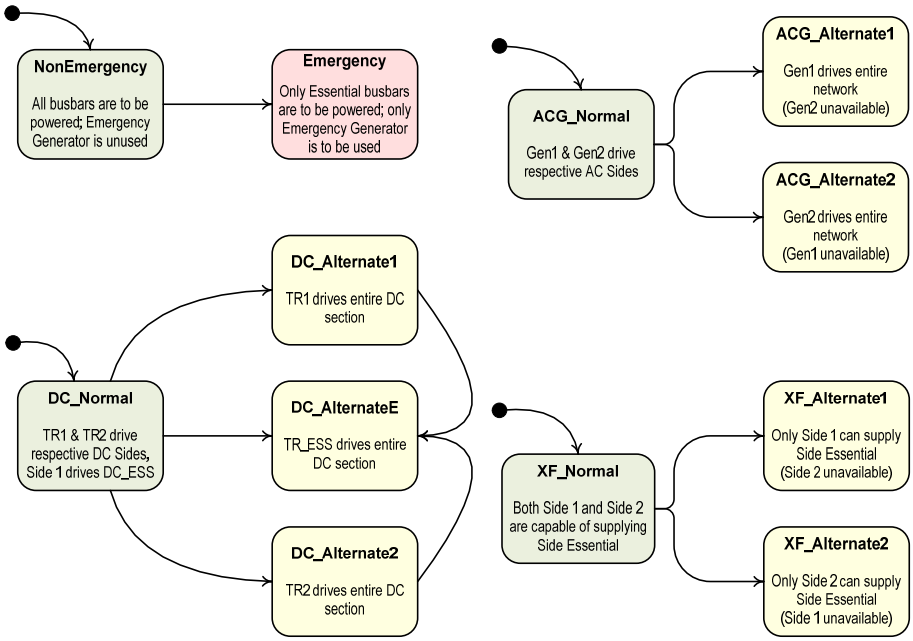


Figure 93 - Outline of the Overall EPDS Mode Model

The EPDS case studies also provided an additional opportunity to evaluate applicability of most of the basic concepts of FLMM[57] including – most importantly – the notion of failure handling states (due to importance of the short circuits and, consequently, the protection provided by circuit breakers) which was not covered by the formative WBS case study.

6.2.5 Case Studies Summary

Overall the case studies contribute to three areas of the evaluation strategy:

a) Non-formative EPDS and WBS / IMA integration case studies provide direct assurance that issues of model composition and reconfiguration (explicitly identified in the thesis proposition) are adequately addressed;

b) Coverage by these two case studies of the key characteristics identified during Fuel System review provides confidence that the FLM Framework addresses pragmatic concerns of real industrial safety critical systems;

c) Coverage of all concepts of the FLMM by the three case studies ensures that all concepts have valid engineering interpretations and, thus, yields confidence in the conceptual soundness of the framework.

Coverage by the three case studies of technical contributions of the thesis, key characteristics of the fuel system identified in the review and concepts of the baseline FLM Framework are summarised in Table 12.

[57] With exception of Normal Events and Normal States – which were extensively used in the IMA/ WBS integration case study (as discussed in the previous section).

Table 12 - Coverage Achieved by the Case Studies

Evaluation Subject		Case Study		
		WBS *(formative)*	**EPDS**	**WBS / IMA Integration**
'Baseline' FLM Framework	Basic FLM concepts: Components, Failure Modes, FM Flows, FM Classes, FM Groups Failures and Failure States	Yes	Yes	Yes
	Void Transition triggers (incl. FM-caused failure state transitions)	Yes	Yes	Yes
	Failure Handling States	No	Yes	Yes
	Normal Events and Normal States	Yes	No	Yes
Chapters 4 & 5 Extensions	Engineering & Allocation Domains, Domain Specific Failure Logic Models (DSFMs)	Small-scale experiments only	No	Yes
	External Causes [of failure]	Small-scale experiments only	No	Yes
	Virtual Components	Yes	No[58]	Yes
	Modes & Mode Spaces	Yes	Yes	No
Fuel System Characteristics	Complex mode logic (orthogonal modes)	n/a	Yes	No
	Intentional architectural limitations	n/a	Yes	No
	Circular dependencies and loops	n/a	Yes	Yes
	Complexity of scale and design decomposition (non-trivial integration)	n/a	No	Yes

6.3 Metamodel Experiments

The FLM Framework presented in this thesis is based on a set of safety engineering concepts and a metamodel. Definitions of concepts provide an *engineering semantics* to the framework, whilst the metamodel specifies the relationship between the concepts. The language-agnostic format of the metamodel ensures that the FLM Framework is not unduly influenced by the constraints of any specification notation and, therefore, inherently provides some confidence in the internal consistency and conceptual soundness of the approach.

[58] In EPDS case study virtual components – Mode Observers – are only necessary at the level of the FLM Framework instantiation in AltaRica OCAS and therefore do not contribute to the FLM Concepts evaluation

In addition to this 'confidence by construction', some direct investigations into the FLMM have been undertaken to provide objective evidence of its internal consistency, completeness and soundness. These experiments are reported in the following sub-section.

6.3.1 FLMM Validation in Eclipse

The Failure Logic Metamodel is formalised in the Eclipse Modelling Framework (EMF) [46, 142] – an open-source environment dedicated to the development of tools based on a structured data model (metamodel). Within Eclipse, the FLMM (metaclass diagram) has been specified using Emfatic – an EMF based editor that provides a "compact and human readable syntax" [47]. A short extract from the FLM Model specification – that defines *Component*, *Complex Component* and *Basic Component* metaclasses – is shown in Figure 94. The editor also provides facilities for syntax checking and for the conversion of specifications into an Eclipse core XMI format – "Ecore". These models, in turn, can be automatically visualised (see Figure 95 for an extract) and, more importantly, "validated" (or, more precisely, checked for internal inconsistencies).

```
package FLM_metamodel;

// ********************************
// * Model Hierarchy Structure *
// ********************************

abstract class Component {
  attr String ID;
  attr Boolean IsVirtual;
  val OutputFM[*] #of exhibits;
  val InputFM[*] #of sensitiveTo;
  val FMGroup[*] #definedIn defines;
  val NormalEvent[*] #affects affectedBy;
  ref ComplexComponent[0..1] #contains definedIn;}

class BasicComponent extends Component {
  val Failure[*] #damages damagedBy;
  val FailureStateSpace[*] #represents hasFailureStateModel;
  val NormalStateSpace[*] #represents hasNormalStateModel;
  val FailureHandlingStateSpace[*] #represents hasFailureHandlingStateModel;}

class ComplexComponent extends Component {
  val Component[+] #definedIn contains;
  val FMFlow[*] #enabledBy enables;
  val ModeSpace[*] #represents hasModeModel;}
```

Figure 94 - FLMM Extract (Emfatic)

The metaclass diagram alone, however, does not provide a complete FLMM specification and has to be supplemented by a number of well-formedness constraints. An example of such a constraint would stipulate that the only component in a failure logic model which is not necessarily contained within some complex component is a DSFM; another example would require every state space of a component to contain exactly one initial state.

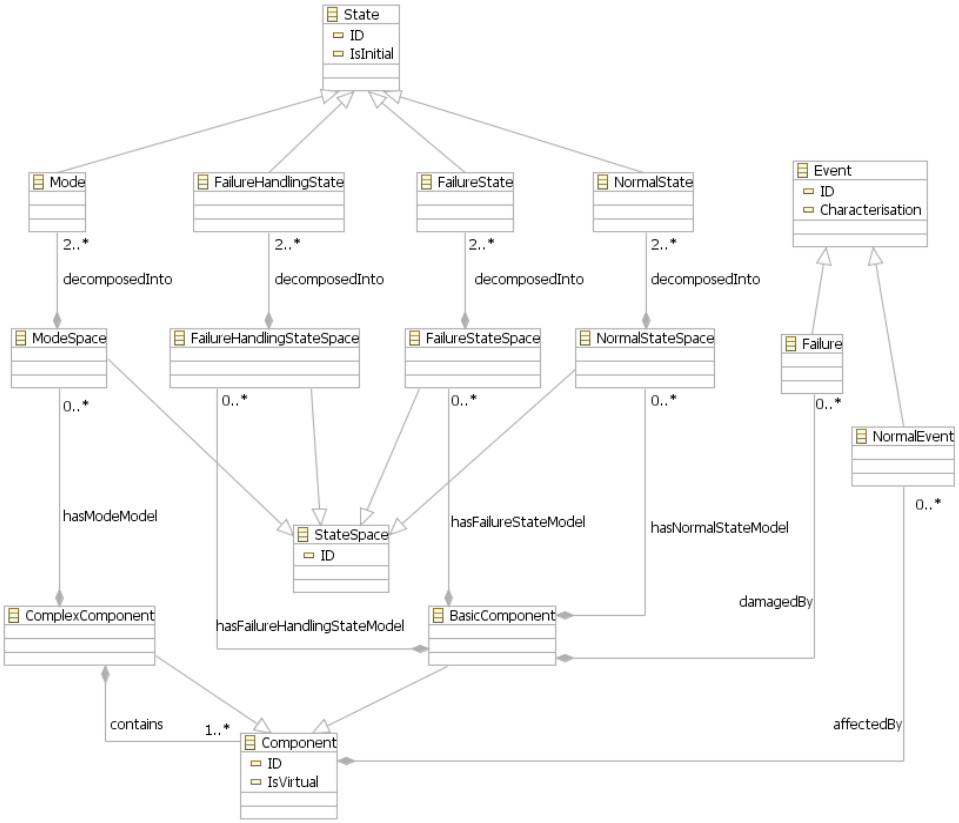


Figure 95 - FLMM Extract (Ecore Diagram)

The constraints of the FLMM are specified in the Epsilon Validation Language (EVL) – part of the Epsilon Framework [45, 86] – which has an intuitive syntax and can be easily related to the Object Constraint Language (OCL). Unlike the OCL, the EVL enables categorisation of constraints as either 'hard' or 'soft' (the latter being called "critiques"); Figure 96 shows an extract from the EVL listing for the two hard constraints of the FLMM mentioned above, along with a soft constraint which highlights an unusual and suspect situation when a non-initial state is not associated with at least one transition specification.

```
context ComplexComponent{
    // Only a DSFM can be an "orphan"
    constraint OnlyDSFMroot{
        check: (not self.definedIn.isDefined()) implies self.isTypeOf(DSFM)
}   }
context StateSpace{
    // Each state must contain exactly one initial state
    constraint InitialStateIdentified{
        check: self.decomposedInto.select(S: State | S.IsInitial).size() = 1
}   // It is unusual for a non-initial state not to have at least one transition
    critique AvoidOrphanStates{
        check: self.decomposedInto.forAll(S: State |
          (not S.IsInitial) implies S.enteredThrough.size() > 0)
}   }
```

Figure 96 - FLMM Constraints Extract (EVL)

The Epsilon Framework provides facilities for automated syntactical checks of the EVL constraints and for checking consistency between the constraints and the Ecore metamodel specification.

Further validation of both parts of the FLMM was carried out through testing on the basis of a simplified component (the BSCU of the WBS case study). The component failure logic has been specified in the OMG's Human-Usable Textual Notation[59] (HUTN) [110] and automatically converted into an Eclipse model. The conformance of this model to the metaclass diagram and EVL constraints could then be automatically checked using functionality provided by Epsilon. The component model has been systematically mutated (including intentional introduction of errors) to ensure appropriate testing coverage.

6.3.2 Mapping between FLMM and Existing Failure Logic Modelling Methods

The thesis claims that the presented FLM Framework subsumes existing failure logic methods such as HiP-HOPS and FPTN. In this section the claim is substantiated.

6.3.2.1 HiP-HOPS

The conceptual metamodel of the 'core' HiP-HOPS method is shown in Figure 97. For brevity, this metamodel ignores some of the extensions such as negation [135] and PANDORA operators [161], integration with design models [116] and the hierarchical mode model of the method (only described by Papadopoulos's thesis[60] [115]).

In terms of model architecture (i.e. of component hierarchy and connectors) the HiP-HOPS metamodel trivially refines the Metamodel presented in this thesis. The only point that is worth noting in this regard is a minor mismatch between HiP-HOPS concept of *Flow* and the FLMM concept of *FM Flow*. Failure Modes in HiP-HOPS are strongly linked, semantically, to flows in design models: FMs are always associated with the design flows and only propagate in the same direction as phenomena in design models. The validity of such a strong constraint is, however, questionable. For example, the EPDS and WBS case studies have demonstrated that dependency paths established between components (e.g. by wiring, pipes or communication protocols) may, in the presence of failures, enable FM flows in *both* directions. A HiP-HOPS "flow" is therefore a

[59] It is important to point out that although HUTN is indeed an intuitive notation that could in principle been used for specification of failure logic models throughout this thesis and in Appendix B, it tends to produce voluminous specifications. For instance, the specification of failure logic of a single BSCU side fully compliant with the FLMM takes over 1500 lines (although a large number of those are trivial).

[60] See discussion in Chapter 5.

constrained specialisation of both the FLMM's FM Group and FM Flow concepts; consequently, if a HiP-HOPS model is to be translated into an FLMM-compliant one, such paths have to be trivially expanded into individual FM Flows.

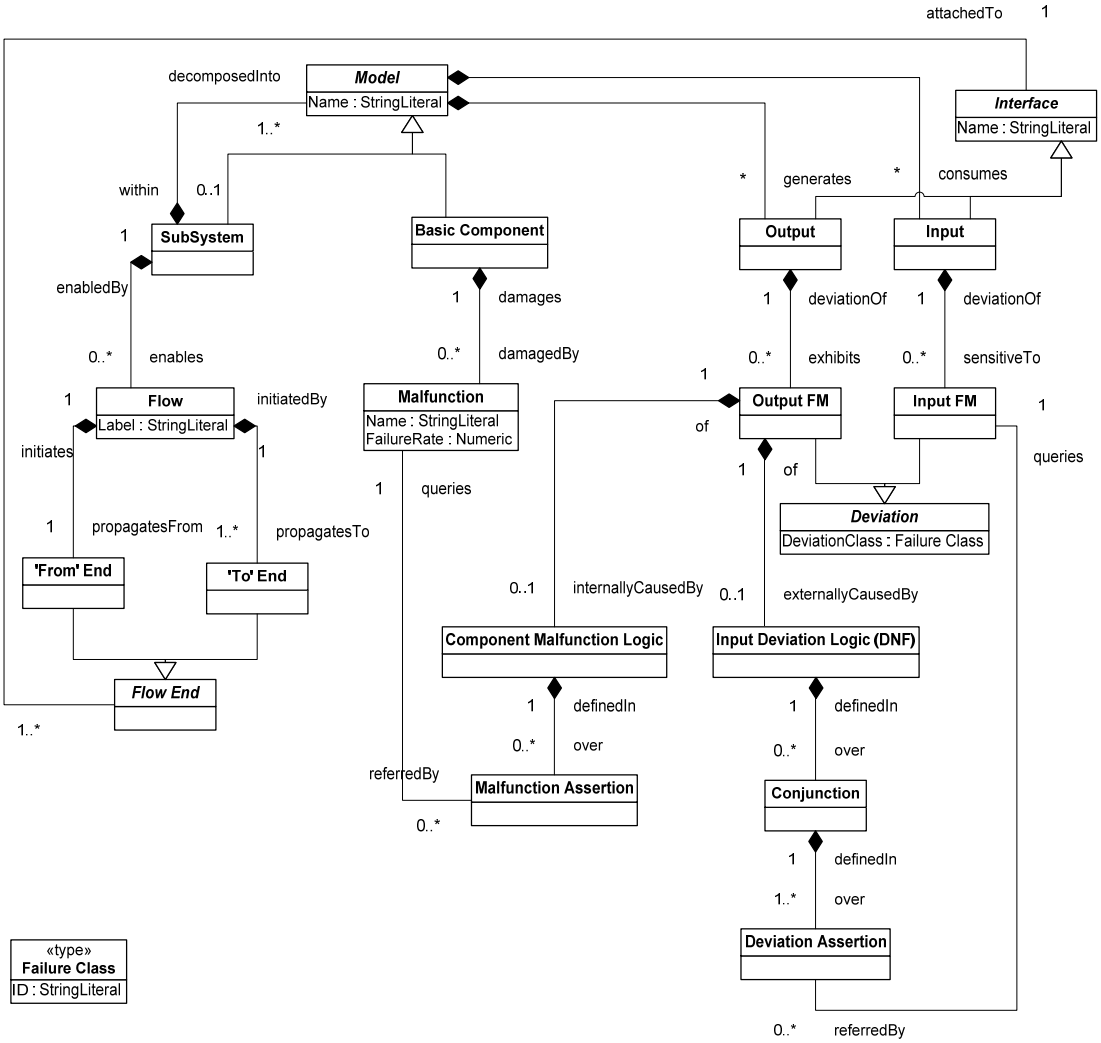


Figure 97 - HiP-HOPS Metamodel ('Core' Method Only)

At the level of basic components, the relationship between the metamodels is also straightforward as illustrated in Figure 98. However, as was mentioned previously, HiP-HOPS does not make a distinction between *Malfunction* as an event and as a state of the component. Furthermore, all *Malfunctions* of a component are considered to be independent from each other. In FLMM terms, this means that each malfunction represents a simple *Failure State Space* as well as a *Failure* itself. The state space consists of two states: an initial privative state and a "failed state". There are no *Transitions* associated with the initial state (i.e. the Failure Logic Metamodel is constrained by the 'once-failed-always-failed assumption'); the failed state can be entered through exactly one *Transition* that is triggered by the *Failure* (also associated with the HiP-HOPS *Malfunction*, as mentioned above) and is predicated on the *Guard* that is always *true* (i.e. it contains no *Conjunctions*). For example, a HiP-HOPS Malfunction, called *Malf*, can be algorithmically

transformed into a basic state model shown in Figure 99 along with an FLM *Failure* (called "Malf" and assigned an identical failure rate to that of the original malfunction).

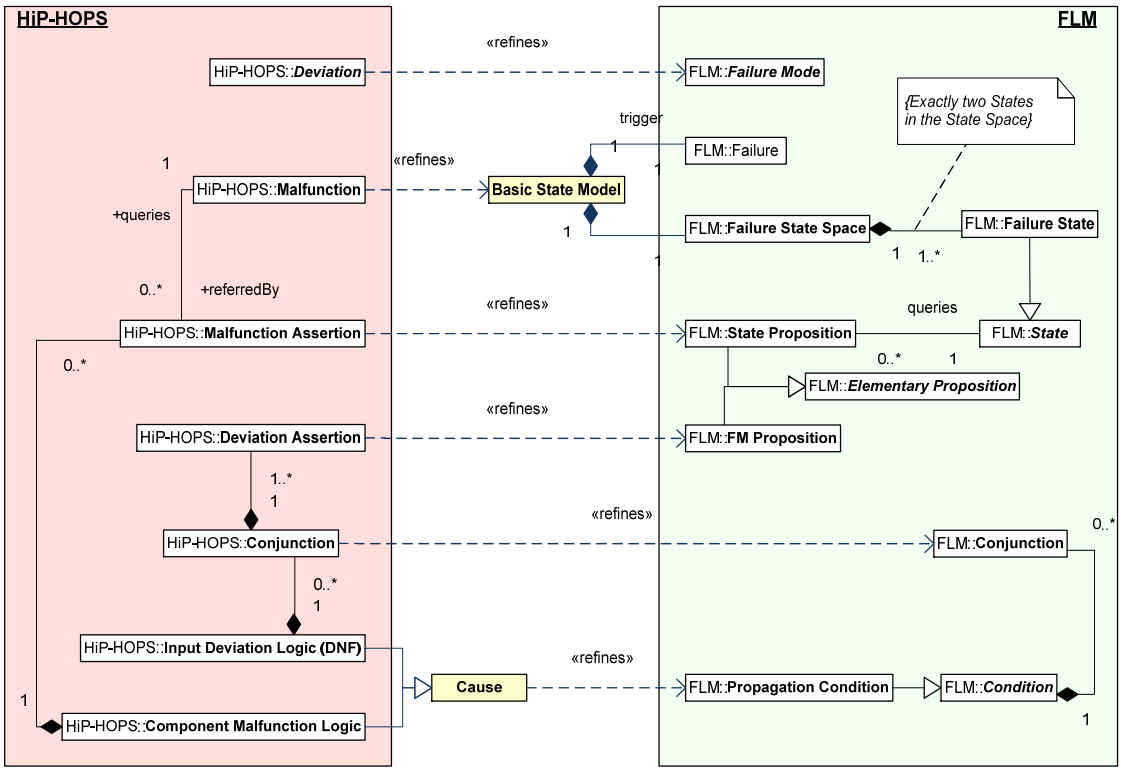


Figure 98 - Relationship between HiP-HOPS and Failure Logic Metamodels

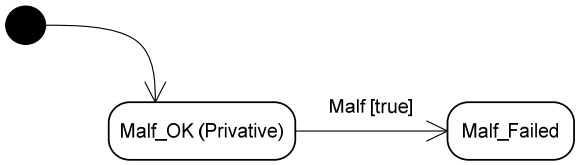


Figure 99 - Basic Failure State Space Model (FLM) for a HiP-HOPS Malfunction "Malf"

6.3.2.2 FPTN

Whilst Failure Propagation and Transformation Notation (FPTN) is broadly similar to HiP-HOPS, the lack of a published systematic definition of the method does not permit a detailed discussion of the constraints it imposes on the Failure Logic Metamodel. However, it can be noted that by contrast to HiP-HOPS the notation does not provide facilities for grouping Failure Modes (except through naming conventions) and doesn't impose constraints on the relationships between FMs and design flows.

In terms of failure, failure states and state spaces the notation offers a standard *GENERATED BY* construct. Public-domain descriptions of the construct allow two interpretations:

(1) As with component malfunction logic in HiP-HOPS, *A GENERATED BY B* construct specifies a causal relationship between a single output failure mode *A* and a failure *B*. This is identical to the HiP-HOPS approach.

(2) *A GENERATED BY B* construct specifies a causal relationship between a failure state *A* and a failure *B*. This is a liberalisation compared to HiP-HOPS, and permits the specification of 'mixed' propagation equations over both input failure modes and internal failure states. However, since the construct does not provide a facility to qualify the transition with a guard, the HiP-HOPS restriction on the shape of the state spaces of components still applies.

Finally, publications on FPTN also mention another construct – *HANDLED BY* – which is used for specifying a mechanism that prevents propagation of a certain input failure mode on the component. The FLMM readily provides facilities for specifying the non-propagation of failure (although by omission rather than by explicit specification). What is interesting to note is that FPTN considers the protection mechanism both 'atomic' and not susceptible to failure. This assumption is revoked in the FLMM in an intuitive and structured fashion by inclusion of the concept of Failure Handling State (and the corresponding state space) as discussed in Chapter 3 and extensively illustrated by the EPDS case study (Chapter 5).

6.3.2.3 Summary: HiP-HOPS as a Set of FLMM Constraints

Discussions in the previous two sections have demonstrated that any HiP-HOPS or FPTN model can be transformed into a failure logic model fully compliant with the Failure Logic Metamodel. The reverse is not true, however: the FLMM exhibits a greater expressive power than either method. Listing of the additional FLMM constraints that are implied by HiP-HOPS both serves as a concise summary of the above discussion and provides a useful insight into the contributions of this thesis.

The constraints imposed by HiP-HOPS are as following:

(i) A model must contain exactly one DSFM;

(ii) Each FM Group may contain a set of Input FMs or Output FMs, but not a combination of both;

(iii) No FM Group may contain more than one Failure Mode associated with the same FM Class;

(iv) If an FM Flow between two Failure Modes exists, then FM Flows between all Failure Modes in respective groups must also exist;

(v) All basic components must have no Normal Events, Normal State Spaces or Failure Handling State Spaces;

(vi) Each Failure State Space (if any) must contain exactly two Failure States. One of these must be both initial and privative and must be associated with no transitions. The other failure state must be associated with exactly one transition with a tangible trigger and an empty (always true) guard;

(vii) Every failure must be associated with (act as) exactly one transition trigger;

(viii) Every conjunction of every propagation condition must contain either a set of FM Propositions (i.e. with no state propositions) or a single state proposition;

(ix) No elementary propositions can be negations;

(x) No complex component may have a Mode Space;

(xi) No failure may have an external cause.

Note that some extensions to the 'core' HiP-HOPS [135, 161] revoke constraint *(ix)* and slightly liberalise *(vi)*; the hierarchical mode model [115] revokes *(x)* and integration with design models and permission of non-FM flows [116] can be seen *informally* as an introduction of Normal Events. For FPTN most of the above constraints are also likely to hold with exception of *(ii)*, *(iv)* and, under some interpretations, *(vii)* and *(viii)*.

For systems where it is sufficient, the constrained version of the FLMM yields some advantages. For example such models can be transformed into Fault Trees extremely efficiently using Papadopoulos's synthesis algorithm [115]. In fact, the definition of the Failure Logic Metamodel under the Eclipse platform can be relatively easily supplemented with automatable transformation rules that permit conversion of the (constrained) failure logic models from a particular specification notation or language (e.g. the HUTN or the AltaRica OCAS) into HiP-HOPS and vice versa.

6.3.3 Relationship Between the FLMM and FTA

Previous sections have demonstrated that the Failure Logic Metamodel subsumes existing failure logic modelling methods such as HiP-HOPS and FPTN. However, both methods are strongly related to the traditional Fault Tree Analysis technique. Indeed, FTA is centred on the concept of "failure space" [157] which is very close to what this thesis terms a "failure domain" and which is described by the FLMM. Demonstration of the consistency between these two views, therefore, yields a high degree of confidence in conceptual soundness of the FLM Framework. However, the relatively informal nature of the FTA and the lack of explicit recognition (at least on a syntactic level) of the concept of the "component" make it impossible to claim that the metamodel fully subsumes FTA. This results in weaker "*traces*" relationships between FTA and FLM concepts which indicate that those represent the same phenomena but at different semantic levels.

The starting point for the comparison presented in this section is the metamodel of the fault tree notation specified by Briones et al [26][61]. For the purpose of comparison, this section focuses on the qualitative part of the FTA. Further, the *Vote Gate* is removed since it can be trivially reduced to the remaining two gates. At the same time the FTA Metamodel is extended to include an External Event (also sometimes referred to as a "house event").

The traces between classes of FTA and Failure Logic metamodels are shown in Figure 100 and to some extent were established implicitly by Papadopoulos's fault tree synthesis algorithm [115] (for a restricted subset of the FLMs). The unconstrained version of the FLMM provides opportunities for a more complete mapping (e.g. its concept of Normal Events reflects FTA's External Events that have no interpretation under HiP-HOPS).

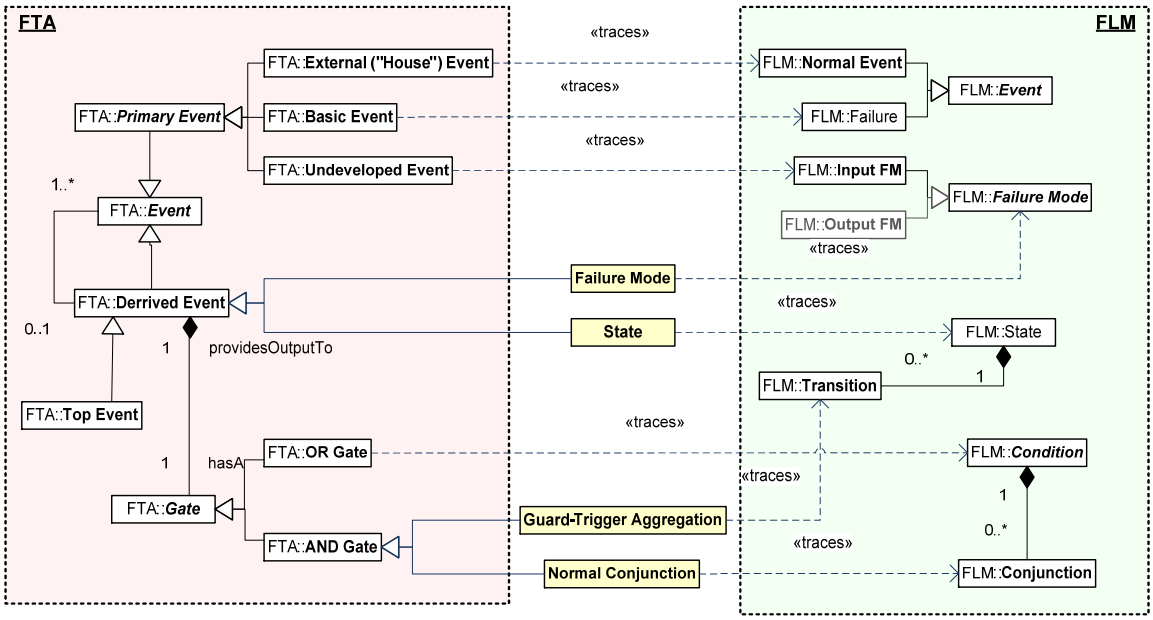


Figure 100 - Mapping between FTA and Failure Logic Metamodels

However, the FTA Metamodel does not adequately describe the Fault Tree Analysis, but merely the syntax of the fault tree notation. The construction rules and guiding principles defined in [157] introduce a number of additional concepts which, whilst not explicitly reflected by the notation, are no less part of the FTA than are its syntactical constructs. It is therefore necessary to consider FTA rules and their relationship to the FLM Framework presented in this thesis.

[61] Whilst Mason developed an alternative, more detailed, FTA Metamodel [102] it focuses on the quantitative aspects of the FTA and adds little value in the context of the comparison here.

The "***No Miracles Rule***" of the FTA[62] is addressed in the next section, where Non Coherent behaviour is discussed. The remainder of this section considers other key rules and principles.

Ground Rule I: "*Write the statements that are entered in the event boxes as faults; state precisely what the fault is and the conditions under which it occurs. Do not mix successes with faults*"

In most cases, the rule is seamlessly implemented in the FLM Framework. The interface of each component is formed by input and output failure modes which are by definition deviations from "success" (or, more precisely, intent). One exception from the first FTA ground rule is the normal state of components in FLMs. The use of these states is relatively rare and normal states are always 'hidden' inside the boundaries of basic components.

Ground Rule II: *"If the answer to the question, 'Is this fault a component failure?' is 'Yes', classify the event as a 'state of component fault'. If the answer is 'No', classify the event as a 'state of system fault'"*

All failure modes and states of basic components defined in the FLMM can be clearly considered as "state of component faults". The metamodel permits two types of "state of system" constructs: the output FMs of complex components and their Modes. It is important to note that since FPTN, FPTC and most versions of HiP-HOPS do not have a concept of "mode" the scope of these methods may be considered as lesser that that of Fault Trees.

The ***Primary-Secondary-Command*** principle of the FTA states that for every state of component intermediate event three categories of causes need to be considered:

- *Primary events* – defined as "any fault of a component that occurs in the environment for which component is qualified" – are loosely mapped into the FLM concept of failure state. More precisely, for a FLM failure state to be considered a "primary event", it must not be predicated on a guard which requires presence of an input failure mode.
- If a transition to a failure state does require an input failure mode, the transition is likely to represent a *secondary fault*. One example of a secondary fault, mentioned in chapter 5, is failure of an electrical generator in response to a short circuit.
- *Command Faults* "involve a proper operation of a component, but at a wrong time or in the wrong place" and map onto the concept of *Input FMs*. More precisely, if an output failure mode can occur as a result only of input failure modes (i.e. with no need for internal failures of the component) every such input FM can be considered a "Command Fault" in the FTA terminology.

[62] The ***No Miracles Rule*** states that: "*If the normal functioning of a component propagates a fault sequence, then it is assumed that the component functions normally*" [157]

In general, Failure Logic Modelling does not make such a clear separation between these three classes of conditions: propagation conditions and guards can mix Primary, Secondary and Command causes as appropriate for accurate representation of component behaviour.

6.3.4 Non-Coherent and Dynamic Behaviour

6.3.4.1 Negation

A negation (or *NOT gate*) is a contentious point in Fault Tree Analysis [10, 75]. Since the models and illustrations used throughout this thesis make extensive use of negation in both propagation conditions and transition guards it is important to briefly justify this.

In the vast majority of the models presented in this thesis, the primary objective of the negation operator is *not* the introduction of "*miracles*" (i.e. coincidental correctness) into the failure logic of components but rather qualitative reduction of the unjustified non-determinism in propagation conditions and transition guards. In other words the "not" operator is used predominantly to provide deterministic characterisation of mutually exclusive failure modes (or state transitions) in situations when a conjunction of their propagation conditions (or guards) would not otherwise always evaluate to *false*.

In fact, the presence of multiple output failure modes of a component (especially when these are associated with the same flow or interaction in the design model) turns a failure logic model into what Andrews calls a "Multitask System" [10, 75]. This holds for the FLM Framework presented in this thesis as well as for all of the more simple pre-existing methods that it subsumes. In fact, Sharvia and Papadopoulos have extended the HiP-HOPS method (and the associated fault tree synthesis algorithm) to allow inclusion of an explicit "not" operator [135]. Interestingly, Wallace's Failure Propagation and Transformation Calculus (FPTC) [163] *implicitly* allows for a non-coherent structure by specifying the failure propagation of components by means of "truth tables" (which may contain logic equivalent to negation). This property of FPTC yields an important general observation: *specification languages and notations may permit non-coherent failure logic even in absence of explicit negation operators*. In particular, implementation of failure modes contained within the same FM group as an enumerated type potentially implicitly introduces the negation.

6.3.4.2 Priority AND Gate

Traditional FTA provides a (limited) facility for capturing sequencing or timing constraints on failure propagation – a *Priority And* (*PAND*) gate. Whilst the FLMM strictly limits propagation

conditions and transition guards to propositional logic, this does not represent a limitation of the FLM approach with respect to the FTA. In fact, *PAND gates imply existence of a state*. For example, a *PAND Gate* over two inputs ***X*** and ***Y*** typically implies that ***X*** moves the system into a state when it is vulnerable to an event ***Y*** (Figure 101).

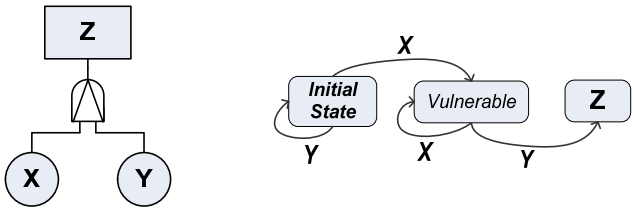


Figure 101 - Typical Interpretation of a PAND gate

In general, a *PAND* gate is an abstraction of an internal state logic of the component. Since the FLM Framework permits explicit representation of the state – the abstraction is no longer necessary.

PAND gates are closely related to the *Transition* specifications. In particular, in the earlier discussion on 'classical' FTA (on page 238), Figure 100 implied an overly pessimistic representation of a decomposition of a transition into a trigger and a guard via a 'classical' AND Gate; a more accurate representation, which acknowledges the fact that the trigger *T* must occur after the guard *G* turns *true*, clearly requires a PAND gate as outlined in Figure 102.

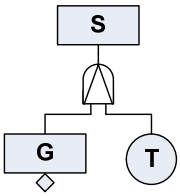


Figure 102 - Improved Fault Tree Representation of a State-Guard-Trigger relationship

However, this relationship is strictly informal and only holds for a small number of specific state space patterns. Firstly, *PAND* gates have no memory and, in the above illustration, once *G* becomes *false* the gate output will return *false*, suggesting that the component has left the state. This is not the case for guards in the FLM Framework. Secondly, FLM state transitions may contain loops (i.e. direct and indirect 'return transitions' to previously visited state). Such looped state structures would yield infinite fault trees (or, alternatively, would make a fault 'tree' a directed *cyclic* graph).

6.3.4.3 Dynamic Fault Tree Gates

Arguably the most prominent of numerous 'dynamic' extensions of the Fault Trees that have emerged in the recent years, Dugan's Dynamic Fault Trees approach [12, 44] proposes two new gates – a *Functional Dependency* (FDEP) gate and a *Cold Spare* (Spare) gate (Figure 103, (a) and (b) respectively). The semantics of these gates is discussed in Chapter Two of the thesis (Section 2.2.2). This section demonstrates how Dougan's DFT gates can be related to the concepts defined in the FLMM.

From the perspective of the FLMM, the FDEP gate is strongly related to the concept of external cause of failure (introduced in Chapter 4) as it permits the interconnection between two independently defined models (in Dugan's case, fault trees; in ours, failure logic models). The FLM concept of 'external cause' as discussed in Chapter 4, however, explicitly recognises that relationships between two models may potentially be more complex and require a translation layer rather than direct connection.

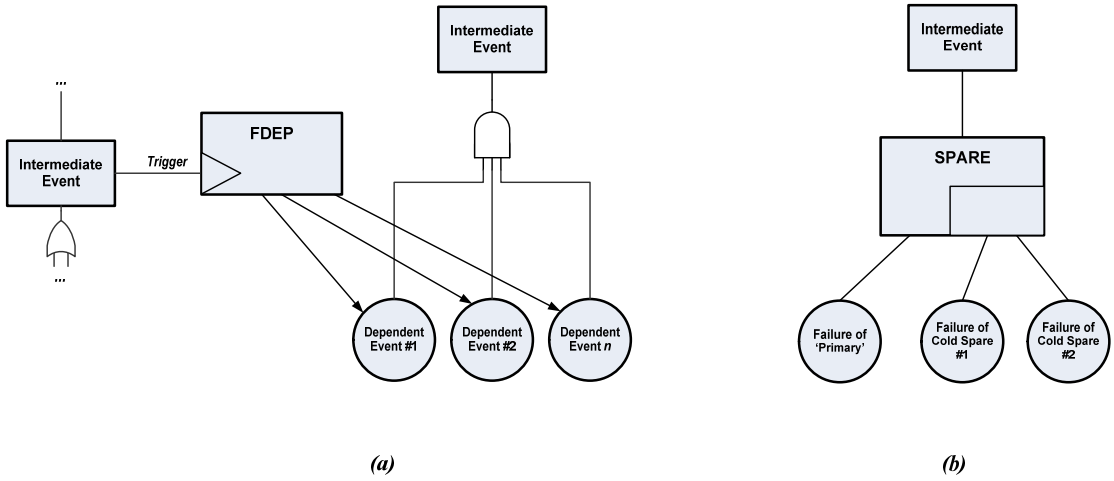


Figure 103 - Dynamic Fault Tree Gates: Functional Dependency (a), Cold Spare (b)

More importantly the framework for rationalising the decomposition of large-scale engineering artefacts (platforms) into Engineering Domains related by Allocation Domains has clarified (even for the Fault Tree context) cases in which the usage of the FDEP gate is permissible. If a common mode failure (indicated by the FDEP trigger input) falls within the scope of the same Engineering Domain, it is unlikely that FDEP gate can be justified (instead the common 'trigger' should be modelled as a secondary or command fault of the component). Just as with external cause in the FLM Framework, Dugan's FDEP gate in FTA context is more likely to be justifiable if it arises from considering allocations between two independently engineered domains of the platform (such as IMA and software partitions of the subscribing system).

By considering the second DFT gate – *Spare* – from the perspective of Failure Logic Modelling one limitation can easily be identified: the gate is unable to reflect the situation whereby a stand-

by component is *wrongly* activated. Existence of a "cold spare" implies existence of an activation input to the component in the design. Just as with any other design interaction this input may carry a *Commission* failure mode. So a Spare gate captures one – very specific – pattern of behaviour associated with systems which exhibit dynamic reconfiguration.

Having said this, the gate highlights an important principle for the construction of failure logic models: *failure state transitions triggered by a failure should never be prohibited by the guards when the failure is physically possible.* If, in a certain state (and under certain input FM conditions), a failure is possible but inconsequential, then the component characterisation should contain a transition for re-entry into the same state.

Under this principle, failure logic models are capable of storing information about "cold spares" which can be utilised for quantitative assessment. The "activation" of the spares is captured in the FLM Framework by inclusion of the mode and input FM assertions in the appropriate transition guards (for normal and abnormal activation respectively).

6.3.5 Evaluation by Metamodel Instantiation

The Failure Logic Metamodel has been instantiated in the AltaRica OCAS language as discussed in Chapters 3, 4 and 5 of the thesis. The fact that the metamodel can be instantiated in a third-party general language confirms both the practicability of the proposed framework and the completeness (and, to some extent, the internal consistency) of the proposed framework it underlies.

It is important to stress that, whilst the AltaRica OCAS was chosen as implementation language before the metamodel definition had been completed, care has been taken throughout the research not to permit features of the language to drive the metamodel definition. All new FLMM concepts have been demonstrated in a language-neutral pseudocode format (e.g. the tabular format adopted in this thesis or a semi-graphical format in [97]) and subjected to a review in a language-independent form.

Also, it is important to note that whilst the 'baseline' FLM Framework (presented in Chapter 3) cleanly maps to elementary AltaRica OCAS language concepts (i.e. events, states, transitions, flows and assertions), the extensions introduced in Chapters 4 and 5 could not be directly supported by the language and required more complex (composite) constructs. Nevertheless for each FLMM concept it was possible to provide a systematic (and, in principle, automatable) implementation procedure. This, once again, provided confidence in the completeness of the metamodel.

Finally, implementation of the FLM Framework in AltaRica helped to identify the pragmatic impact of limitations of the language (in the FLMM context) as well as clarify criteria for a more suitable implementation language yielded by the metamodel (such as permitting states of composite/complex components and supporting directed weak synchronisations between events). This in itself is a contribution of this research.

6.4 Evaluation Through Peer Review

Discharging the thesis proposition has relied on peer review to assure the adequacy of the other, more formal, means of evaluation (outlined in the previous two sections) as well as to mitigate against any limitations of case studies and metamodel experiments. Furthermore, the evaluation of the FLM Framework with respect to the criteria of the Proposition (such as being "well-defined" and "pragmatic") calls, to some extent, upon engineering judgement that, whilst being informed by other forms of evaluation, can only be elicited in the peer review.

The research has been subjected to peer reviews in two fora discussed in dedicated sub-sections below: the Airbus Dependability Network project and the EU-funded MISSA project. The research has also been treated to regular peer review in the High Integrity Systems Engineering group at the University of York.

6.4.1 Airbus Dependability Network

The Airbus Dependability Network (DepNet) was a collaborative project funded by Airbus between January 2004 and February 2007. The network comprised three European research organisations: Kuratorium OFFIS e.V. (Germany), l'Office National d'Études et de Recherches Aérospatiales[63] (France) and the University of York (UK).

The first phase of the project looked into the issues of compositional safety assessment and resulted in publication of a research report (the "white paper") in June 2005 [74]. The report proposed a preliminary framework for compositional assessment with significant contributions from the author. From the perspective of the FLM Framework presented in this thesis, the White Paper presented both the baseline framework outlined in Chapter 3 and the preliminary definitions of the extensions proposed in Chapter 4. Having been formally reviewed and positively received by the DepNet steering committee (which comprises of highly-qualified safety engineers and program managers from all key Airbus sites and divisions), the report has been recommended by

[63] Referred to as "ONERA"

Airbus as a foundation for the MISSA project proposal (subsequently submitted and granted under the EU Framework 7 programme of research).

6.4.2 The MISSA Project

More Integrated Systems Safety Assessment (MISSA) is a European collaborative project funded by the European Commission under Framework 7 programme[64]. Having started in April 2008 and projected to run for 36 months, the MISSA project is being carried out by a consortium coordinated by Airbus (UK) and consisting of 13 companies and research organisations including: Alenia Aeronautica, Dassault Aviation, EADS APSYS, OFFIS, ONERA, Thales Avionique and the University of York. Work Package 4 of this project is largely focused on the evaluation and further development of the FLM Framework presented in this thesis.

The quarterly project meetings and formal deliverables have been extensively used by the author to conduct a peer review of the proposed FLM Framework, its instantiation in the AltaRica OCAS and the case studies presented in this thesis. Project partners have been broadly divided into the following (overlapping) categories:

- Research partners with an interest in the conceptual and methodological aspects of the FLM Framework: ONERA and OFFIS (the later with the focus on the metamodel adequacy);
- AltaRica & AltaRica OCAS experts: Dassault Aviation, ONERA, EADS APSYS, Thales Avionique;
- Industrial partners with an interest in the practical and methodological aspects of the framework and associated guidance: Airbus, Alenia Aeronautica, Thales Avionique
- Industrial partners with a specific interest in case studies: Alenia Aeronautica and Airbus

The key formal MISSA report that outlined the key principles of the FLM Framework was the MISSA mid-term Development Description Report (D4.10, Issue A). This also included a draft of ***FLM Handbook*** compiled (solely) by the author. Fashioned after the FTA Handbooks [158, 157], this document describes (and illustrates) all key FLM concepts and the overarching 'philosophy' of failure logic modelling approach as well as presenting the step-by-step guidance on the system assessment process which can be used to systematically construct the models. The handbook also contains a formal definition of the FLMM and instantiation schema in the AltaRica OCAS. Formally reviewed by project partners, the draft Handbook has attracted very positive feedback with constructive suggestions of improvements, but no major or conceptual criticisms.

[64] Grant agreement ACP7-GA-2008-212088 [6]

In addition to this formal review a number of extensive presentations have been delivered by the author to the project consortium throughout Phase I to elicit comments on the Framework and the Case Studies. The three most significant such presentations were as following:

1) *Bristol (UK), April 2008:* Baseline FLM Framework
2) *Trento (Italy), January 2009:* A major half-day presentation covering most aspects of the work, including:
    - Results of the review of the aircraft Fuel System
    - In-depth review of the FLM Framework extensions with respect to reconfigurable systems
    - Review of the FLM Framework with respect to model composition
    - Common computation and communications platform (IMA) case study
    - FLM Framework implementation in AltaRica (incl. case studies) and identified limitations of the language
3) *Stockholm (Sweden), June 2009:* An in-depth presentation of the EPDS case study
    - Review of the FLM Framework with respect to modes
    - EPDS model hierarchy
    - Identified limitations in Cecilia OCAS
    - Preliminary results of the application of a lightweight refinement relation

Finally, a number of small-scale review meeting and discussions were held with the individual project partners including discussions with a senior safety engineer of Airbus UK on the conclusions of the aircraft fuel system review, review of the Failure Logic Metamodel by researchers in OFFIS, discussions with researchers at ONERA and engineers at Dassault Aviation on implementation of the FLM Framework in AltaRica OCAS, review of the EPDS with Alenia Aeronautica representative in MISSA project and review of FLM Handbook by the engineers in Thales Avionique.

6.4.3 Peer Review Summary

The FLM Framework presented in this thesis and the evaluation activities presented in this chapter have been extensively peer reviewed by author's collaborators in DepNet and MISSA projects as well as by colleagues in the HISE group at York. The comments received have been positive overall, supporting the thesis proposition in general and finer-grained hypotheses outlined in Evaluation Strategy (see section 6.1 above) in particular. The key review instances are summarised in Table 13 below.

Table 13 - Summary of Peer Review Instances

Review Instance	Basis for Review	Reviewers	Scope
York (UK), January 2005	Research Seminar (presentation)	HISE group researchers	Baseline FLM Framework
June 2005	1st DepNet White Paper (technical report)	Safety engineers of Airbus (DepNet Steering Committee)	Baseline FLM Framework and approach to composition of independently defines DSFMs
Bristol (UK), April 2008	MISSA WP 4 Presentation	MISSA Consortium, EC project officer	Outline of the complete FLM Framework
York (UK), May 2008	Research Seminar (presentation)	HISE group researchers	Engineering & Allocation Domains, DSFMs and model composition
Trento (Italy), January 2009	Series of presentations (quarterly MISSA project meeting)	ONERA, Thales, Airbus, Alenia	FLM Framework extension for modelling reconfigurable systems
		Airbus, Alenia	Review of Aircraft Fuel System & Identified Characteristics
		Airbus, Apsys, Alenia, ONERA	Composition of multiple models (DSFMs, external causes of failure)
		Airbus, Thales, Apsys, Dassault	IMA case study
		ONERA, Apsys, Dassault, Thales	FLM Framework implementation in AltaRica (including resolution of strong dependencies)
York (UK), May 2009	Research Seminar (presentation)	HISE group researchers	FLM Framework extension for modelling reconfigurable systems
			Non-compositional aspects of FLM Framework (context dependency & loops) and approaches to mitigation

Stockholm (Sweden), June 2009	Presentation (quarterly MISSA project meeting)	MISSA Consortium	FLM Framework extension for modelling reconfigurable systems (final review)
		Alenia, Airbus, APSYS	EPDS case study (hierarchy of models)
		ONERA, APSYS	Implementation in AltaRica OCAS (incl. language and tools limitations & adopted mitigations)
		Alenia, Airbus	Preliminary results of lightweight refinement evaluation
Mid-term MISSA project review, September 2009	D4.10: Development Description Report (Issue A)	MISSA Consortium, EC Project Officer	FLM Framework
	Draft FLM Handbook (Annex A of D4.10 above)	MISSA Consortium	Key concepts and philosophy of the approach
		Airbus, Thales, Apsys	Guidance for construction of models
		OFFIS	Failure Logic Metamodel
		ONERA, Thales, Apsys	Framework implementation in AltaRica OCAS

6.5 Identified Limitations and Mitigations

Whilst yielding a positive result overall, the evaluation activities reported in this chapter have highlighted some fundamental limitations of the original underlying principles of failure logic modelling. In this section, these are summarised along, wherever applicable, with the pragmatic mitigations of these limitations taken by the author.

It is important to stress that identified challenges and limitations are applicable to any failure logic modelling approach, regardless of the details of the model specification language or notation used to implement it. An ability to identify such fundamental issues in the approach as a whole highlights the utility of the language-agnostic metamodel-based framework and contributes to author's long-term goal of comparing relative advantages and disadvantages of different model-

based safety assessment approaches that formed part of the motivation for the research presented in this thesis (see section 1.2).

6.5.1 Volume of Results

One of the well-established, although not frequently publicised, challenges posed by any model-based safety assessment method is the accessibility of the results of the analysis. Failure logic modelling is not an exception in this respect: some of the analyses of the models presented in this thesis contained hundreds or even thousands of minimal cut sets (and even a greater number of minimal cut sequences) of relatively modest cardinality.

This problem is more severe for model-based safety assessment than it is for traditional analysis methods such as, for instance, the fault tree analysis. Under the latter safety engineers construct separate models (fault trees) for each unsafe condition with no formal enforcement of consistency between models. Whilst the lack of a 'consistency guarantee' between fault trees is often criticised, and is one of the motivational factors for model-based safety assessment methods it is often pragmatically useful. In particular, safety engineers can select the level of granularity of the model appropriate for each *individual* condition of interest, thus, abstracting from details which, whilst they may be more significant in some other context, have little impact for the selected top level event. Safety engineers may aggregate a number of failures in a single basic event or leave some failure modes (possibly also aggregated) as undeveloped events.

By contrast, model-based approaches are, by definition, based on a single model of the system defined at a particular fixed level of detail which must support multiple analyses. Consequently the model has to be constructed at the 'lowest common denominator' of granularity, thus, leading to voluminous analysis results that can often be perceived as 'too detailed'. The overall result is that it is impractical to review analysis results exhaustively, either for the purpose of establishing safety qualities of the proposed system architecture or to review, validate and debug models themselves.

In the light of this challenge, some of the case studies have adopted an incremental modelling approach whereby less detailed (and, strictly speaking, less accurate) models are constructed first. In subsequent iterations, models are improved and levels of detail are progressively added. However, instead of reviewing all of the analysis results at each iteration minimal cut sets are compared to those obtained in the earlier iterations. Only significant differences between the results are reviewed manually.

To identify such significant differences a lightweight refinement relation has been defined. The relation holds over two sets of minimal cut sets if and only if for every 'concrete' MCS it is

possible to find an identical or worse abstract MCS; where one cut set is said to be worse than another if the later subsumes the former. Concrete cut sets that violate this refinement relation provide significant *new* safety information not previously seen by the safety engineers at more abstract iterations of the assessment. The comparison process has been automated and applied to the EPDS case study. It has proven to be useful both in terms of significantly reducing the number of cut sets requiring review in general and in facilitating more efficient model validation and review in particular.

The basic refinement relation above has been supplemented with a notion of equivalence mapping between failures (i.e. the vocabulary over which MCSes are defined). The original motivation for this extension was the need to compare results obtained from models defined by different engineers (i.e. in the absence of identical naming conventions for components and their failures). However, some preliminary experiments on the application of the mapping to control the level of granularity of the analysis have been undertaken. For example, the mapping was used in the IMA case study to group similar failures of similar components (e.g. to consider all scheduling failures of different CPIOMs as equivalent) and present shorter aggregated or summarised results to the user. Nevertheless, further experiments in this area are necessary (and are planned under the MISSA project).

6.5.2 Strong Circular Dependencies

The first challenge specific to the failure logic modelling approaches is the one posed by strong circular dependencies between the failure logic of components in the context of abstract models. To the author's best knowledge, this issue has not been addressed in the literature and was identified during the evaluation stage of the present research. As was discussed earlier in the chapter, during model simulation and analysis such strong dependencies (unless explicitly handled) lead to the model stabilising in an incorrect local equilibrium regardless of the analysis method used. The problem is ultimately caused by the level of abstraction inherent to the FLM Framework (that mandates description of *deviation* of behaviour from intent rather than system behaviour itself) especially when the models purposely abstract from exact reconfiguration rules (e.g. EPDS #1 and EPDS #2 models, described in Section 6.2.4, that only capture the physical architecture of the system). The problem is effectively that of co-dependency between redundant parts of the system (e.g. power distribution paths or fuel tanks) in terms of the provision of a particular service.

It is important to stress that this problem is conceptually different from the more widely recognised issue of closed loop control systems. Control loops pose little challenge to the FLM Framework as they frequently do not result in loops in the failure logic models (since sensors or monitors are only dependent on the failure modes of the 'primary' channels if the latter can

mislead or incapacitate the former) and, if they do, can be trivially resolved in the context of inductive (bottom-up) model analysis. It is also important to note that resolution of control loops, typically based on the introduction of minor delays, poses more significant risks to the validity of models based on system "success" (i.e. *non failure logic modelling* approaches) as delays are more likely to interfere with the timing and sequencing aspects of the behaviour captured in the models.

Returning to the FLM-specific strong circular dependencies, a pragmatic solution to the problem has been found and successfully applied to the EPDS case study (which exhibits such dependency). The solution relies on the injection of transient behaviour into the model for an infinitely short time (undetectable by the analysis) to force analysis tools to search for a correct non-local equilibrium before confirming a fixed state. Whilst in case studies performed during the research (including the EPDS and the model of the tank architecture of the fuel system) the detection of circular dependencies was automated and the injected behaviour was trivial, it is important to admit that this transient behaviour has no engineering semantics and cannot be established at the level of individual components. Therefore the solution is not covered by the FLMM and cannot be expressed in its terms.

Furthermore, the problem affects failure logic models constructed at the earliest stages of the development process (e.g. where the objective of the analysis is to assess the feasibility of the physical architecture of the system) – the very stage where the use of the failure logic modelling approach is more likely to be justifiable. Subsequently, this problem clearly requires further research in order to assess its impact on the feasibility of the failure logic modelling approach and its comparison to other model-based safety assessment paradigms.

6.5.3 Complex Modes and Reconfiguration Logic

Whilst the research presented in this thesis has demonstrated that the FLM Framework can be consistently and systematically extended to enable modelling of systems capable of dynamic reconfiguration, the resultant framework and model construction process is significantly more complex than is currently admitted in other descriptions of other failure logic modelling approaches. Construction of the mode models of systems and complex components is a non-trivial process that requires an understanding not only of behaviour of the system but also of the *rationale* for this behaviour. Consequently, the degree of intellectual enquiry and, to some extent, of creativity, necessary for the elicitation of modes, the specification of their transitions and the reflection of the impact on individual components (within the relevance scope of particular mode) are likely to render the FLM Framework less applicable for *rapid* 'what-if' assessment of the impact of changes to the logic of system controllers (which determines a system's concrete reconfiguration rules).

However, this apparent inefficiency of the framework indirectly aids assessment of the scale of design change. Indeed, seemingly minor changes of the controller logic (e.g. changes of a single reconfiguration condition) are capable of having a significant impact on the behaviour of the system. From the perspective of the FLM Framework such changes are (appropriately) seen as affecting the system (and model) architecture and thus requiring substantial re-consideration by safety engineers. In other words, the framework insists that the extent of safety (re-)assessment should be determined by the impact of the design change on the overall behaviour of the system rather than by the ease of implementing the change (e.g. changing controller code) itself.

Furthermore, whilst specification of the system modes (especially failure handling modes) under the failure logic modelling approach is undeniably a labour-intensive activity, it is neither mechanistic nor mundane and ensures that safety engineers gain a thorough understanding of the system architecture and its operational 'philosophy'. Also, this activity essentially forces engineers to conduct a *thorough review* of the system design from a perspective that is *conceptually dissimilar* from that typically adopted in the 'core' design and implementation processes. This can be seen as fulfilling one of the core (albeit frequently implicit) objectives of the safety assessment process by ensuring a sufficient degree of independence and analytic redundancy between the safety assurance and design *processes*.

It is expected that the FLM Framework is more likely to be perceived as adding value to the assessment of radical system designs (that propose innovative architectures and implementation technologies) than in the context of repeated well-established designs (often relied upon in traditionally conservative civil aerospace sector).

6.5.4 Reuse and Composability of the Component Characterisations

This investigation into challenges posed by the dynamically reconfigurable systems has brought into focus the problems of the reuse and composability of the failure logic characterisations of components. Indeed, as was discussed in Chapter 5 in the general case component characterisations under the failure logic modelling approach are neither reusable nor composable.

The issue of composability is the more trivial of the two problems, and can be mitigated by a proposed two-stage model construction methodology (see Section 5.4). During the first stage, safety engineers identify components and key safety threats of the system and use lightweight scenario-based assessment to establish the appropriate granularity for the FM interfaces. At the second stage precise failure logic characterisation of the basic components is established and

mode models are completed with the specification of transitions. Overall, this two-stage process minimises the incidence of non-composable component characterisations, provides opportunity for early feedback to the design process and provides confidence that the constructed failure logic model is suitable for anticipated analyses.

The problem of non-reusability and, generally, of the context dependency of component characterisations is much more significant. It is also largely unique to the failure logic modelling approach and doesn't affect 'competing' approaches based on modelling system behaviour under conditions of failure from the perspective of the success domain. For example, under the FLM Framework, almost every component of the EPDS case study had to be modelled separately despite the fact that the design of many components (switches, circuit breakers, transformers and junctions) is identical. The problem is that the context of each such component and, in particular, their intent in different modes of the system, was unique.

However, it was observed during the case study that characterisations of components of similar types exhibit common patterns. Trivially, this allows reusing large sections of characterisations with modifications limited to state propositions over system modes. More importantly it highlighted a promising long-term conceptual solution to 'restoring' the reusability of FLM components. In particular, it was observed that if in transition guards and propagation conditions system-level modes are stated from the perspective of a component, the characterisation in terms of such 'local modes' becomes significantly more stable. For example, from the perspective of individual circuit breakers, modes are grouped into two categories: modes in which circuit breakers are expected to be powered and modes in which they are not. The classification of the EPDS modes into these two groups is indeed unique for almost every circuit breaker; however, failure logic of circuit breakers in terms of these two local modes is identical for all components.

The overall solution should therefore extend the FLMM by the concept of "local modes" (for both complex and basic components). Unlike other states of the components, these modes must never be associated with transitions, but rather mapped on to modes (local or otherwise) of higher level complex components. Furthermore, the guards and propagation conditions of the components must only refer to higher-level modes by virtue of these local representations. This solution allows for the segregation of reusable propagation conditions and guards from instance-specific mappings between local and higher-level modes.

This approach to failure logic modelling has been successfully trialled on the EPDS case study. However, since the case study is formative with respect to this extension, it has not been included in the Failure Logic Metamodel presented in this thesis and requires further evaluation.

6.5.5 Complexity of Model Construction

Related to the issue of reuse and composability of the failure logic models (discussed above) is the problem of model construction in general. Whilst all of the extensions of the failure logic metamodel introduced in this thesis are necessary for modelling modern complex safety critical systems, these extensions make model construction a significantly more complex process. At the level of individual components safety engineers have to consider a large number of variables including component failure-, failure handling- and normal- states, input failure modes as well as, potentially various higher-level modes.

At the level of models of systems and complex components the research reported in this thesis has demonstrated that establishing the architecture of the failure logic models is a non-trivial task; an inadequate model architecture can result in a time-consuming modelling roll-backs that affect many components. In particular, the author's discussion with industrial collaborators and elicitation of the systems' (or complex components') modes and mode models was highlighted as a particularly challenging step of model construction.

To alleviate some of the problems of complexity of the model construction, guidance for systematic system assessment and elicitation of the failure logic model has been compiled by the author (in the form of a "FLM Handbook" mentioned previously in this chapter). To the author's knowledge, no such guidance had been previously provided for any model-based safety assessment technique with existing publications focussing on the particular model specification notations or, in some cases, the definition of safety engineering concepts underlying the technique rather than a process of model construction.

However, whilst the Handbook has been reviewed by a number of the author's collaborators, more independent case study based evaluation is necessary to ensure that the guidance is sufficiently clear, comprehensive and – most importantly – repeatable. Provisions for such evaluation of the guidance has been made under the auspices of the MISSA project (ongoing at the time of writing).

6.5.6 Complexity of Model Analysis

The duration of the analysis of failure logic models has been previously highlighted as being at times inadequate (see sections 4.8 and 5.6.4). Whilst the author has argued that, to the large extent, the excessive analysis duration can be attributed to the 'brute force' strategy embedded in the analysis tools used as well as limitations of the AltaRica OCAS specification language, it is also important to admit that introduction of the notions of component state as well as presence of the potentially global modes in the model inevitably increase model complexity. However, the

expressive power of the methodology presented in this thesis is significantly greater that that of traditional combinatorial techniques (such as fault trees, reliability block diagrams, cause-consequence diagrams) and all existing failure logic modelling techniques that the author is aware of (including FPTC, FPTN and HiP-HOPS). Furthermore, the various notions of state have been demonstrated to be necessary for adequate modelling of the modern industrial-scale safety critical systems.

Finally, it should be noted that the metamodel-based approach permits a clear definition of the *methodology constraints* under which the failure logic models can be efficiently 'analysed' by model transformation and parsing techniques (such as Papadopoulos's fault tree synthesis algorithm [115]). Therefore decisions about whether or not to 'sacrifice' accuracy of the model for more time-efficient model analysis can be made and justified on a case-by-case basis given particular challenges of the system at hand and the stage of the development process.

6.6 Summary

This chapter has presented the evaluation argument adopted to support the proposition defended by the thesis (as defined in section 1.3). Section 6.1 outlined the Evaluation Strategy, systematically decomposing the overall thesis proposition into manageable claims and, ultimately, backing evidence. Sections 6.2 through 6.4 have summarised the evidence that supports the evaluation argument. The proposition has been defended on the basis of three major forms of evidence: case studies, peer reviews and metamodel experiments.

The chapter has concluded with an outline of some of the limitations identified in the FLM Framework (section 6.5) along with the pragmatic mitigations adopted during the research and evaluation. Some of these are returned to in the following chapter as promising directions for further research. However, the identification of fundamental limitations to failure logic modelling as a general family of safety assessment approaches (rather than to individual methods, languages or notations) demonstrates the utility of the language-agnostic metamodel-based framework and is in itself a major contribution of the thesis.

Chapter 7: Conclusions

This chapter summarises the key contributions of the Thesis, presents some overall concluding remarks, and identifies possible areas of future work prompted by the work reported in the thesis.

7.1 Summary of Contributions

The thesis has characterised a specific family of model-based safety assessment methods – failure logic modelling – and has presented a FLM Framework that unifies and subsumes existing failure logic modelling techniques. The framework has been demonstrated to be applicable and capable with respect to the challenges posed by real industrial safety critical systems. However, the thesis has also identified a number of important limitations inherent with this approach to safety assessment (as described in detail in Section 6.5 of the preceding chapter). These limitations, whilst shared by all of the existing techniques that the framework subsumes, to date have not been reported.

Overall, the thesis contributions include:

- *Definition of a general but instantiable,* ***Failure Logic Metamodel*** *that defines and unifies a prominent family of model-based safety assessment methods and techniques.*
- *Definition and evaluation of an* ***approach that enables the composition of multiple failure logic models*** *in the realistic context of large-scale industrial safety-critical platforms*
- *Definition of an* ***approach that enables the failure modelling of complex reconfigurable and multimodal safety-critical systems****; demonstration that the proposed approach addresses various types and patterns of modes, reconfiguration and architectural limitations of realistic systems*

In addition to the above three contributions – broadly related to Chapters 3, 4 and 5 – there is a fourth, 'emergent' contribution of the thesis as it has become clear that the goal of a fully automatable or fully compositional process for safety analysis, based on failure logic models, is not attainable. Whilst this may seem disappointing, there are also advantages as the approach allows safety engineers to engage with the safety process at an appropriate intellectual level.

The contributions of the Thesis are summarised in the following sections.

7.1.1 Unifying Failure Logic Metamodel

Chapter Two has presented a survey of the existing model-based safety assessment techniques. However, these techniques are currently only categorised and related in an *ad hoc* fashion in the published literature. This thesis has proposed a classification of the existing techniques and identified a coherent family of model-based safety assessment methods, namely *failure logic modelling*. The thesis has presented a ***Failure Logic Metamodel*** that unambiguously defines this family and subsumes all of the techniques such as FPTN, FPTC and HiP-HOPS. The metamodel also unifies failure logic modelling techniques with the general specification languages used in the model-based safety assessment. Until now, the publications on the application of such languages have not defined a clear *engineering* semantics of the use of language constructs for failure logic modelling. At the same time publications on the failure logic modelling techniques that do address engineering semantics typically introduced *idiosyncratic notations.* The metamodel-based approach overcomes these limitations.

In particular, Chapters Three through Five have demonstrated that the general, notation-independent, *Failure Logic Metamodel can be instantiated in a third-party specification language* (AltaRica OCAS/Dataflow) in a systematic fashion. Whilst *effective* specification in a particular language may impose *constraints* on the FLM Framework, the metamodel facilitates identification and assessment of the impact of these constraints. Subsequently, it facilitates a rational approach to language selection as well as structuring the safety case argument concerned with the justification of safety assessment results obtained on the basis of failure logic models.

7.1.2 Composition of Multiple Failure Logic Models

Whilst publications on existing failure logic modelling techniques have made claims of model composability (e.g. [162, 163, 165]), they have assumed that all necessary interfaces are identifiable at the level of individual models. For the large-scale safety critical platforms, where constituent parts are typically designed in relative isolation and by different engineering stakeholders, this view is unrealistic. Furthermore, it potentially undermines the explorative nature of the safety assessment tasked with *identification* of significant interactions between the 'parts' of the overall safety-critical platform. Chapter Four has presented a flexible ***abstract model of platform decomposition***, based on a concept of ***Domains*** that combine complementary notions of a system model "*containment hierarchy*" and "*architecture views*". The defined approach captures and rationalises different relationships between engineered artefacts within the platform.

The FLM Framework, as extended in Chapter Four, provides a pragmatic means for composing failure logic models utilising the concepts of ***Domain-Specific Failure Logic Models (DSFMs)***, ***virtual translation components*** (accommodated in the model ***translation layer***) and the explicit

modelling of ***external causes of failure***. The resultant approach permits the composition of independently defined failure logic models in absence of *a priori* well-identified or harmonised failure mode interfaces.

7.1.3 Modelling Reconfigurable and Multimodal Systems

Of the existing failure logic modelling techniques surveyed, only one – described in [115] – has attempted to address the issue of dynamic reconfiguration – a characteristic present in a number of real industrial systems. This technique, however, was shown to be incapable of adequately capturing the behaviour of reconfigurable systems in practice. In particular, it proposes an over-constrained model of modes and sub-modes that has no natural interpretation for some systems. Furthermore, being embedded in an otherwise combinatorial framework, the HiP-HOPS approach cannot accurately reflect the persistent effects of failures sustained in one mode on system operations in other modes.

A more flexible approach to handling reconfiguration has been incorporated into the FLM Framework. Based on the well-defined concepts of ***modes*** and ***mode spaces*** of complex components introduced in Chapter Five, and the *dynamic behaviour of basic components* as defined in Chapter Three, the approach presented in this thesis permits the modelling of *various types of reconfigurations found in industrial safety critical systems.* These include dissimilar modes of system use (such as electron and photon treatment modes of the Therac-25 machine), operation through a standard sequence of phases (such as take-off, climb, cruise, descend, land and taxi 'flight phases' for a civil aircraft) or system reconfiguration upon detection of failure (including fault accommodation and 'graceful degradation' provisions). Chapter Five has also demonstrated that this approach can prove extremely useful in analysing the *effects* of limitations of architectural designs of systems and complex components. This issue has not been addressed by any of the surveyed failure logic modelling techniques.

Finally, the research into the challenges posed by system reconfiguration to failure logic modelling approach has demonstrated that simple application of this family of safety assessment techniques does *not* guarantee reusable models. In particular, this thesis has clearly shown that failure logic *characterisations of components are inherently context-dependent* regardless of whether the context is dynamic or static. This result has not been previously reported by any of the publications on existing failure logic modelling techniques.

7.1.4 The Non-Automatable and Non-Decomposable Nature of Failure Logic Modelling

Through the research presented in this thesis it has emerged that construction of failure logic models of realistic systems requires significantly *more thorough assessment of the system than has been previously acknowledged in publications*. In particular, various publications have suggested that failure logic models can be simply and trivially composed from component characterisations [114, 115]. In contrast, this thesis has demonstrated that in order to define composable component characterisations significant effort must be invested into non-decomposable architecture-level *assessment of the system*. Chapters Four and Five have ***defined the structure and provided the guidance for such assessment process***.

The role of the failure logic modelling in the overall development and safety engineering processes requires significant consideration. At the earliest stages of design failure logic models are susceptible to strong circular dependencies that are not trivial to resolve (and, in some cases, to detect), whilst at the later stages of the design, when detailed information about reconfiguration emerges, the specification of failure logic models becomes labour-intensive – requiring significant analytical effort. However, failure logic modelling requires very significant intellectual engagement of safety engineers with the proposed design of the system and its justification. This is likely to yield a thorough understanding of the system that is beneficial to the system safety engineering process as a whole.

7.2 Further Work Areas

In the course of research reported in this thesis a number of areas for further research have been identified. These include:

- ***Transformation of failure logic models*** *between different 'concrete' specification notations*
- *Improving the* ***qualitative and quantitative analyses*** *of failure logic models*
- *Addressing issues of* ***modelling time*** *in the failure logic modelling of complex systems*
- *Developing approaches to* ***improving the reusability*** *of the failure logic models*
- *Use of* ***alternative modelling paradigms*** *in the context of the Failure Logic Metamodel*
- *Performing a systematic comparison with, and establishing relationships with,* ***other model-based safety assessment approaches***

7.2.1 Transformation of Failure Logic Models

The Failure Logic Metamodel can facilitate the transformation of models across different notations in a semantics-preserving fashion. To achieve such transformations the metamodels of individual notations must be captured (e.g. similar to the AltaRica Dataflow ontology developed by Mokos et al [108]) along with a clear understanding of the constraints imposed by the notation on the instantiation of the FLMM and transformation rules[65]. In addition to this, customised parsers may need to be developed for the precise model formats of individual tools. A similar principled metamodel-centred approach can also be taken to the synthesis of fault trees (if deemed necessary) or other 'classical' safety assessment formats from failure logic models.

Overall, the Failure Logic Metamodel can provide a central interchange format for transformations avoiding the need to establish a large number of pairwise translations. Furthermore, it will enforce a principled and clear approach to failure logic model translation that preserves engineering semantics of the safety assessment artefacts.

7.2.2 Model Analysis

The main focus of the research presented in this thesis was on qualitative construction of failure logic models and, to a lesser extent, their analysis. In terms of the time-complexity of the qualitative analysis, it has been noted that the 'brute force' approach, based on exhaustive simulation and search, adapted by the existing tools yields inadequate results. Further work is necessary to develop more complex tools capable of optimising internal model representation and, thus, reducing analysis run-time. Some of the work is currently underway to adapt NuSMV-SA platform for analysis of failure logic models specified in OCAS and Dataflow dialects of the AltaRica language.

Also, whilst the Failure Logic Metamodel enables the specification of probabilistic aspects of system behaviour in 'failure logic domain', this aspect of the metamodel has not been sufficiently evaluated. The current prevailing approach to quantitative safety analysis in model-based assessment relies upon the export of analysis results – typically Minimal Cut Sets – to commercial RAMS tools (such as Isograph's Fault Tree Plus). This approach clearly doesn't preserve dynamic aspects of behaviour and can result in, potentially significant, overestimation of the probabilities of system-level failure conditions.

[65] Given that the FLMM is specified in the Eclipse Modelling Framework and well-formedness constraints in Epsilon Constraint Language (ECL), the transformation rules can be naturally specified in the Epsilon Transformation Language (ETL) [45, 86]

To achieve a more accurate analysis two approaches can be taken:

- Failure logic models can be transformed to a standard 'state-based' reliability analysis format (such as Markov Models or Petri Nets) using the approach described in the previous section
- Tools can be developed specifically for standard modelling languages (such as AltaRica, AADL Error Model Annex, SCADE/Lustre or StateMate/StateFlow) and their respective modelling environments. In practice, such tools are likely to be based on Monte Carlo simulations or an emerging concept of Probabilistic Model Checking.

Since quantitative analysis plays an important role in both the safety engineering and certification of safety-critical systems, this is both a feasible and necessary area for further work.

7.2.3 Issues of Modelling Time

The review of the Aircraft Fuel System highlighted that for some systems modelling the timing and duration of conditions (including failure modes) is critical to fully capturing failure behaviour. For such systems it is unclear whether the level of abstraction inherent in current failure logic modelling methods is adequate. Further systematic investigation into the challenges posed by such systems to the FLM Framework presented in this thesis is therefore necessary.

7.2.4 Improving Reusability

Whilst it was demonstrated that components in failure logic models are context-dependent and, thus, non-reusable an approach to separating context-independent and context-dependent parts of component characterisations was proposed during the EPDS case study reported in Chapter Five (Section 5.6.3). Based on the notion of a mapping between system modes and local modes (as 'perceived' by components), the approach, however, has not been thoroughly validated through non-formative case studies and formal peer review. Formalisation and evaluation of this approach is therefore a promising and viable area for further work.

7.2.5 Alternative Modelling Paradigms

Whilst, as has been stressed throughout the Thesis, the Failure Logic Metamodel has not been tailored to any particular specification language, it has been influenced by the prevailing modelling language paradigms of data flow and state machines. Consequently, considering failure logic modelling in the context of other paradigms may yield important insights into this approach to model-based safety assessment and highlight possible improvements. In particular, principles of constraint programming [129] appear to be well-aligned with the general philosophy of safety assessment. A view can be taken that 'safety models' should not attempt to model deterministic

behaviour of the system, but rather incrementally rule out non-viable or mutually excusive conditions and causal dependencies. Furthermore, the notion of channelling constraints [30], that enforce consistency between the models, appears to have natural interpretation in the context of composition of multiple failure logic models.

Another two modelling paradigms, which may provide an interesting alternative or complementary perspective of 'failure logic domain', are object-oriented (OO) modelling and aspect oriented programming (AOP). The later was previously highlighted by Joshi et al as a potential area for further research [77].

7.2.6 Other Model-Based Safety Assessment Approaches

This thesis has focused on a single family of model-based safety assessment methods demonstrating its applicability to modern safety-critical system as well as identifying some of the fundamental weaknesses of the approach as a whole. However, to the author's knowledge, no similar research has been conducted to-date on other model-based approaches. It is therefore desirable to formalise the conceptual framework underlying various existing approaches (such as failure injection, failure effects modelling and various forms of 'hybrid' techniques) in order to be able to systematically compare their relative strengths and weaknesses.

In practice it is expected that no single approach could be deemed as 'superior'. In the simplest form, different techniques will provide the best results at different stages of system design development process. A metamodel-based approach, if employed for formalisation of other techniques, will enable the correlation of different models used in safety assessment. This can significantly improve the confidence gained in the analysis results and overall safety of the system. Therefore, a promising area for further work is the formalisation of the metamodels underlying other prominent model-based safety assessment techniques and establishing the traceability between such metamodels and the FLMM presented in this thesis.

7.3 Coda

During the research presented in this thesis it has become apparent that failure logic modelling cannot be considered an approach for the 'automation' of safety analysis. The approach requires significant assessment of the system that cannot be simply decomposed to the level of individual components. This, however, should not be simplistically regarded as a limitation of failure logic modelling. By carrying out such assessment and constructing the models, safety engineers intellectually engage with the system design – gaining a thorough understanding of the system behaviour and its implications of safety.

Whilst automated safety assessment approaches may reduce assessment effort, on their own they are unlikely to yield such a degree of understanding of the system. In the words of 19th century English scientist and philosopher William Whewell [164], by relying solely on automated assessment and drawing confidence in results solely from the application of formal specification languages and verification techniques *"we are carried along as in a rail-road carriage, entering it at one station, and coming out of it at another, without having any choice in our progress in the intermediate space. It is plain that... [it] is not a mode of exercising our own locomotive powers; and in the same manner analytical processes [sic] are not a mode of exercising our reasoning powers."*

However, the author of this thesis believes that the automated approaches *are* capable of adding significant value to the safety assessment process. An example of their complimentary use can be identification of *inconsistencies* between safety analysis results (obtained by other means) and design models, which clearly require justification in the system safety case. Returning to Whewell, by adopting failure logic modelling that necessitates review of the system from a perspective that is conceptually dissimilar than that of design engineers *"we tread the ground ourselves at every step feeling ourselves firm."*

Appendix A:
Failure Logic Metamodel

This appendix presents the Failure Logic Metamodel (FLMM) as defined in Chapters 3, 4 and 5 of the thesis. Only the final version of the metamodel is shown. The appendix is organised as follows:

Section A1 specifies the FLMM defined under the Eclipse Modelling Framework (EMF) [46, 142]; the specification uses Emfatic syntax [47]

Section A2 presents well-formedness constraints, specified in the Eclipse Validation Language (EVL) [45, 86]

For convenience, Figure 104 below shows graphical a representation of the FLMM (as an Ecore Diagram automatically generated from the textual specification).

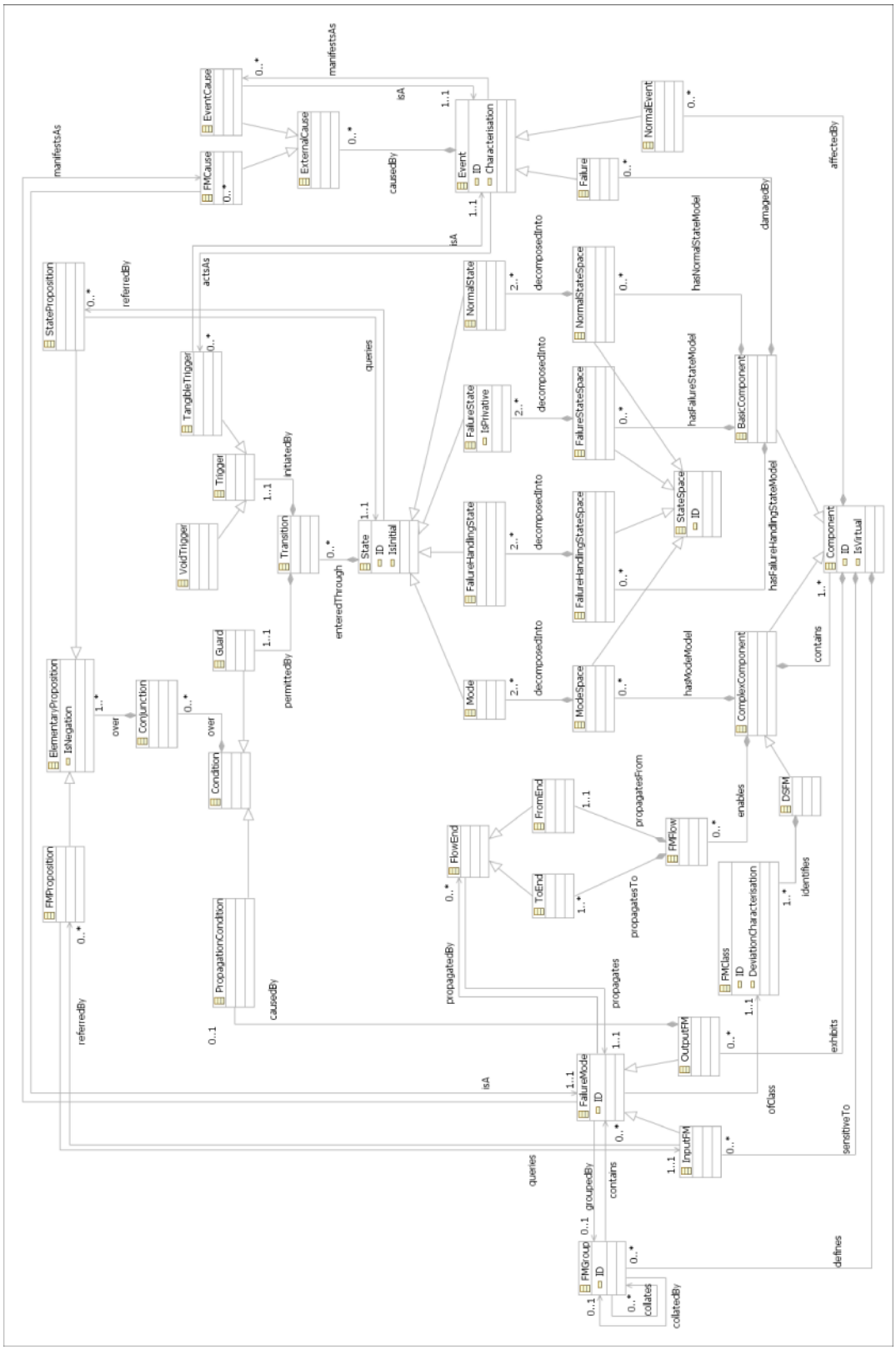


Figure 104 - Failure Logic Metamodel (Ecore Diagram)

A1. Metamodel Specification

```
@namespace(uri="lisagor/FLMM", prefix="flm")
package FLM_metamodel;

// ***********************************
// * Model Hierarchy Structure *
// ***********************************

abstract class Component {
  attr String ID;
  attr Boolean IsVirtual;
  val OutputFM[*] #of exhibits;
  val InputFM[*] #of sensitiveTo;
  val FMGroup[*] #definedIn defines;
  val NormalEvent[*] #affects affectedBy;
  ref ComplexComponent[0..1] #contains definedIn;
}

class BasicComponent extends Component {
  val Failure[*] #damages damagedBy;
  val FailureStateSpace[*] #represents hasFailureStateModel;
  val NormalStateSpace[*] #represents hasNormalStateModel;
  val FailureHandlingStateSpace[*] #represents hasFailureHandlingStateModel;
}

class ComplexComponent extends Component {
  val Component[+] #definedIn contains;
  val FMFlow[*] #enabledBy enables;
  val ModeSpace[*] #represents hasModeModel;
}

class DSFM extends ComplexComponent {
   val FMClass[+] #identifiedIn identifies;
}

// ************************************************
// * Component interface structure:          *
// *  Failure Modes, Classes and Groups  *
// ************************************************

class FMClass{
   attr String[1] ID;
   attr String DeviationCharacterisation;
   ref DSFM[1] #identifies identifiedIn;
}

abstract class FailureMode {
  attr String ID;
  ref FMClass[1] ofClass;
  ref FlowEnd[*] #propagates propagatedBy;
  ref FMGroup #contains groupedBy;
  ref FMCause[*] #isA manifestsAs;
}

class InputFM extends FailureMode {
  ref Component[1] #sensitiveTo of;
  ref FMProposition[*] #queries referredBy;
}
```

```
class OutputFM extends FailureMode {
  val PropagationCondition #of causedBy;
  ref Component[1] #exhibits of;
}

class FMGroup {
  attr String ID;
  ref FMGroup[*] #collatedBy collates;
  ref FMGroup[0..1] #collates collatedBy;
  ref FailureMode[*] #groupedBy contains;
  ref Component[1] #defines definedIn;
}

// ***************************************
// * Internal component structure: *
// *  - States and Events              *
// ***************************************

// *** Events ***
abstract class Event {
  attr String ID;
  attr String Characterisation;
  val ExternalCause[*] #causes causedBy;
  ref TangibleTrigger[*] #isA actsAs;
  ref EventCause[*] #isA manifestsAs;
}

class NormalEvent extends Event {
  ref Component[1] #affectedBy affects;
}

class Failure extends Event {
  ref BasicComponent[1] #damagedBy damages;
}

// *** State Spaces ***
abstract class StateSpace {
  attr String ID;
}

class FailureStateSpace extends StateSpace {
  val FailureState[2..*] #groupedBy decomposedInto;
  ref BasicComponent[1] #hasFailureStateModel represents;
}

class NormalStateSpace extends StateSpace {
  val NormalState[2..*] #groupedBy decomposedInto;
  ref BasicComponent[1] #hasNormalStateModel represents;
}

class FailureHandlingStateSpace extends StateSpace {
  val FailureHandlingState[2..*] #groupedBy decomposedInto;
  ref BasicComponent[1] #hasFailureHandlingStateModel represents;
}

class ModeSpace extends StateSpace {
  val Mode[2..*] #groupedBy decomposedInto;
  ref ComplexComponent[1] #hasModeModel represents;
}

// *** States ***
abstract class State {
  attr String ID;
  attr Boolean IsInitial;
```

```
  val Transition[*] #resultsIn enteredThrough;
  ref StateProposition[*] #queries referredBy;
}

class FailureState extends State {
  attr Boolean IsPrivative;
  ref FailureStateSpace[1] #decomposedInto groupedBy;
}

class NormalState extends State {
  ref NormalStateSpace[1] #decomposedInto groupedBy;
}

class FailureHandlingState extends State {
  ref FailureHandlingStateSpace[1] #decomposedInto groupedBy;
}

class Mode extends State {
  ref ModeSpace[1] #decomposedInto groupedBy;
}

// ************************************************
// * Component Behaviour:                         *
// *  - Propagation conditions and transitions *
// ************************************************

class PropagationCondition extends Condition {
  ref OutputFM[1] #causedBy of;
}

// *** State Transitions and their Structure ***
class Transition {
  val Trigger[1] #triggers initiatedBy;
  val Guard[1] #permits permittedBy;
  ref State[1] #enteredThrough resultsIn;
}

abstract class Trigger {
  ref Transition[1] #initiatedBy triggers;
}

class TangibleTrigger extends Trigger {
  ref Event[1] #actsAs isA;
}

class VoidTrigger extends Trigger {
}

class Guard extends Condition {
  ref Transition[1] #permittedBy permits;
}

// *** Common basis for propagation conditions and guards ***

abstract class Condition {
  val Conjunction[*] #inDisjunct over;
}

class Conjunction {
  val ElementaryProposition[+] #inConjunct over;
  ref Condition[1] #over inDisjunct;
}
```

```
abstract class ElementaryProposition {
  attr Boolean IsNegation;
  ref Conjunction[1] #over inConjunct;
}

class StateProposition extends ElementaryProposition {
  ref State[1] #referredBy queries;
}

class FMProposition extends ElementaryProposition {
  ref InputFM[1] #referredBy queries;
}

// ***************************************************************
// * FLM Architecture / cross-component dependencies: *
// *  - FM flow & their structure                                *
// *  - External causes of events (inter-DSFM links)       *
// ***************************************************************

// *** FM Flows (typical dependencies) ***
class FMFlow {
  val ToEnd[+] #initiatedBy propagatesTo;
  val FromEnd[1] #initiates propagatesFrom;
  ref ComplexComponent[1] #enables enabledBy;
}

class ToEnd extends FlowEnd {
  ref FMFlow[1] #propagatesTo initiatedBy;
}

class FromEnd extends FlowEnd {
  ref FMFlow[1] #propagatesFrom initiates;
}

abstract class FlowEnd {
  ref FailureMode[1] #propagatedBy propagates;
}

//*** External Causes (inter-DSFM dependencies ***
abstract class ExternalCause {
  ref Event[1] #causedBy causes;
}

class EventCause extends ExternalCause {
  ref Event[1] #manifestsAs isA;
}

class FMCause extends ExternalCause {
  ref FailureMode[1] #manifestsAs isA;
}
```

A2. Well-Formedness Constraints

```
// ******************************
// * Component Hierarchy *
// ******************************
context Component{
        // Every complex component that has "children"
        // must be a parent of these components
        constraint ConsistentDecomposition{
                guard: self.isTypeOf(ComplexComponent)
                check: self.contains.forAll(C : Component |
                                        C.definedIn = self)
}       // Only a DSFM can be an "orphan"
        constraint OnlyDSFMroot{
                check: (not self.definedIn.isDefined()) implies self.isTypeOf(DSFM)
}       // Every model must contain exactly one root (orphan) DSFM
        constraint SingleRootDSFM{
                check : DSFM.allInstances.select(D:DSFM |
                  not D.definedIn.isDefined()).size() = 1
}       // No complex component may contain itself
        constraint TreeStucture{
                check : not (self.definedIn = self)
}       }

// ******************************
// * Structure of FM Flows *
// ******************************
context FailureMode{
        // Every FM can be connected to at most one "inflow"
        constraint AtMostOneFlowIn{
                check: self.propagatedBy.select(E: FlowEnd|
                        E.isTypeOf(ToEnd)).size() < 2
}       // It is unusual for FM flow to be connected to
        // more than one "outflow" ("soft" constraint)
        critique AtMostOneFlowOut{
                check: self.propagatedBy.select(E: FlowEnd|
                        E.isTypeOf(FromEnd)).size() < 2
}       }
context InputFM{
        // Only top-level DSFM may have free input FMs
        constraint FreeInputsOnlyAtRoot{
                check: (self.propagatedBy.select(E: FlowEnd|
                        E.isTypeOf(ToEnd)).size() = 0)
                        implies ( self.of.isTypeOf(DSFM)
                         and (not self.of.definedIn.isDefined()))
}       }
context OutputFM{
        // Output FMs of complex components
        // must be connected to an inflow (from the "inside")
        constraint ComplexOutputFMConnected{
                guard: self.of.isTypeOf(ComplexComponent)
                check: self.propagatedBy.select(E: FlowEnd|
                        E.isTypeOf(ToEnd)).size() = 1
}       }
```

```
context ToEnd{
        // FM Flows do not transform failure modes
        // in transit (equivalent to strong typing)
        constraint StronglyTypedFlows{
                check: self.propagates.ofClass =
                  self.initiatedBy.propagatesFrom.propagates.ofClass
}       // FM flows should not connect a component to itself
        constraint NoImmediateLoop{
                check: not (self.propagates.of =
                  self.initiatedBy.propagatesFrom.propagates.of)
}       }
context FMFlow{
        // Any Flow from an input must be declared within the same
        // complex component (as the input FM)
        constraint FlowsFromInputsValidLevel{
                check: self.propagatesFrom.propagates.isTypeOf(InputFM)
                  implies (self.propagatesFrom.propagates.of = self.enabledBy)

}       // Any Flow from an input must lead exclusively to inputs of
        // components declared at the same level (within the same complex
        // component) as the flow
        constraint FlowsFromInputsValidTarget{
                guard: self.satisfies('FlowsFromInputsValidLevel')
                check: self.propagatesFrom.propagates.isTypeOf(InputFM)
                  implies (self.propagatesTo.forAll(E : ToEnd|
                    E.propagates.of.definedIn = self.enabledBy))
}       // Any flow from an output must be declared at the same
        // level (within the same parent) as the source component
        constraint FlowsFromOutputsValidLevel{
                check: self.propagatesFrom.propagates.isTypeOf(OutputFM)
                  implies (self.propagatesFrom.propagates.of.definedIn = self.enabledBy)
}       // Any Flow from an output may only lead to an output of the parent component
        // or in input of a sibling component
        constraint FlowsFromInputsValidTarget{
                guard: self.satisfies('FlowsFromOutputsValidLevel')
                check: self.propagatesFrom.propagates.isTypeOf(OutputFM)
                  implies self.propagatesTo.forAll(E : ToEnd |
                    ((E.propagates.of.definedIn = self.enabledBy) and
                     (E.propagates.isTypeOf(InputFM))) or
                    ((E.propagates.of = self.enabledBy) and
                     (E.propagates.isTypeOf(OutputFM))))
}}

// ***********************************
// * Failure Modes and Groups *
// ***********************************
context FailureMode{
        // Only Output FMs of basic components
        // may be associated with a propagation condition
        constraint OnlyBasicFMwithLogic{
                guard: self.isTypeOf(OutputFM)
                check: self.causedBy.size() > 0 implies self.of.isTypeOf(BasicComponent)
}       // If FMs are grouped then the group must be
        // defined in the same component
        constraint ValidGroupingFM{
                guard: self.groupedBy.isDefined()
                check: self.of = self.groupedBy.definedIn
}}
```

```
context FMGroup{
        // If FM Group is further grouped then both
        // groups must be defined in the same component
        constraint ValidGroupHierarchy{
                guard: self.collatedBy.isDefined()
                check : self.definedIn = self.collatedBy.definedIn
}       // Each FM Group should contain at least one FM or another group
        constraint NonEmptyGroup{
                check: self.collates.isDefined() or self.contains.isDefined()
}}

// *********************************
// * States and State Spaces *
// *********************************
context StateSpace{
        // Each state must contain exactly one initial state
        constraint InitialStateIdentified{
                check: self.decomposedInto.select(S: State | S.IsInitial).size() = 1
}       // It is unusual for a non-initial state not to have
        // at least one transition (into itself)
        critique AvoidOrphanStates{
                check: self.decomposedInto.forAll(S: State |
                  (not S.IsInitial) implies S.enteredThrough.size() > 0)
}       }

// ***********************
// * State Transitions *
// ***********************
context Transition{
        // If transition has a void trigger its guard must contain
        // Note: this is a weak contraint that doesn't address all
        // scenarios of runaway transitions (a loop of transitions
        // initiated by void guards)
        constraint NoRunawayTransitions{
                check: self.initiatedBy.isTypeOf(VoidTrigger) implies
                        self.permittedBy.over.size() > 0
}       // It is unusual for a failure state to be entered through
        // entirely normal circumstances (e.g. no failure, no
        // exposure to a failure mode) while component is still
        // in the initial state
        critique FailureStateEntry{
                check: (self.initiatedBy.isTypeOf(VoidTrigger) or
                  (self.initiatedBy.isTypeOf(TangibleTrigger) and
                   self.initiatedBy.isA.isTypeOf(NormalEvent)))
                     implies (self.permittedBy.over.forAll(C:Conjunction |
                             C.over.exists(P:Proposition |
                               (P.isTypeOf(StateProposition) and
                                 ((P.IsNegation and P.queries.IsInitial) or
                                  (not P.IsNegation and not P.queries.IsInitial))) or
                               (P.isTypeOf(FMProposition) and not P.IsNehation))))
}       // It is unusual for Failure Handling State to be triggered by
        // a failure
        critique FailureHandlStateEntry{
                check: self.resultsIn.isTypeOf(FailureHandlingState)
                  implies not self.initiatedBy.isA.isTypeOf(Failure)
}}
```

```
// ************************
// * External Causes *
// ************************
context ExternalCause{
        // An event cannot cause itself
        // (i.e. no immediate circular dependency)
        constraint CanNotCauseItself{
                guard: self.isTupeOf(EventCause)
                check: not (self.isA = self.causes)
}       // It is higly unusual for an external cause
        // of the event to come from the same model
        // as the event itself (its not really very
        // "external" then)
        critique CauseAndEffectInDifferentModels{
                check: not (self.causes.theDaddyDSFM() = self.isA.theDaddyDSFM())
}}

// ***********************************************
// * Visibility Constraints                      *
// * (see also operations defined below) *
// ***********************************************
context FMProposition{
        constraint ValidFMProposition{
                check: self.inConjunct.inDisjunct.theDaddy().CollectVisibleInputFMs().includes(self.queries)
}       // Whilst valid it is highly unusual for FM proposition to refer to non-local FM
        critique LocalFMProposition{
                check: self.inConjunct.inDisjunct.theDaddy() = self.queries.of
}}
context StateProposition{
        constraint ValidStateProposition{
                check: self.inConjunct.inDisjunct.theDaddy().CollectVisibleStates().includes(self.queries)
}}
context TangibleTrigger{
        constraint ValidTrigger{
                check: self.theDaddy().CollectVisibleEvents().includes(self.isA)
}}


//  Shared Operations
// --------------------------
// ************************************
// * Collecting elements visible    *
// * at the level of the component *
// ************************************

operation Component CollectVisibleStates(){
        // At the level of component states of the same
        // component and (modes of) all of its parents are visible
        var States : Collection;
        if (self.isTypeOf(BasicComponent))
          {States := self.hasFailureStateModel.collect(c|c.decomposedInto) +
            self.hasFailureHandlingStateModel.collect(c|c.decomposedInto) +
            self.hasNormalStateModel.collect(c|c.decomposedInto) +
            self.definedIn.CollectVisibleStates();}
        else {if (self.definedIn.isDefined())
                {States := self.hasModeModel.collect(c|c.decomposedInto) +
                  self.definedIn.CollectVisibleStates();}
              else {States := self.hasModeModel.collect(c|c.decomposedInto);}}
        return States.flatten();}
```

```
operation Component CollectVisibleEvents(){
        // At the level of component events of the same component
        // and all of its children are visible
        var Events : Collection;
        if (self.isTypeOf(BasicComponent))
                {Events := self.damagedBy + self.affectedBy;}
        else
                {Events := self.affectedBy +
                 self.contains.collect(C|C.CollectVisibleEvents());}
        return Events.flatten();}

operation Component CollectVisibleInputFMs(){
        // At the level of component input FMs of the same component
        // and all of its children are visible
        var FMs : Collection;
        if (self.isTypeOf(BasicComponent))
                {FMs := self.sensitiveTo;}
        else
                {FMs := self.sensitiveTo +
                 self.contains.collect(C|C.CollectVisibleInputFMs());}
        return FMs.flatten();}

// ************************************************
// * Shortcut for finding the component     *
// * in which logical expression is defined *
// ************************************************
operation Trigger theDaddy(){
return self.triggers.resultsIn.groupedBy.represents;}

operation Guard theDaddy(){
return self.permits.resultsIn.groupedBy.represents;}

operation PropagationCondition theDaddy(){
return self.of.of;}

// ****************************************************
// * Shortcut for finding the "next up" DSFM *
// ****************************************************
operation Component theDaddyDSFM(){
if (self.isTypeOf(DSFM))
{return self;}
else {return self.definedIn.theDaddyDSFM;}}

operation NormalEvent theDaddyDSFM(){
return self.affects.theDaddyDSFM();}

operation Failure theDaddyDSFM(){
return self.damages.theDaddyDSFM();}

operation FailureMode theDaddyDSFM(){
return self.of.theDaddyDSFM();}
```

Appendix B:

Wheel Braking System Case Study

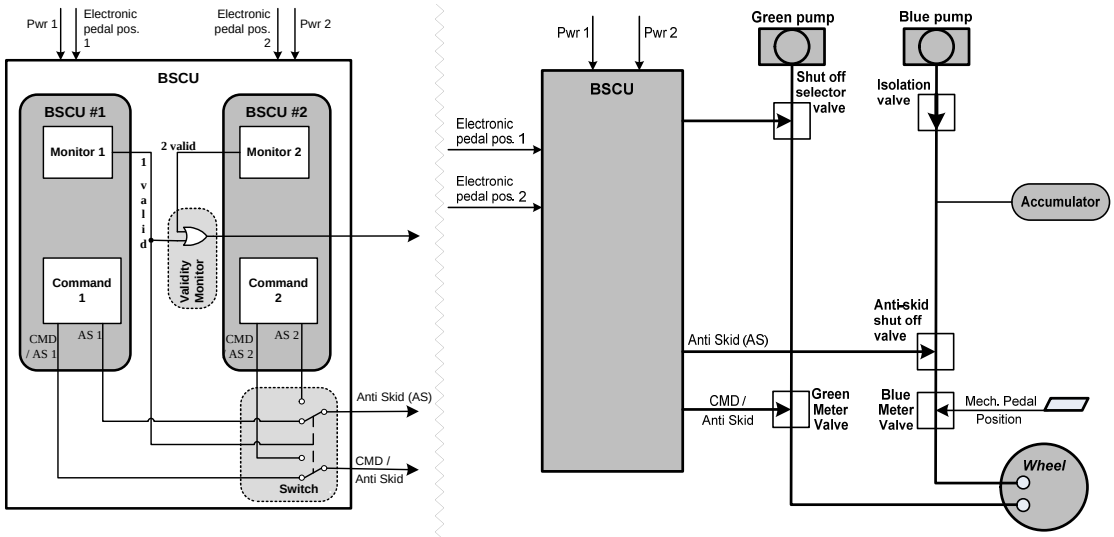
This appendix presents further details of the Wheel Braking System (WBS), case study complementing discussions in Chapters Three and Five of the thesis. The system design is closely related to the presentation in Appendix L of ARP 4761 document [139], although it has been simplified, and is described in Section 3.1.1 (Chapter Three). This appendix is organised as follows:

Section A1 presents a complete failure logic model of the WBS recorded in an informal tabular pseudocode format (as referred to in sections 3.2 through 3.4 of the Thesis). For brevity the section focuses on component characterisations; FM Flows are only shown between components within a single BSCU side.

Section A2 presents the same model recorded in AltaRica Dataflow (as referred to in Section 3.6 of the main body of the thesis)

Section A3 presents a modified model of the Braking System Control Unit (BSCU), recorded in AltaRica, which captures design limitations and reconfiguration modes of the BSCU (as referred to in section 5.2 of the Thesis)

For convenience, Figure 105 below presents the architecture of the WBS and the Control Unit (reproducing Figure 20 and Figure 21 of the Chapter Three):



(a) Braking System Control Unit *(b) Top-Level WBS Architecture*

Figure 105 - Architecture of the Hypothetical Aircraft Wheel Braking System

B2. WBS Failure Logic Model: “Pseudo-code”

B2.1 Braking System Control Unit

COM 1(2)
[in BSCU1(2) in BSCU]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
Pedal1.DemandLow	LowValue	Pedal1
Pedal1.DemandOmiss	Omission	Pedal1
Pedal1.DemandCommiss	Commission	Pedal1
Pedal2.DemandLow	LowValue	Pedal2
Pedal2.DemandOmiss	Omission	Pedal2
Pedal2.DemandCommiss	Commission	Pedal2
PowerOmiss	Omission	PWR

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
ASprocessStuck	n/a	*(none)*	
ASprocessTerminated	n/a		
ProcessorError	n/a		
CMDprocessStuck	n/a		
CMDprocessTerminated	n/a		

Failure States & Entry Logic

Failure State	Trigger	Guard
CalculationError (1)	ProcessorError	~(PowerOmiss | CalculationError | CMDstuckOn | CMDstuckOff)
ASstuckOn (2)	ASprocessStuck	~(PowerOmiss | ASstuckOff)
ASstuckOff (2)	ASprocessTerminated	~PowerOmiss
CMDstuckOn (1)	CMDprocessStuck	~(PowerOmiss | CMDstuckOff)
CMDstuckOff (1)	CMDprocessTerminated	~PowerOmiss

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
InadvertentBraking	Commission	CMD	((Pedal1.DemandCommiss | Pedal2.DemandCommiss) & ~(CMDstuckOff | PowerOmiss)) | (CMDstuckOn & ~PowerOmiss)
LackOfBraking	Omission	CMD	(Pedal1.DemandOmiss & Pedal2.DemandOmiss & ~CMDstuckOn) | CMDstuckOff | PwrOmiss
LackOfAntiskid	HighValue	CMD	ASstuckOff & ~(CMDstuckOn | CMDstuckOn | Pedal1.DemandCommiss | Pedal2.DemandCommiss | Pedal1.DemandOmiss | Pedal2.DemandOmiss | PowerOmiss)
TooLittleBraking	LowValue	CMD	(ASstuckOn | CalculationError | Pedal1.DemandOmiss | Pedal2.DemandOmiss | Pedal1.DemandLow | Pedal2.DemandLow) & ~(CMDstuckOn | CMDstuckOn | Pedal1.DemandCommiss | Pedal2.DemandCommiss | Pedal1.DemandOmiss & Pedal2.DemandOmiss | PowerOmiss)
AntiSkidOmiss	Omission	AS	ASstuckOff | PowerOmiss
AntiSkidCommiss	Commission	AS	ASstuckOn & ~PowerOmiss

MON 1(2)
[in BSCU1(2) in BSCU]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
InadvertentBraking	Commission	CMD
LackOfBraking	Omission	CMD
LackOfAntiskid	HighValue	CMD
TooLittleBraking	LowValue	CMD
Pedal1.DemandLow	LowValue	Pedal1
Pedal1.DemandOmiss	Omission	Pedal1
Pedal1.DemandCommiss	Commission	Pedal1
Pedal2.DemandLow	LowValue	Pedal2
Pedal2.DemandOmiss	Omission	Pedal2
Pedal2.DemandCommiss	Commission	Pedal2
PowerOmiss	Omission	PWR

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
ProcessStuck	n/a	*(none)*	
ProcessTerminated	n/a		
ProcessorError	n/a		

Failure States & Entry Logic

Failure State	Trigger	Guard
StuckPos	ProcessorError	~(PowerOmiss | StuckPos | StuckNeg)
	ProcessTerminated	~PowerOmiss
StuckNeg	ProcessStuck	~(PowerOmiss | StuckPos | StuckNeg)

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
FalseNeg	Commission	Validity	(Pedal1.DemandLow & Pedal2.DemandLow & TooLittleBraking | Pedal1.DemandOmiss & Pedal2.DemandOmiss & LackOfBraking | Pedal1.DemandCommiss & Pedal2.DemandCommiss & InadvertentBraking | StuckNeg) & ~(StuckPos | PowerOmiss)
FalsePos	Omission	Validity	StuckPos & ~(Pedal1.DemandLow | Pedal2.DemandLow | TooLittleBraking | Pedal1.DemandOmiss | Pedal2.DemandOmiss | LackOfBraking | Pedal1.DemandCommiss | Pedal2.DemandCommiss | InadvertentBraking | LackOfAntiskid | PowerOmiss)

BSCU #1 (#2) [COMPLEX COMPONENT in BSCU]		
Input Failure Modes		
Failure Mode (ID)	**FM Class**	**FM Group**
Pedal1.DemandLow	LowValue	Pedal1
Pedal1.DemandOmiss	Omission	Pedal1
Pedal1.DemandCommiss	Commission	Pedal1
Pedal2.DemandLow	LowValue	Pedal2
Pedal2.DemandOmiss	Omission	Pedal2
Pedal2.DemandCommiss	Commission	Pedal2
PowerOmiss	Omission	PWR
Output Failure Modes		
Failure Mode (ID)	**FM Class**	**FM Group**
InadvertentBraking	Commission	CMD
LackOfBraking	Omission	CMD
LackOfAntiskid	HighValue	CMD
TooLittleBraking	LowValue	CMD
AntiSkidOmiss	Omission	AS
AntiSkidCommiss	Commission	AS
FalseNeg	Commission	Validity
FalsePos	Omission	Validity

From			To		
Component ID	**Component Type**	**Failure Mode (incl. group)**	**Component ID**	**Component Type**	**Failure Mode (incl. group)**
BSCU1 (2)	Complex / Parent	Pedal1.DemandLow	COM1 (2)	Basic	Pedal1.DemandLow
			MON1 (2)	Basic	Pedal1.DemandLow
		Pedal1.DemandOmiss	COM1 (2)	Basic	Pedal1.DemandOmiss
			MON1 (2)	Basic	Pedal1.DemandOmiss
		Pedal1.DemandCommiss	COM1 (2)	Basic	Pedal1.DemandCommiss
			MON1 (2)	Basic	Pedal1.DemandCommiss
		Pedal2.DemandLow	COM1 (2)	Basic	Pedal2.DemandLow
			MON1 (2)	Basic	Pedal2.DemandLow
		Pedal2.DemandOmiss	COM1 (2)	Basic	Pedal2.DemandOmiss
			MON1 (2)	Basic	Pedal2.DemandOmiss
		Pedal2.DemandCommiss	COM1 (2)	Basic	Pedal2.DemandCommiss
			MON1 (2)	Basic	Pedal2.DemandCommiss
		PWR.PowerOmiss	COM1 (2)	Basic	PWR.PowerOmiss
			MON1 (2)	Basic	PWR.PowerOmiss
COM1 (2)	Basic	CMD.InadvertentBraking	MON1 (2)	Basic	CMD.InadvertentBraking
			BSCU1 (2)	Complex / Parent	CMD.InadvertentBraking
		CMD.LackOfBraking	MON1 (2)	Basic	CMD.LackOfBraking
			BSCU1 (2)	Complex / Parent	CMD.LackOfBraking
		CMD.TooLittleBraking	MON1 (2)	Basic	CMD.TooLittleBraking
			BSCU1 (2)	Complex / Parent	CMD.TooLittleBraking
		CMD.LackOfAntiskid	MON1 (2)	Basic	CMD.LackOfAntiskid
			BSCU1 (2)	Complex / Parent	CMD.LackOfAntiskid
		AS.AntiSkidOmiss	BSCU1 (2)	Complex / Parent	AS.AntiSkidOmiss
		AS.AntiSkidCommiss	BSCU1 (2)	Complex / Parent	AS.AntiSkidCommiss
MON1 (2)	Basic	Validity.FalsePos	BSCU1 (2)	Complex / Parent	Validity.FalsePos
		Validity.FalseNeg	BSCU1 (2)	Complex / Parent	Validity.FalseNeg

Validity Monitor
[in BSCU]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
In1.FalsePos	Omission	In1
In1.FalseNeg	Commission	In1
In2.FalsePos	Omission	In2
In2.FalseNeg	Commission	In2
PWR1.PowerOmiss	Omission	PWR1
PWR2.PowerOmiss	Omission	PWR2

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
ProcessStuck	n/a	*(none)*	
ProcessTerminated	n/a		

Failure States & Entry Logic

Failure State	Trigger	Guard
StuckPos	ProcessTerminated	~(PWR1.PowerOmiss & PWR2.PowerOmiss)
StuckNeg	ProcessStuck	~(PWR1.PowerOmiss & PWR2.PowerOmiss) & ~StuckPos

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
FalsePos	Omission	Validity	(In1.FalsePos | In2.FalsePos | StuckPos) & ~StuckNeg & ~(PWR1.PowerOmiss & PWR2.PowerOmiss)
FalseNeg	Commission	Validity	(In1.FalseNeg | In2.FalseNeg | StuckNeg) & ~StuckPos & ~(PWR1.PowerOmiss & PWR2.PowerOmiss)

Switch
[in BSCU]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
CMD1.InadvertentBraking	Commission	CMD1
CMD1.LackOfBraking	Omission	CMD1
CMD1.LackOfAntiskid	HighValue	CMD1
CMD1.TooLittleBraking	LowValue	CMD1
AS1.AntiSkidOmiss	Omission	AS1
AS1.AntiSkidCommiss	Commission	AS1
CMD2.InadvertentBraking	Commission	CMD2
CMD2.LackOfBraking	Omission	CMD2
CMD2.LackOfAntiskid	HighValue	CMD2
CMD2.TooLittleBraking	LowValue	CMD2
AS2.AntiSkidOmiss	Omission	AS2
AS2.AntiSkidCommiss	Commission	AS2
FalsePos	Omission	Validity
FalseNeg	Commission	Validity

Output Failure Modes & Propagation Conditions			
Failure Mode (ID)	**FM Class**	**FM Group**	**Propagation Condition**
InadvertentBraking	Commission	CMD	(CMD1.InadvertentBraking & FalseNeg) | (CMD2.InadvertentBraking & (FalsePos | ~FalseNeg & (CMD1.InadvertentBraking | CMD1.LackOfBraking | CMD1.LackOfAntiskid | CMD1.TooLittleBraking)))
LackOfBraking	Omission	CMD	(CMD1.LackOfBraking & FalseNeg) | (CMD2.LackOfBraking & (FalsePos | ~FalseNeg & (CMD1.InadvertentBraking | CMD1.LackOfBraking | CMD1.LackOfAntiskid | CMD1.TooLittleBraking)))
LackOfAntiskid	HighValue	CMD	(CMD1.LackOfAntiskid & FalseNeg) | (CMD2.LackOfAntiskid & (FalsePos | ~FalseNeg & (CMD1.InadvertentBraking | CMD1.LackOfBraking | CMD1.LackOfAntiskid | CMD1.TooLittleBraking)))
TooLittleBraking	LowValue	CMD	(CMD1.TooLittleBraking & FalseNeg) | (CMD2.TooLittleBraking & (FalsePos | ~FalseNeg & (CMD1.InadvertentBraking | CMD1.LackOfBraking | CMD1.LackOfAntiskid | CMD1.TooLittleBraking)))
AntiSkidOmiss	Omission	AS	(AS1.AntiSkidOmiss & FalseNeg) | (AS2.AntiSkidOmiss & (FalsePos | ~FalseNeg & (CMD1.InadvertentBraking | CMD1.LackOfBraking | CMD1.LackOfAntiskid | CMD1.TooLittleBraking)))
AntiSkidCommiss	Commission	AS	(AS1.AntiSkidCommiss & FalseNeg) | (AS2.AntiSkidCommiss & (FalsePos | ~FalseNeg & (CMD1.InadvertentBraking | CMD1.LackOfBraking | CMD1.LackOfAntiskid | CMD1.TooLittleBraking)))

BSCU #1 (#2) [COMPLEX COMPONENT]		
Input Failure Modes		
Failure Mode (ID)	**FM Class**	**FM Group**
Pedal1.DemandLow	LowValue	Pedal1
Pedal1.DemandOmiss	Omission	Pedal1
Pedal1.DemandCommiss	Commission	Pedal1
Pedal2.DemandLow	LowValue	Pedal2
Pedal2.DemandOmiss	Omission	Pedal2
Pedal2.DemandCommiss	Commission	Pedal2
PowerOmiss	Omission	PWR1
PowerOmiss	Omission	PWR2
Output Failure Modes		
Failure Mode (ID)	**FM Class**	**FM Group**
InadvertentBraking	Commission	CMD
LackOfBraking	Omission	CMD
LackOfAntiskid	HighValue	CMD
TooLittleBraking	LowValue	CMD
AntiSkidOmiss	Omission	AS
AntiSkidCommiss	Commission	AS
FalseNeg	Commission	Validity
FalsePos	Omission	Validity

B1.2 Hydro-Mechanical Components

Blue (Green) Pump			
Input Failure Modes			
Failure Mode (ID)		**FM Class**	**FM Group**
Leak		Commission	Out.BWD
Events			
Failure (ID)	**Probability Characterisation**	**Normal Event (ID)**	**Comments**
MechanicalFailure			
Contamination			
Rupture			
Failure States & Entry Logic			
Failure State	**Trigger**	**Guard**	
TotalLoss	MechanicalFailure	~Leaking	
	<Void>	Leak & ~Leaking	
Struggling	Contamination	~Leaking & ~TotalLoss	
Leaking	Rupture	*(true)*	
Output Failure Modes & Propagation Conditions			
Failure Mode (ID)	**FM Class**	**FM Group**	**Propagation Condition**
PressureOmission	Omission	Out.FWD	TotalLoss
TooLowPressure	LowValue	Out.FWD	Struggling
Leak	Commission	Out.FWD	Leaking

Shut-off Selector Valve
[Green Channel]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
FWD.Leak	Commission	HydOut.FWD
PressureOmission	Omission	HydIn.FWD
TooLowPressure	LowValue	HydIn.FWD
FalsePos	Omission	CTRL
FalseNeg	Commission	CTRL
BWD.Leak	Commission	HydOut.BWD

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
SpringFailure			
UnlocksSpontaneously			
Rupture			

Failure States & Entry Logic

Failure State	Trigger	Guard
StuckOpen (1)	SpringFailure	~StuckClosed
StuckClosed (1)	UnlocksSpontaneously	~StuckOpen
Leaking (2)	Rupture	*(true)*

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
Leak	Commission	HydIn.BWD	Leaking | BWD.Leak & ~StuckClosed & (StuckOpen | ~FalsePos)
PressureOmission	Omission	HydOut.FWD	StuckClosed | Leaking | PressureOmission | FWD.Leak | (FalsePos & ~StuckOpen)
PressureCommission	Commission	HydOut.FWD	(StuckOpen | FalseNeg) & ~StuckClosed & ~Leaking & ~PressureOmission & ~ FWD.Leak & ~TooLowPressure
LowPressureCommiss	LowCommission	HydOut.FWD	(StuckOpen | FalseNeg) & TooLowPressure & ~StuckClosed & ~Leaking & ~PressureOmission & ~ FWD.Leak
TooLowPressure	LowValue	HydOut.FWD	TooLowPressure & ~StuckClosed & ~StuckOpen & ~Leaking & ~PressureOmission & ~ FWD.Leak & ~FalsePos & ~FalseNeg

Green Meter Valve

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
Leak	Commission	HydOut.BWD
PressureOmission	Omission	HydIn.FWD
PressureCommission	Commission	HydIn.FWD
LowPressureCommiss	LowCommission	HydIn.FWD
TooHighPressure	TooHightValue	HydIn.FWD
TooLowPressure	LowValue	HydIn.FWD
InadvertentBraking	Commission	CTRL
LackOfBraking	Omission	CTRL
LackOfAntiskid	HighValue	CTRL
TooLittleBraking	LowValue	CTRL

Events			
Failure (ID)	**Probability Characterisation**	**Normal Event (ID)**	**Comments**
SpringFailure			
JamClosed			
JamOpen			
MotorFailure			
Contamination			
Rupture			

Failure States & Entry Logic		
Failure State	**Trigger**	**Guard**
StuckOpen (1)	SpringFailure	~StuckClosed
	JamOpen	~StuckClosed
StuckClosed (1)	MotorFailure	~StuckOpen
	JamClosed	~StuckOpen
Struggling (1)	Contamination	~StuckClosed & ~StuckOpen
Leaking (2)	Rupture	*(true)*

Output Failure Modes & Propagation Conditions			
Failure Mode (ID)	**FM Class**	**FM Group**	**Propagation Condition**
Leak	Commission	HydIn.BWD	Leaking | Leak & ~StuckClosed & (StuckOpen | ~LackOfBraking)
PressureOmission	Omission	HydOut.FWD	Leaking |StuckClosed | PressureOmission | (LackOfBraking & (~StuckOpen | ~PressureCommission)) | (InadvertentBraking | LackOfAntiskid | TooLittleBraking) & ~(PressureCommission | LowPressureCommiss)
PressureCommission	Commission	HydOut.FWD	(StuckOpen | InadvertentBraking) & ~StuckClosed &~Leaking & ~PressureOmission & (PressureCommission | LowPressureCommiss | ~ (InadvertentBraking | LackOfBraking | LackOfAntiskid | TooLittleBraking))
TooHighPressure	TooHightValue	HydOut.FWD	(LackOfAntiskid & PressureCommission & ~StuckClosed & ~StuckOpen & ~Leaking) | (Struggling & ~StuckClosed & ~StuckOpen &~Leaking & ~PressureOmission & ~TooLowPressure & ~LowPressureCommiss & ~LackOfBraking & ~InadvertentBraking & ~LackOfAntiskid & ~TooLittleBraking)
TooLowPressure	LowValue	HydOut.FWD	(TooLittleBraking & PressureCommission & ~StuckClosed & ~StuckOpen & ~Leaking) | (LowPressureCommiss & ~InadvertentBraking & ~LackOfBraking & ~StuckClosed & ~StuckOpen & ~Leaking)

Isolation Valve

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
PressureOmission	Omission	In.FWD
TooLowPressure	LowValue	In.FWD
In.FWD.Leak	Commission	In.FWD
Out.BWD.Leak	Commission	Out.BWD

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
Contamination			
Jam			
Rupture			

Failure States & Entry Logic

Failure State	Trigger	Guard
StuckClosed (1)	Contamination	*(true)*
StuckOpen (1)	Jam	(PressureOmission | In.FWD.Leak) & ~StuckClosed
Leaking (2)	Rupture	*(true)*

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
PressureOmission	Omission	Out.FWD	StuckClosed & ~Leaking | PressureOmission & ~Leaking | In.FWD.Leak & ~StuckOpen
TooLowPressure	LowValue	Out.FWD	TooLowPressure & ~StuckClosed & ~StuckOpen & ~Leaking
In.BWD.Leak	Commission	In.BWD	Leaking | Out.BWD.Leak & ~StuckClosed
Out.FWD.Leak	Commission	Out.FWD	Leaking | In.FWDLeak & StuckOpen

Accumulator
[Blue Channel]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
PressureOmission	Omission	In
TooLowPressure	LowValue	In
Leak	Commission	In

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
ConnectorBlocked		UsedOnce	Normal event Assume probability = 1
PartialDecompression			
CompleteDecompression			
Rupture			

Normal States & Entry Logic

Normal State	Trigger	Guard
Empty	UsedOnce	Full & PressureOmission & ~Isolated & ~TotalLoss & ~Leaking
Full	[Initial state, no re-entry]	

Failure States & Entry Logic

Failure State	Trigger	Guard
Isolated	ConnectorBlocked	*(true)*
TotalLoss	<void>	Leak & ~Leaking & ~Isolated
	CompleteDecompression	~Leaking & ~Isolated
Leaking	Rupture	~Isolated
PartialLoss	PartialDecompression	~Empty & ~TotalLoss & ~Leaking & ~Isolated

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
PressureOmission	Omission	Out	Empty | Isolated | TotalLoss
TooLowPressure	LowValue	Out	PartialLoss & Full & ~Isolated & ~TotalLoss & ~Leaking
Leak	Commission	Out	Leaking

Pipe Junction
[Blue Channel]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
Pump.FWD.PressureOmission	Omission	Pump.FWD
Pump.FWD.TooLowPressure	LowValue	Pump.FWD
Pump.FWD.Leak	Commission	Pump.FWD
Accum.FWD.PressureOmission	Omission	Accum.FWD
Accum.FWD.TooLowPressure	LowValue	Accum.FWD
Accum.FWD.Leak	Commission	Accum.FWD
Out.BWD.Leak	Commission	Out.BWD

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
Pump.BWD.Leak	Commission	Pump.BWD	Accum.FWD.Leak | Out.BWD.Leak
Accum.BWD.Leak	Commission	Accum.BWD	Pump.FWD.Leak | Out.BWD.Leak
Accum.BWD. PressureOmission	Omission	Accum.BWD	Pump.FWD.PressureOmission &~Out.BWD.Leak
PressureOmission	Omission	Out.FWD	Accum.FWD.Leak | Pump.FWD.Leak | Pump.FWD.PressureOmission & Accum.FWD.PressureOmission
TooLowPressure	LowValue	Out.FWD	Pump.FWD.TooLowPressure & ~Accum.FWD.Leak | Pump.FWD.PressureOmission & Accum.FWD.TooLowPressure

Mech. Pedal Position (link)

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
DemandLow	LowValue	Pedal
DemandOmiss	Omission	Pedal
DemandCommiss	Commission	Pedal

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
Contamination			
Jam			
MechanicalFailure			

Failure States & Entry Logic

Failure State	Trigger	Guard
LimitedAmplitude	Contamination	~StuckUp & ~StuckDown
StuckUp	MechanicalFailure	
StuckDown	Jam	~StuckUp

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
InadvertentBraking	Commission	CMD	DemandCommiss & ~StuckUp | StuckDown
LackOfBraking	Omission	CMD	DemandOmiss & ~StuckDown | StuckUp
TooLittleBraking	LowValue	CMD	(DemandLow | LimitedAmplitude) & ~StuckDown & ~StuckUp & ~DemandOmiss & ~DemandCommiss

AS Shut-off Valve
[Blue Channel]

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
Leak	Commission	HydOut.BWD
PressureOmission	Omission	HydIn.FWD
TooLowPressure	LowValue	HydIn.FWD
AntiSkidOmiss	Omission	CTRL
AntiSkidCommiss	Commission	CTRL

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
LoadFailure			
JamOpen			
MotorFailure			
Contamination			
Rupture			

Failure States & Entry Logic

Failure State	Trigger	Guard
StuckOpen (1)	MotorFailure	~StuckClosed
	JamOpen	~StuckClosed
StuckClosed (1)	LoadFailure	~StuckOpen
Struggling (1)	Contamination	~StuckClosed & ~StuckOpen
Leaking (2)	Rupture	*(true)*

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
Leak	Commission	HydIn.BWD	Leaking | Leak & ~StuckClosed
PressureOmission	Omission	HydOut.FWD	StuckClosed | Leaking | PressureOmission
TooHighPressure	TooHightValue	HydOut.FWD	(AntiSkidOmiss | StuckOpen | Struggling) & ~PressureOmission & ~TooLowPressure & ~Leaking & ~StuckClosed
TooLowPressure	LowValue	HydOut.FWD	(TooLowPressure | AntiSkidCommiss & ~StuckOpen) & ~PressureOmission & ~StuckClosed & ~Leaking

Blue Meter Valve

Input Failure Modes

Failure Mode (ID)	FM Class	FM Group
Leak	Commission	HydOut.BWD
PressureOmission	Omission	HydIn.FWD
TooHighPressure	TooHightValue	HydIn.FWD
TooLowPressure	LowValue	HydIn.FWD
InadvertentBraking	Commission	CTRL
LackOfBraking	Omission	CTRL
TooLittleBraking	LowValue	CTRL

Events

Failure (ID)	Probability Characterisation	Normal Event (ID)	Comments
SpringFailure (1)			
JamClosed (1)			
JamOpen (1)			
MotorFailure (1)			
Rupture (2)			

Failure States & Entry Logic

Failure State	Trigger	Guard
StuckOpen (1)	SpringFailure	~StuckClosed
	JamOpen	~StuckClosed
StuckClosed (1)	MotorFailure	~StuckOpen
	JamClosed	~StuckOpen
Leaking (2)	Rupture	*(true)*

Output Failure Modes & Propagation Conditions

Failure Mode (ID)	FM Class	FM Group	Propagation Condition
Leak	Commission	HydIn.BWD	Leaking | Leak & ~StuckClosed & (StuckOpen | ~LackOfBraking)
PressureOmission	Omission	HydOut.FWD	StuckClosed | Leaking | PressureOmission | (LackOfBraking & ~StuckOpen)
PressureCommission	Commission	HydOut.FWD	(StuckOpen | InadvertentBraking) & ~StuckClosed & ~Leaking & ~PressureOmission
TooHighPressure	TooHightValue	HydOut.FWD	TooHighPressure & ~StuckClosed & ~StuckOpen &~Leaking & ~PressureOmission & ~TooLowPressure & ~InadvertentBraking & ~LackOfBraking & ~TooLittleBraking
TooLowPressure	LowValue	HydOut.FWD	(TooLittleBraking | TooLowPressure) & ~StuckClosed & ~StuckOpen & ~Leaking & ~PressureOmission & ~LackOfBraking & ~InadvertentBraking

B2. WBS Failure Logic Model: AltaRica Dataflow

This section is automatically generated by the "Translate Model" function of Cecilia OCAS with minor manual post-formatting. The model includes the specification of the failure behaviour of the components and the system (partially presented in Chapter Three). Visualisation information and other OCAS-specific annotations are removed.

```
domain ControlFMs = {OK, InadvertentBraking, LackOfAntiskid, LackOfBraking, TooLittleBraking};

domain PressureFMs = {OK, LowPressureCommis, PressureCommission, PressureOmission,
          TooHighPressure, TooLowPressure};
domain PowerFMs = {OK, PowerOmiss};
domain LeakFM = {OK, Leak};
domain AntiskidFMs = {OK, AntiSkidCommis, AntiSkidOmiss};
domain PumpPresFMs = {OK, Leak, LowPressureCommis, PressureCommission, PressureOmission,
         TooHighPressure, TooLowPressure};
domain ValidityFMs = {OK, FalseNeg, FalsePos };

node PedalStub
     // Virtual inputs component: (Altarica does not permit free inputs at system level):
     // Mimics command failure modes from the cockpit (transmitted to the WBS electronically or mechanically)
 flow
  Out:ControlFMs:out;
 state
  FailSt:ControlFMs;
 event
  INPUT_LackOfBraking,
  INPUT_InadvertentBraking,
  INPUT_TooLittleBraking;
 trans
  (FailSt = OK) |- INPUT_InadvertentBraking -> FailSt := InadvertentBraking;
  (FailSt = OK) |- INPUT_LackOfBraking -> FailSt := LackOfBraking;
  (FailSt = OK) |- INPUT_TooLittleBraking -> FailSt := TooLittleBraking;
 assert
  Out = FailSt;
 init
  FailSt := OK;
edon

node PowerStub
     // Virtual inputs component: (Altarica does not permit free inputs at system level):
     // Mimics power failure modes (that the BSCU is sensitive to)
 flow
  Out:PowerFMs:out;
 state
  FailSt:PowerFMs;
 event
  INPUT_PowerOmission;
 trans
  (FailSt = OK) |- INPUT_PowerOmission -> FailSt := PowerOmiss;
 assert
  Out = FailSt;
 init
  FailSt := OK;
edon

node LeakStub
```

```
    // Virtual inputs component: (Altarica does not permit free inputs at system level):
    // Mimics leak failure modes (exhibited by the brakes assembly)
 flow
  Out:LeakFM:out;
 state
  FailSt:LeakFM;
 event
  INPUT_Leak;
 trans
  (FailSt = OK) |- INPUT_Leak -> FailSt := Leak;
 assert
  Out = FailSt;
 init
  FailSt := OK;
edon

node WheelBrakes
    // Virtual observer component: calculates WBS Failure Conditions based on the
    // failure modes exhibited by green and blue hydraulic lines
 flow
  InBlue:PressureFMs:in;
  InGreen:PressureFMs:in;
  FailureCondition:{InadvertentBraking, InsufficientBraking, None, OmissionOfBraking, SuboptimalBraking}:out;
 assert
  FailureCondition = case {
   ((InGreen = PressureCommission) or (InBlue = PressureCommission))      : InadvertentBraking,
   ((InGreen = PressureOmission) and (InBlue = PressureOmission))         : OmissionOfBraking,
   (((InGreen = TooLowPressure) or (InGreen = PressureOmission))
    and ((InBlue = TooLowPressure) or (InBlue = PressureOmission)))       : InsufficientBraking,
   ((InGreen = TooHighPressure) or (InBlue = TooHighPressure))            : SuboptimalBraking,
   else None
  },
edon

node GreenMeterValve
 flow
  HydIn^FWD:PressureFMs:in;
  HydIn^BWD:LeakFM:out;
  Ctrl:ControlFMs:in;
  HydOut^FWD:PressureFMs:out;
  HydOut^BWD:LeakFM:in;
 state
  FailSt1:{OK, Struggling, StuckClosed, StuckOpen};
  FailSt2:{Leaking, OK};
 event
  Contamination,
  JamOpen,
  SpringFailure,
  MotorFailure,
  JamClosed,
  Rupture;
 trans
  (FailSt1 # StuckClosed) |- SpringFailure -> FailSt1 := StuckOpen;
  (FailSt1 # StuckClosed) |- JamOpen -> FailSt1 := StuckOpen;
  (FailSt1 # StuckOpen) |- MotorFailure -> FailSt1 := StuckClosed;
  (FailSt1 # StuckOpen) |- JamClosed -> FailSt1 := StuckClosed;
  ((FailSt1 # StuckClosed) and (FailSt1 # StuckOpen)) |- Contamination -> FailSt1 := Struggling;
  true |- Rupture -> FailSt2 := Leaking;
```

```
 assert
  HydIn^BWD = case {
   ((FailSt2 = Leaking) or
    (((HydOut^BWD = Leak) and (FailSt1 # StuckClosed)) and ((FailSt1 = StuckOpen) or
    (Ctrl # LackOfBraking)))) : Leak,
   else OK
  },
  HydOut^FWD = case {
   (((((FailSt1 = StuckClosed) or ((Ctrl = LackOfBraking) and (FailSt1 # StuckOpen)) ) or
     (HydIn^FWD = PressureOmission)) or (FailSt2 = Leaking)) or
    ((Ctrl # OK) and (not ((HydIn^FWD = LowPressureCommis) or
   (HydIn^FWD = PressureCommission))))) : PressureOmission,
   ((((((FailSt1 = StuckOpen) or
      (Ctrl = InadvertentBraking)) and (FailSt1 # StuckClosed)) and (FailSt2 # Leaking))
      and (HydIn^FWD # PressureOmission)) and (((HydIn^FWD = PressureCommission) or
          (HydIn^FWD = LowPressureCommis)) or (Ctrl = OK))) : PressureCommission,
   (((((((Ctrl = LackOfAntiskid) and (HydIn^FWD = PressureCommission)) and (FailSt1 # StuckOpen))
       and (FailSt1 # StuckClosed)) and (FailSt2 = OK)) or
    ((((FailSt1 = Struggling) and (FailSt2 = OK)) and (Ctrl = OK)) and (HydIn^FWD = OK))) : TooHighPressure,
   (((((((Ctrl = TooLittleBraking) and ((HydIn^FWD = PressureCommission) or
     (HydIn^FWD = LowPressureCommis))) and (FailSt1 # StuckClosed)) and (FailSt1 # StuckOpen))
     and (FailSt2 = OK))
    or (((((HydIn^FWD = LowPressureCommis) and (Ctrl = OK)) and (FailSt1 # StuckClosed))
       and (FailSt1 # StuckOpen)) and (FailSt2 = OK))) : TooLowPressure,
   else OK
  };
 init
  FailSt1 := OK,
  FailSt2 := OK;
edon

node MechPedalPosition
 flow
  Pedal:ControlFMs:in;
  CMD:ControlFMs:out;
 state
  FailSt:{LimitedAmplitude, OK, StuckDown, StuckUp};
 event
  Contamination,
  Jam,
  MechanicalFailure;
 trans
  true |- MechanicalFailure -> FailSt := StuckUp;
  (FailSt # StuckUp) |- Jam -> FailSt := StuckDown;
  (FailSt = OK) |- Contamination -> FailSt := LimitedAmplitude;
 assert
  CMD = case {
   ((FailSt = StuckDown) or ((Pedal = InadvertentBraking) and (FailSt # StuckUp))) : InadvertentBraking,
   ((FailSt = StuckUp) or ((Pedal = LackOfBraking) and (FailSt # StuckDown))) : LackOfBraking,
   (((( (Pedal = TooLittleBraking) or (FailSt = LimitedAmplitude))
       and (Pedal # LackOfBraking)) and (Pedal # InadvertentBraking)) and (FailSt # StuckUp))
       and (FailSt # StuckDown)) : TooLittleBraking,
   else OK
  };
 init
  FailSt := OK;
edon
```

```
node BlueMeterValve
 flow
  HydIn^FWD:PressureFMs:in;
  HydIn^BWD:LeakFM:out;
  CTRL:ControlFMs:in;
  HydOut^FWD:PressureFMs:out;
  HydOut^BWD:LeakFM:in;
 state
  FailSt1:{OK, StuckClosed, StuckOpen};
  FailSt2:{Leaking, OK};
 event
  SpringFailure,
  JamClosed,
  JamOpen,
  MotorFailure,
  Rupture;
 trans
  (FailSt1 # StuckClosed) |- JamOpen -> FailSt1 := StuckOpen;
  (FailSt1 # StuckClosed) |- SpringFailure -> FailSt1 := StuckOpen;
  (FailSt1 # StuckOpen) |- JamClosed -> FailSt1 := StuckClosed;
  (FailSt1 # StuckOpen) |- MotorFailure -> FailSt1 := StuckClosed;
  true |- Rupture -> FailSt2 := Leaking;
 assert
  HydIn^BWD = case {
   ((FailSt2 = Leaking) or
   (((HydOut^BWD = Leak) and (FailSt1 # StuckClosed))
    and ((CTRL # LackOfBraking) or (FailSt1 = StuckOpen)))) : Leak,
   else OK
  },
  HydOut^FWD = case {
   ((((FailSt1 = StuckClosed) or (FailSt2 = Leaking)) or (HydIn^FWD = PressureOmission)) or
    ((CTRL = LackOfBraking) and (FailSt1 # StuckOpen)))                    : PressureOmission,
   (((((CTRL = InadvertentBraking) or (FailSt1 = StuckOpen))
      and (HydIn^FWD # PressureOmission)) and (FailSt1 # StuckClosed))
     and (FailSt2 = OK))                                                   : PressureCommission,
   ((((HydIn^FWD = TooHighPressure) and (FailSt1 = OK))
     and (FailSt2 = OK)) and (CTRL = OK))                                  : TooHighPressure,
   (((((((HydIn^FWD = TooLowPressure) or (CTRL = TooLittleBraking))
        and (HydIn^FWD # PressureOmission)) and (CTRL # InadvertentBraking))
       and (CTRL # LackOfBraking)) and (FailSt2 = OK)) and (FailSt1 = OK))  : TooLowPressure,
   else OK
  };
 init
  FailSt1 := OK,
  FailSt2 := OK;
edon

node ASShutOffValve
 flow
  HydIn^FWD:PressureFMs:in;
  HydIn^BWD:LeakFM:out;
  CTRL:AntiskidFMs:in;
  HydOut^FWD:PressureFMs:out;
  HydOut^BWD:LeakFM:in;
 state
  FailSt1:{OK, Struggling, StuckClosed, StuckOpen};
  FailSt2:{Leaking, OK};
 event
  LoadFailure,
  JamOpen,
  MotorFailure,
  Contamination,
  Rupture;
```

```
 trans
  (FailSt1 = OK) |- Contamination -> FailSt1 := Struggling;
  (FailSt1 # StuckClosed) |- MotorFailure -> FailSt1 := StuckOpen;
  (FailSt1 # StuckClosed) |- JamOpen -> FailSt1 := StuckOpen;
  (FailSt1 # StuckOpen) |- LoadFailure -> FailSt1 := StuckClosed;
  true |- Rupture -> FailSt2 := Leaking;
 assert
  HydIn^BWD = case {
   ((FailSt2 = Leaking) or ((HydOut^BWD = Leak) and (FailSt1 # StuckClosed))) : Leak,
   else OK
  },
  HydOut^FWD = case {
   (((FailSt1 = StuckClosed) or (FailSt2 = Leaking)) or (HydIn^FWD = PressureOmission)) : PressureOmission,
   ((((((CTRL = AntiSkidOmiss) or (FailSt1 = StuckOpen)) or (FailSt1 = Struggling))
      and (FailSt2 = OK)) and (FailSt1 # StuckClosed)) and (HydIn^FWD = OK))          : TooHighPressure,
   (((((HydIn^FWD = TooLowPressure) or
      ((CTRL = AntiSkidCommis) and (FailSt1 # StuckOpen))) and (HydIn^FWD # PressureOmission))
     and (FailSt2 = OK)) and (FailSt1 # StuckClosed))                                : TooLowPressure,
   else OK
  };
 init
  FailSt1 := OK,
  FailSt2 := OK;
edon

node ShutOffSelectorValve
 flow
  HydIn^FWD:PumpPresFMs:in;
  HydIn^BWD:LeakFM:out;
  CTRL:ValidityFMs:in;
  HydOut^FWD:PressureFMs:out;
  HydOut^BWD:LeakFM:in;
 state
  FailSt1:{OK, StuckClosed, StuckOpen};
  FailSt2:{Leaking, OK};
 event
  SpringFailure,
  UnlocksSpontaneously,
  Rupture;
 trans
  (FailSt1 = OK) |- SpringFailure -> FailSt1 := StuckOpen;
  (FailSt1 = OK) |- UnlocksSpontaneously -> FailSt1 := StuckClosed;
  true |- Rupture -> FailSt2 := Leaking;
 assert
  HydIn^BWD = case {
   ((FailSt2 = Leaking) or
    (((HydOut^BWD = Leak) and (FailSt1 # StuckClosed)) and
     ((CTRL # FalsePos) or (FailSt1 = StuckOpen))))                                : Leak,
   else OK
  },
  HydOut^FWD = case {
   (((((FailSt1 = StuckClosed) or
      (FailSt2 = Leaking)) or (HydIn^FWD = PressureOmission)) or (HydIn^FWD = Leak)) or
    ((CTRL = FalsePos) and (FailSt1 # StuckOpen)))                                : PressureOmission,
   ((((((FailSt1 = StuckOpen) or (CTRL = FalseNeg))
      and (FailSt1 # StuckClosed)) and (FailSt2 # Leaking)) and (HydIn^FWD = OK))  : PressureCommission,
   ((((((FailSt1 = StuckOpen) or (CTRL = FalseNeg))
      and (FailSt1 # StuckClosed)) and (FailSt2 # Leaking))
     and (HydIn^FWD = TooLowPressure))                                            : LowPressureCommis,
   ((((HydIn^FWD = TooLowPressure) and (FailSt1 = OK))
     and (FailSt2 = OK)) and (CTRL = OK))                                         : TooLowPressure,
   else OK
  };
```

```
 init
  FailSt1 := OK,
  FailSt2 := OK;
edon

node Accumulator
 flow
  In:PumpPresFMs:in;
  Out:PumpPresFMs:out;
 state
  FailSt:{Isolated, Leaking, OK, PartialLoss, TotalLoss};
  NormSt:{Empty, Full};
 event
  ConnectorBlocked,
  PartialDecompression,
  CompleteDecompression,
  Rupture,
  NORMAL_UsedOnce,
  Void;
 trans
  true |- ConnectorBlocked -> FailSt := Isolated;
  ((FailSt = OK) and (NormSt = Full)) |- PartialDecompression -> FailSt := PartialLoss;
  ((FailSt # Isolated) and (FailSt # Leaking)) |- CompleteDecompression -> FailSt := TotalLoss;
  ((((FailSt # Leaking) and (FailSt # Isolated)) and (FailSt # TotalLoss)) and (In = Leak))
            |- Void -> FailSt := TotalLoss;
  (FailSt # Isolated) |- Rupture -> FailSt := Leaking;
  (((NormSt = Full) and (In = PressureOmission)) and ((FailSt = OK) or (FailSt = PartialLoss)))
            |- NORMAL_UsedOnce -> NormSt := Empty;
 assert
  Out = case {
   (((FailSt = Isolated) or (FailSt = TotalLoss)) or (NormSt = Empty))      : PressureOmission,
   ((FailSt = PartialLoss) and (NormSt = Full))                             : TooLowPressure,
   (FailSt = Leaking)                                                       : Leak,
   else OK
  };
 init
  FailSt := OK,
  NormSt := Full;
 extern
  law <event NORMAL_UsedOnce> = Dirac(1);
  law <event Void> = Dirac(0);
edon

node Junction
 flow
  Pump^FWD:PumpPresFMs:in;
  Pump^BWD:LeakFM:out;
  AccumIn:PumpPresFMs:in;
  AccumOut:PumpPresFMs:out;
  Out^FWD:PressureFMs:out;
  Out^BWD:LeakFM:in;
 assert
  Pump^BWD = case {
   ((AccumIn = Leak) or (Out^BWD = Leak)) : Leak,
   else OK
  },
  AccumOut = case {
   ((Pump^FWD = Leak) or (Out^BWD = Leak))                  : Leak,
   ((Pump^FWD = PressureOmission) and (Out^BWD # Leak))     : PressureOmission,
   else OK
  },
```

```
  Out^FWD = case {
   (((AccumIn = Leak) or (Pump^FWD = Leak)) or ((Pump^FWD = PressureOmission)
    and (AccumIn = PressureOmission)))                                    : PressureOmission,
   (((Pump^FWD = TooLowPressure) and (AccumIn # Leak)) or
    ((Pump^FWD = PressureOmission) and (AccumIn = TooLowPressure)))       : TooLowPressure,
   else OK
  };
edon

node IsolationValve
 flow
  In^FWD:PumpPresFMs:in;
  In^BWD:LeakFM:out;
  Out^FWD:PumpPresFMs:out;
  Out^BWD:LeakFM:in;
 state
  FailSt1:{OK, StuckClosed, StuckOpen};
  FailSt2:{Leaking, OK};
 event
  Contamination,
  Jam,
  Rupture;
 trans
  true |- Contamination -> FailSt1 := StuckClosed;
  ((FailSt1 # StuckClosed) and ((In^FWD = PressureOmission) or (In^FWD = Leak)))
            |- Jam -> FailSt1 := StuckOpen;
  true |- Rupture -> FailSt2 := Leaking;
 assert
  In^BWD = case {
   ((FailSt2 = Leaking) or ((Out^BWD = Leak) and (FailSt1 # StuckClosed)))  : Leak,
   else OK
  },
  Out^FWD = case {
   ((FailSt2 = Leaking) or ((In^FWD = Leak) and (FailSt1 = StuckOpen)))     : Leak,
   ((FailSt2 # Leaking) and
    (((FailSt1 = StuckClosed) or (In^FWD = PressureOmission)) or
     ((In^FWD = Leak) and (FailSt1 # StuckOpen))))                         : PressureOmission,
   (((In^FWD = TooLowPressure) and (FailSt1 # StuckClosed)) and (FailSt2 # Leaking)) : TooLowPressure,
   else OK
  };
 init
  FailSt1 := OK,
  FailSt2 := OK;
edon

node Pump
 flow
  Out^FWD:PumpPresFMs:out;
  Out^BWD:LeakFM:in;
 state
  FailSt:{Leaking, OK, Struggling, TotalLoss};
 event
  MechanicalFailure,
  Contamination,
  Rupture,
  Void;
 trans
  true |- Rupture -> FailSt := Leaking;
  (FailSt = OK) |- Contamination -> FailSt := Struggling;
  (FailSt # Leaking) |- MechanicalFailure -> FailSt := TotalLoss;
  (((FailSt # Leaking) and (Out^BWD = Leak)) and (FailSt # TotalLoss)) |- Void -> FailSt := TotalLoss;
```

```
  assert
    Out^FWD = case {
      (FailSt = Leaking)          : Leak,
      (FailSt = TotalLoss)        : PressureOmission,
      (FailSt = Struggling)       : TooLowPressure,
      else OK
    };
  init
    FailSt := OK;
  extern
    law <event Void> = Dirac(0);
edon

node Switch
  flow
    AS1:AntiskidFMs:in;
    AS2:AntiskidFMs:in;
    CMD1:ControlFMs:in;
    CMD2:ControlFMs:in;
    AS:AntiskidFMs:out;
    CMD:ControlFMs:out;
    Validity:ValidityFMs:in;
  assert
    AS = case {
      ((Validity = FalseNeg) or ((CMD1 = OK) and (Validity # FalsePos))) : AS1,
      else AS2
    },
    CMD = case {
      ((Validity = FalseNeg) or ((CMD1 = OK) and (Validity # FalsePos))) : CMD1,
      else CMD2
    };
edon

node ValidityMonitor
  flow
    In1:ValidityFMs:in;
    In2:ValidityFMs:in;
    PWR1:PowerFMs:in;
    PWR2:PowerFMs:in;
    Validity:ValidityFMs:out;
  state
    FailSt:{OK, StuckNeg, StuckPos};
  event
    ProcessStuck,
    ProcessTerminated;
  trans
    ((PWR1 = OK) and (PWR2 = OK)) |- ProcessTerminated -> FailSt := StuckPos;
    (((FailSt = OK) and (PWR1 = OK)) and (PWR2 = OK)) |- ProcessStuck -> FailSt := StuckNeg;
  assert
    Validity = case {
      (((((In1 = FalseNeg) or (In2 = FalseNeg)) or (FailSt = StuckNeg))
        and (FailSt # StuckPos)) and ((PWR1 = OK) or (PWR2 = OK)))          : FalseNeg,
      (((((In1 = FalsePos) or (In2 = FalsePos)) or (FailSt = StuckPos))
        and (FailSt # StuckNeg)) and ((PWR1 = OK) or (PWR2 = OK)))          : FalsePos,
      else OK
    };
  init
    FailSt := OK;
edon
```

```
node MonitorModule
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  CMD:ControlFMs:in;
  PWR:PowerFMs:in;
  Validity:ValidityFMs:out;
 state
  FailSt:{OK, StuckNeg, StuckPos};
 event
  ProcessStuck,
  ProcessTerminated,
  ProcessorError;
 trans
  (PWR = OK) |- ProcessTerminated -> FailSt := StuckPos;
  ((FailSt = OK) and (PWR = OK)) |- ProcessorError -> FailSt := StuckPos;
  ((FailSt = OK) and (PWR = OK)) |- ProcessStuck -> FailSt := StuckNeg;
 assert
  Validity = case {
   ((((((CMD = Pedal1) and (CMD = Pedal2)) and (CMD # OK)) or
      (FailSt = StuckNeg)) and (FailSt # StuckPos)) and (PWR = OK))          : FalseNeg,
   (((((FailSt = StuckPos) and (Pedal1 = OK)) and (Pedal2 = OK))
     and (CMD = OK)) and (PWR = OK))                                         : FalsePos,
    else OK
  };
 init
  FailSt := OK;
edon

node CommandModule
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  PWR:PowerFMs:in;
  AS:AntiskidFMs:out;
  CMD:ControlFMs:out;
 state
  FailSt2:{ASstuckOff, ASstuckOn, OK};
  FailSt1:{CMDstuckOff, CMDstuckOn, CalculationError, OK};
 event
  ASprocessStuck,
  ASprocessTerminated,
  ProcessorError,
  CMDprocessStuck,
  CMDprocessTerminated;
 trans
  ((PWR # PowerOmiss) and (FailSt1 = OK)) |- ProcessorError -> FailSt1 := CalculationError;
  ((PWR # PowerOmiss) and (FailSt1 # CMDstuckOff)) |- CMDprocessStuck -> FailSt1 := CMDstuckOn;
  (PWR # PowerOmiss) |- CMDprocessTerminated -> FailSt1 := CMDstuckOff;
  ((PWR # PowerOmiss) and (FailSt2 # ASstuckOff)) |- ASprocessStuck -> FailSt2 := ASstuckOn;
  (PWR # PowerOmiss) |- ASprocessTerminated -> FailSt2 := ASstuckOff;
 assert
  AS = case {
   ((FailSt2 = ASstuckOff) or (PWR = PowerOmiss))                  : AntiSkidOmiss,
   ((FailSt2 = ASstuckOn) and (PWR = OK))                          : AntiSkidCommis,
   else OK
  },
  CMD = case {
   (((((Pedal1 = InadvertentBraking) or (Pedal2 = InadvertentBraking)) or (FailSt1 = CMDstuckOn))
     and (FailSt1 # CMDstuckOff)) and (PWR = OK))                                  : InadvertentBraking,
   ((PWR = PowerOmiss) or
    ((((Pedal1 = LackOfBraking) and (Pedal2 = LackOfBraking)) or
      (FailSt1 = CMDstuckOff)) and (FailSt1 # CMDstuckOn)))                        : LackOfBraking,
```

```
    ((((((((((FailSt2 = ASstuckOn) or (FailSt1 = CalculationError)) or (Pedal1 = LackOfBraking)) or
          (Pedal2 = LackOfBraking)) or (Pedal1 = TooLittleBraking)) or
        (Pedal2 = TooLittleBraking)) and (FailSt1 = OK)) and (PWR = OK))
     and (Pedal1 # InadvertentBraking)) and (Pedal2 # InadvertentBraking)) and (not ((Pedal1 = LackOfBraking)
     and (Pedal2 = LackOfBraking))))                                          : TooLittleBraking,
    (((((FailSt2 = ASstuckOff) and (PWR = OK)) and (FailSt1 = OK)) and (Pedal1 = OK))
     and (Pedal2 = OK))                                                       : LackOfAntiskid,
    else OK
  };
 init
  FailSt2 := OK,
  FailSt1 := OK;
edon

node BSCUchannel
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  AS:AntiskidFMs:out;
  PWR:PowerFMs:in;
  CMD:ControlFMs:out;
  Validity:ValidityFMs:out;
 sub
  MON:MonitorModule;
  COM:CommandModule;
 assert
  AS = COM.AS,
  CMD = COM.CMD,
  Validity = MON.Validity,
  MON.Pedal1 = Pedal1,
  MON.Pedal2 = Pedal2,
  MON.CMD = COM.CMD,
  MON.PWR = PWR,
  COM.Pedal1 = Pedal1,
  COM.Pedal2 = Pedal2,
  COM.PWR = PWR;
edon

node BSCU
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  PWR1:PowerFMs:in;
  PWR2:PowerFMs:in;
  AS:AntiskidFMs:out;
  CMD:ControlFMs:out;
  Validity:ValidityFMs:out;
 sub
  Switch:Switch;
  ValidityMonitor:ValidityMonitor;
  BSCU2:BSCUchannel;
  BSCU1:BSCUchannel;
 assert
  AS = Switch.AS,
  CMD = Switch.CMD,
  Validity = ValidityMonitor.Validity,
  Switch.AS1 = BSCU1.AS,
  Switch.AS2 = BSCU2.AS,
  Switch.CMD1 = BSCU1.CMD,
  Switch.CMD2 = BSCU2.CMD,
  Switch.Validity = BSCU1.Validity,
  ValidityMonitor.In1 = BSCU1.Validity,
  ValidityMonitor.In2 = BSCU2.Validity,
  ValidityMonitor.PWR1 = PWR1,
```

```
    ValidityMonitor.PWR2 = PWR2,
    BSCU2.Pedal1 = Pedal1,
    BSCU2.Pedal2 = Pedal2,
    BSCU2.PWR = PWR2,
    BSCU1.Pedal1 = Pedal1,
    BSCU1.Pedal2 = Pedal2,
    BSCU1.PWR = PWR1;
edon

node main
 sub
    Pedal1:PedalStub;
    Pedal2:PedalStub;
    PedalM:PedalStub;
    WheelBrakes:WheelBrakes;
    BusbarB:PowerStub;
    BusbarA:PowerStub;
    BlueBrakesAssembly:LeakStub;
    GreenBrakesAssembly:LeakStub;
    GreenMeterValve:GreenMeterValve;
    MechPedalPosition:MechPedalPosition;
    BlueMeterValve:BlueMeterValve;
    ASShutOffValve:ASShutOffValve;
    ShutOffSelectorValve:ShutOffSelectorValve;
    Accumulator:Accumulator;
    Junction:Junction;
    IsolationValve:IsolationValve;
    BluePump:Pump;
    GreenPump:Pump;
    BSCU:BSCU;
 assert
    WheelBrakes.InBlue = BlueMeterValve.HydOut^FWD,
    WheelBrakes.InGreen = GreenMeterValve.HydOut^FWD,
    GreenMeterValve.HydIn^FWD = ShutOffSelectorValve.HydOut^FWD,
    GreenMeterValve.Ctrl = BSCU.CMD,
    GreenMeterValve.HydOut^BWD = GreenBrakesAssembly.Out,
    MechPedalPosition.Pedal = PedalM.Out,
    BlueMeterValve.HydIn^FWD = ASShutOffValve.HydOut^FWD,
    BlueMeterValve.CTRL = MechPedalPosition.CMD,
    BlueMeterValve.HydOut^BWD = BlueBrakesAssembly.Out,
    ASShutOffValve.HydIn^FWD = Junction.Out^FWD,
    ASShutOffValve.CTRL = BSCU.AS,
    ASShutOffValve.HydOut^BWD = BlueMeterValve.HydIn^BWD,
    ShutOffSelectorValve.HydIn^FWD = GreenPump.Out^FWD,
    ShutOffSelectorValve.CTRL = BSCU.Validity,
    ShutOffSelectorValve.HydOut^BWD = GreenMeterValve.HydIn^BWD,
    Accumulator.In = Junction.AccumOut,
    Junction.Pump^FWD = IsolationValve.Out^FWD,
    Junction.AccumIn = Accumulator.Out,
    Junction.Out^BWD = ASShutOffValve.HydIn^BWD,
    IsolationValve.In^FWD = BluePump.Out^FWD,
    IsolationValve.Out^BWD = Junction.Pump^BWD,
    BluePump.Out^BWD = IsolationValve.In^BWD,
    GreenPump.Out^BWD = ShutOffSelectorValve.HydIn^BWD,
    BSCU.Pedal1 = Pedal1.Out,
    BSCU.Pedal2 = Pedal2.Out,
    BSCU.PWR1 = BusbarA.Out,
    BSCU.PWR2 = BusbarB.Out;
edon
```

B3. Revised BSCU Model: AltaRica Dataflow

This section presents a revision of the failure logic model of the BSCU as discussed in Chapter Five of the Thesis. Figure 106 shows the top-level BSCU model architecture which now contains two virtual "mode observer" components and a "translator" component. The remainder of this section provides the AltaRica Dataflow listing as generated by the Cecilia OCAS "Translate Model" function (with minor post-formatting)

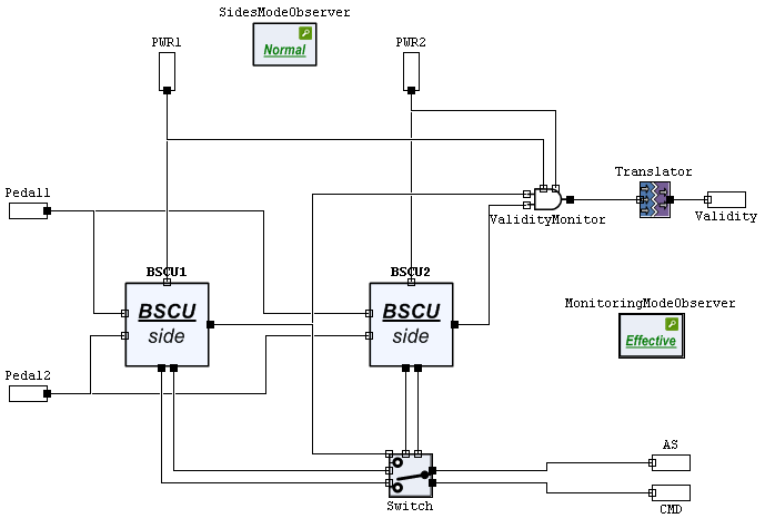


Figure 106 - Revised Architecture of the BSCU Model in AltaRica OCAS

```
node BSCUTranslator
 flow
  In:ValidityFMs:in;
  Out:ValidityFMs:out;
 state
  Mode:{Effective, Ineffective};
 event
  BroadcastIneffective;
 trans
  true |- BroadcastIneffective -> Mode := Ineffective;
 assert
  Out = case {
   ((Mode = Ineffective) and (In = OK)) : FalseNeg,
   else In
  };
 init
  Mode := Effective;
edon

node MonitoringModeObserver
 state
  Mode:{Effective, Ineffective};
  BroadcastPending:bool;
 event
  TriggerSwitchStuck,
  TriggerSwitchTerminated,
  BroadcastIneffective;
```

```
 trans
  (Mode = Effective) |- TriggerSwitchStuck -> Mode := Ineffective, BroadcastPending := true;
  (Mode = Effective) |- TriggerSwitchTerminated -> Mode := Ineffective, BroadcastPending := true;
  (BroadcastPending and (Mode = Ineffective)) |- BroadcastIneffective -> BroadcastPending := false;
 init
  Mode := Effective,
  BroadcastPending := false;
 extern
  law <event BroadcastIneffective> = Dirac(0);
edon

node ModeObserver
 state
  Mode:{Alternate1, Alternate2, Normal, Untrustworthy};
  BroadcastPending:bool;
 event
  COM1failed,
  COM2failed,
  BroadcastAlternate1,
  BroadcastAlternate2,
  BroadcastUntrustworthy;
 trans
  // Reading mode events:
  (Mode = Normal) |- COM1failed -> Mode := Alternate2, BroadcastPending := true;
  (Mode = Normal) |- COM2failed -> Mode := Alternate1, BroadcastPending := true;
  (Mode = Alternate1) |- COM1failed -> Mode := Untrustworthy, BroadcastPending := true;
  (Mode = Alternate2) |- COM2failed -> Mode := Untrustworthy, BroadcastPending := true;
  (Mode = Alternate2) |- COM1failed -> Mode := Alternate2;
  (Mode = Alternate1) |- COM2failed -> Mode := Alternate1;
  (Mode = Untrustworthy) |- COM1failed -> Mode := Untrustworthy;
  (Mode = Untrustworthy) |- COM2failed -> Mode := Untrustworthy;
  // Broadcasting modes:
  ((Mode = Alternate1) and BroadcastPending) |- BroadcastAlternate1 -> BroadcastPending := false;
  ((Mode = Alternate2) and BroadcastPending) |- BroadcastAlternate2 -> BroadcastPending := false;
  ((Mode = Untrustworthy) and BroadcastPending) |- BroadcastUntrustworthy -> BroadcastPending := false;
 init
  Mode := Normal,
  BroadcastPending := false;
 extern
  law <event BroadcastAlternate1> = Dirac(0);
  law <event BroadcastAlternate2> = Dirac(0);
  law <event BroadcastUntrustworthy> = Dirac(0);
edon

node Switch
 flow
  AS1:AntiskidFMs:in;
  AS2:AntiskidFMs:in;
  CMD1:ControlFMs:in;
  CMD2:ControlFMs:in;
  AS:AntiskidFMs:out;
  CMD:ControlFMs:out;
  Validity:ValidityFMs:in;
 state
  ModePending:bool;
  FailSt:{Lost, OK, Stuck1};
 event
  ProcessTerminated,
  ProcessStuck,
  TriggerTerminated,
  TriggerStuck;
```

```
 trans
  (FailSt = OK) |- ProcessStuck -> FailSt := Stuck1, ModePending := true;
  (FailSt = OK) |- ProcessTerminated -> FailSt := Lost, ModePending := true;
  (ModePending and (FailSt = Stuck1)) |- TriggerStuck -> ModePending := false;
  (ModePending and (FailSt = Lost)) |- TriggerTerminated -> ModePending := false;
 assert
  AS = case {
   (FailSt = Lost) : AntiSkidOmiss,
   (((Validity = FalseNeg) or (FailSt = Stuck1)) or ((CMD1 = OK) and (Validity # FalsePos))) : AS1,
   else AS2
  },
  CMD = case {
   (FailSt = Lost) : LackOfBraking,
   (((Validity = FalseNeg) or (FailSt = Stuck1)) or ((CMD1 = OK) and (Validity # FalsePos))) : CMD1,
   else CMD2
  };
 init
  ModePending := false,
  FailSt := OK;
 extern
  law <event TriggerTerminated> = Dirac(0);
  law <event TriggerStuck> = Dirac(0);
edon

node ValidityMonitor
 flow
  In1:ValidityFMs:in;
  In2:ValidityFMs:in;
  PWR1:PowerFMs:in;
  PWR2:PowerFMs:in;
  Validity:ValidityFMs:out;
 state
  FailSt:{OK, StuckNeg, StuckPos};
  Mode:{Alternate1, Alternate2, Normal, Untrustworthy};
 event
  ProcessStuck,
  ProcessTerminated,
  BroadcastAlternate1,
  BroadcastAlternate2,
  BroadcastUntrustworthy;
 trans
  ((PWR1 = OK) and (PWR2 = OK)) |- ProcessTerminated -> FailSt := StuckPos;
  (((FailSt = OK) and (PWR1 = OK)) and (PWR2 = OK)) |- ProcessStuck -> FailSt := StuckNeg;
  // 'Copying' modes into a local state variable:
  true |- BroadcastAlternate1 -> Mode := Alternate1;
  true |- BroadcastAlternate2 -> Mode := Alternate2;
  true |- BroadcastUntrustworthy -> Mode := Untrustworthy;
 assert
  Validity = case {
   (((((In1 = FalseNeg) or (In2 = FalseNeg)) or (FailSt = StuckNeg))
     and (FailSt # StuckPos)) and ((PWR1 = OK) or (PWR2 = OK)))                    : FalseNeg,
   (((((In1 = FalsePos) or (In2 = FalsePos)) or (FailSt = StuckPos)) and (FailSt # StuckNeg))
    and ((PWR1 = OK) or (PWR2 = OK)))                                             : FalsePos,
   else OK
  };
 init
  FailSt := OK,
  Mode := Normal;
edon
```

```
node MonitorModule
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  CMD:ControlFMs:in;
  PWR:PowerFMs:in;
  Validity:ValidityFMs:out;
 state
  FailSt:{OK, StuckNeg, StuckPos};
 event
  ProcessStuck,
  ProcessTerminated,
  ProcessorError;
 trans
  (PWR = OK) |- ProcessTerminated -> FailSt := StuckPos;
  ((FailSt = OK) and (PWR = OK)) |- ProcessorError -> FailSt := StuckPos;
  ((FailSt = OK) and (PWR = OK)) |- ProcessStuck -> FailSt := StuckNeg;
 assert
  Validity = case {
   ((((((CMD = Pedal1) and (CMD = Pedal2)) and (CMD # OK)) or (FailSt = StuckNeg))
     and (FailSt # StuckPos)) and (PWR = OK))                          : FalseNeg,
   (((((FailSt = StuckPos) and (Pedal1 = OK)) and (Pedal2 = OK))
     and (CMD = OK)) and (PWR = OK))                                   : FalsePos,
   else OK
  };
 init
  FailSt := OK;
edon

node CommandModule
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  PWR:PowerFMs:in;
  AS:AntiskidFMs:out;
  CMD:ControlFMs:out;
 state
  FailSt2:{ASstuckOff, ASstuckOn, OK};
  FailSt1:{CMDstuckOff, CMDstuckOn, CalculationError, OK};
  ModePending:bool;
 event
  ASprocessStuck,
  ASprocessTerminated,
  ProcessorError,
  CMDprocessStuck,
  CMDprocessTerminated,
  TriggerFailed;
 trans
  ((PWR # PowerOmiss) and (FailSt1 = OK)) |- ProcessorError
          -> FailSt1 := CalculationError, ModePending := true;
  ((PWR # PowerOmiss) and (FailSt1 # CMDstuckOff)) |- CMDprocessStuck
          -> FailSt1 := CMDstuckOn, ModePending := true;
  (PWR # PowerOmiss) |- CMDprocessTerminated -> FailSt1 := CMDstuckOff, ModePending := true;
  ((PWR # PowerOmiss) and (FailSt2 # ASstuckOff)) |- ASprocessStuck
          -> FailSt2 := ASstuckOn, ModePending := true;
  (PWR # PowerOmiss) |- ASprocessTerminated -> FailSt2 := ASstuckOff, ModePending := true;
  // Broadcasting mode event::
  (ModePending and ((FailSt1 # OK) or (FailSt2 # OK))) |- TriggerFailed -> ModePending := false;
 assert
  AS = case {
   ((FailSt2 = ASstuckOff) or (PWR = PowerOmiss))                      : AntiSkidOmiss,
   ((FailSt2 = ASstuckOn) and (PWR = OK))                              : AntiSkidCommis,
   else OK
  },
```

```
   CMD = case {
    (((((Pedal1 = InadvertentBraking) or (Pedal2 = InadvertentBraking)) or (FailSt1 = CMDstuckOn))
      and (FailSt1 # CMDstuckOff)) and (PWR = OK))                                  : InadvertentBraking,
    ((PWR = PowerOmiss) or
     ((((Pedal1 = LackOfBraking) and (Pedal2 = LackOfBraking)) or
       (FailSt1 = CMDstuckOff)) and (FailSt1 # CMDstuckOn)))                        : LackOfBraking,
    ((((((((((FailSt2 = ASstuckOn) or (FailSt1 = CalculationError)) or (Pedal1 = LackOfBraking)) or
           (Pedal2 = LackOfBraking)) or (Pedal1 = TooLittleBraking)) or
         (Pedal2 = TooLittleBraking)) and (FailSt1 = OK)) and (PWR = OK))
      and (Pedal1 # InadvertentBraking)) and (Pedal2 # InadvertentBraking)) and (not ((Pedal1 = LackOfBraking)
      and (Pedal2 = LackOfBraking))))                                               : TooLittleBraking,
    (((((FailSt2 = ASstuckOff) and (PWR = OK)) and (FailSt1 = OK)) and (Pedal1 = OK))
     and (Pedal2 = OK))                                                             : LackOfAntiskid,
    else OK
   };
  init
   FailSt2 := OK,
   FailSt1 := OK,
   ModePending := false;
  extern
   law <event TriggerFailed> = Dirac(0);
edon

node BSCUchannel
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  AS:AntiskidFMs:out;
  PWR:PowerFMs:in;
  CMD:ControlFMs:out;
  Validity:ValidityFMs:out;
 sub
  MON:MonitorModule;
  COM:CommandModule;
 assert
  AS = COM.AS,
  CMD = COM.CMD,
  Validity = MON.Validity,
  MON.Pedal1 = Pedal1,
  MON.Pedal2 = Pedal2,
  MON.CMD = COM.CMD,
  MON.PWR = PWR,
  COM.Pedal1 = Pedal1,
  COM.Pedal2 = Pedal2,
  COM.PWR = PWR;
edon

node BSCU
 flow
  Pedal1:ControlFMs:in;
  Pedal2:ControlFMs:in;
  PWR1:PowerFMs:in;
  PWR2:PowerFMs:in;
  AS:AntiskidFMs:out;
  CMD:ControlFMs:out;
  Validity:ValidityFMs:out;
```

```
event
  TriggerCOM1fail,
  TriggerCOM2fail,
  BroadcastAlternate1,
  BroadcastAlternate2,
  BroadcastUntrustworthy,
  TriggerSwitchStuck,
  TriggerSwitchTerminated,
  BroadcastIneffective;
sub
  Translator:BSCUTranslator;
  MonitoringModeObserver:MonitoringModeObserver;
  Switch:Switch;
  ValidityMonitor:ValidityMonitor;
  BSCU2:BSCUchannel;
  BSCU1:BSCUchannel;
  SidesModeObserver:ModeObserver;
assert
  AS = Switch.AS,
  CMD = Switch.CMD,
  Validity = Translator.Out,
  Translator.In = ValidityMonitor.Validity,
  Switch.AS1 = BSCU1.AS,
  Switch.AS2 = BSCU2.AS,
  Switch.CMD1 = BSCU1.CMD,
  Switch.CMD2 = BSCU2.CMD,
  Switch.Validity = BSCU1.Validity,
  ValidityMonitor.In1 = BSCU1.Validity,
  ValidityMonitor.In2 = BSCU2.Validity,
  ValidityMonitor.PWR1 = PWR1,
  ValidityMonitor.PWR2 = PWR2,
  BSCU2.Pedal1 = Pedal1,
  BSCU2.Pedal2 = Pedal2,
  BSCU2.PWR = PWR2,
  BSCU1.Pedal1 = Pedal1,
  BSCU1.Pedal2 = Pedal2,
  BSCU1.PWR = PWR1;
sync
  <TriggerCOM1fail , BSCU1.COM.TriggerFailed , SidesModeObserver.COM1failed>,
  <TriggerCOM2fail , BSCU2.COM.TriggerFailed , SidesModeObserver.COM2failed>,
  <BroadcastAlternate1 , SidesModeObserver.BroadcastAlternate1 , ValidityMonitor.BroadcastAlternate1>,
  <BroadcastAlternate2 , SidesModeObserver.BroadcastAlternate2 , ValidityMonitor.BroadcastAlternate2>,
  <BroadcastUntrustworthy , SidesModeObserver.BroadcastUntrustworthy ,
          ValidityMonitor.BroadcastUntrustworthy>,
  <TriggerSwitchStuck , MonitoringModeObserver.TriggerSwitchStuck , Switch.TriggerStuck>,
  <TriggerSwitchTerminated , MonitoringModeObserver.TriggerSwitchTerminated , Switch.TriggerTerminated>,
  <BroadcastIneffective , MonitoringModeObserver.BroadcastIneffective , Translator.BroadcastIneffective>;
extern
  law <event TriggerCOM1fail> = Dirac(0);
  law <event TriggerCOM2fail> = Dirac(0);
  law <event BroadcastAlternate1> = Dirac(0);
  law <event BroadcastAlternate2> = Dirac(0);
  law <event BroadcastUntrustworthy> = Dirac(0);
  law <event TriggerSwitchStuck> = Dirac(0);
  law <event TriggerSwitchTerminated> = Dirac(0);
  law <event BroadcastIneffective> = Dirac(0);
edon
```

Appendix C:

Computation and Communications Platform DSFM

This appendix supplements Section 4.6 of Chapter Four by presenting case study material for integration of the DSFMs of the Wheel Braking System and a simplified hypothetical Common Computing and Communications Infrastructure (for simplicity referred to as the IMA). The Appendix is organised as following:

Section C1 presents modifications to the failure logic model of the Braking System Control Unit (BSCU) of the WBS

Section C2 discusses the general principles and assumed architecture of the Infrastructure

Section C3 presents the Domain Specific Failure Logic Model (DSFM) of the Infrastructure

C1. Modification of the BSCU Model (WBS DSFM)

This section briefly returns to the failure logic model of the wheel braking system to implement some trivial – but significant from the composition perspective – modifications.

The key necessary modification is the explicit introduction of the functionally passive components (conceptual data flows) into the model. A new component is introduced into the failure logic model of the BSCU for each communicating pair of components. For the sake of consistency these components are introduced as close as possible to the software component receiving data; Figure 52 and Figure 53 show the new model architectures of the BSCU and its single channel respectively.

Internally, the data flow components are trivial and contain exactly two identical groups of input and output failure modes. Normally, components simply propagate input FMs unchanged. However, the dataflow components may themselves fail, thus, causing any of the output FMs. Figure 109 shows the complete AltaRica code of one of the dataflow components that transmits *CMD* signals. This characterisation is identical for all flow components that transmit this type of data (that is, all flows from both Pedal inputs to *COM* and *MON* modules, as well as the *CMD* flows from *COM* modules to the respective side's *MON* module and BSCU's switch). Similar components are defined for *Validity* status and antiskid (*AS*) signal flows. Although the BSCU dataflow components were manually defined, the simplicity and regular structure of their characterisations allows automatic generation of code for any given AltaRica flow type.

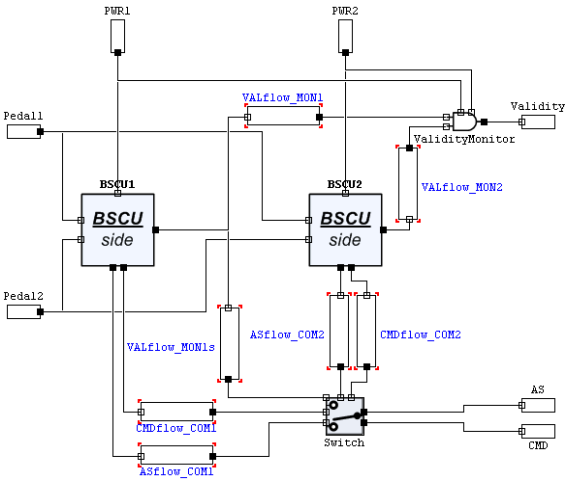


Figure 107 - Revised Model Architecture of BSCU

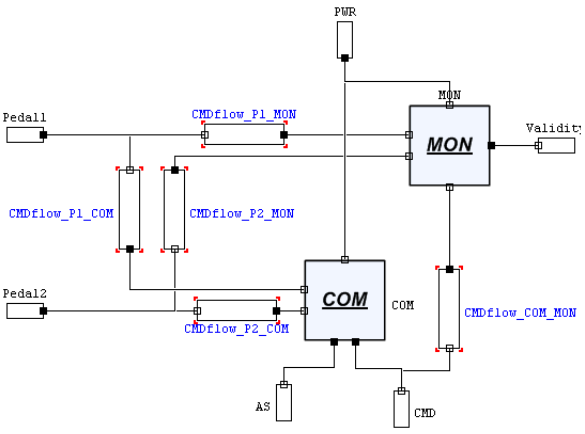


Figure 108 - Revised Model Architecture of a BSCU Channel

```
node WBS_CMDflow
    flow
      In : WBS_ControlFMs : in ;
      Out : WBS_ControlFMs : out ;
    state
      FailSt : {OK,TotalLoss,Corrupt,Spontaneous} ;
    event
      Disintegrates, Corruption, ExternalSource ;
    init
      FailSt := OK ;
    trans
      // Failures and Failure State transitions
      true |- Disintegrates -> FailSt := TotalLoss;
      FailSt != TotalLoss |- ExternalSource -> FailSt := Spontaneous;
      FailSt = OK |- Corruption -> FailSt := Corrupt;
    assert
      // Failure Mode propagation conditions
      Out = ( case {
                FailSt = TotalLoss              : LackOfBraking,
                FailSt = Spontaneous            : InadvertentBraking,
                FailSt = Corrupt
                and In != LackOfBraking
                and In != InadvertentBraking    : TooLittleBraking,
                else In });
edon
```

Figure 109 - Failure Logic Characterisation of CMD Dataflow Component (AltaRica)

It is also important to stress that this modification of the BSCU model is in no way dependent on any of the IMA considerations. The model is merely extended to follow the "best practice" for definition of composable models discussed in Section 4.4 (Chapter Four) of the Thesis.

The second group of modifications is concerned with replication of the internal failures (and failure state transitions) in some of the basic components of the BSCU. Namely, the characterisation of BSCU's *COM* and *MON* components needs to be changed to allow for external causes of failures. Figure 110 shows part of the new characterisation of the *MON* component. Whilst these modifications are specific to integration with IMA (e.g., the reason why there are two external causes of the *ProcessError* failure will become clear in later sections) and seemingly

violate the non-intrusion principle of DSFM composition, they are only necessary because of the limitation of the AltaRica OCAS language (see section 4.5.2 of the Thesis) and, again, are trivial and easily automatable.

```
node WBS_MonitorModule
  flow
    Pedal1 : WBS_ControlFMs : in ;
    Pedal2 : WBS_ControlFMs : in ;
    CMD : WBS_ControlFMs : in ;
    PWR : WBS_PowerFMs : in ;
    Validity : WBS_ValidityFMs : out ;
  state
    FailSt : {OK,StuckPos,StuckNeg} ;
  event
    ProcessStuck, ProcessTerminated, ProcessorError,
    ExternalProcessStuck, ExternalProcessTerminated,
    ExternalProcessorError1, ExternalProcessorError2 ;
  init
    FailSt := OK ;
  trans
    // Failure state entry logic
    PWR = OK |- ProcessTerminated -> FailSt := StuckPos;
    FailSt = OK and PWR = OK |- ProcessorError -> FailSt := StuckPos;
    FailSt = OK and PWR = OK |- ProcessStuck -> FailSt := StuckNeg;

    // Emulating non-independent synchronisations (DSFM composition)
    PWR = OK |- ExternalProcessTerminated -> FailSt := StuckPos;
    FailSt = OK and PWR = OK |- ExternalProcessorError1 -> FailSt :=
StuckPos;
    FailSt = OK and PWR = OK |- ExternalProcessorError2 -> FailSt :=
StuckPos;
    FailSt = OK and PWR = OK |- ExternalProcessStuck -> FailSt :=
StuckNeg;
```

Figure 110 - Revised BSCU Monitor Component with "External" Failure Causes

Finally, the entire failure logic model of the WBS is converted into AltaRica OCAS equipment node and “frozen” for further editing.

C2. Introduction to the Computation Infrastructure and Architecture Description

This section outlines some of the key principles of Integrated Modular Avionics (section C2.1) and presents the assumed architecture of the platform (section C2.2)

C2.1 Introduction to the Integrated Modular Avionics

Having emerged in late 1990s Integrated Modular Avionics (IMA) emulates the traditional federated approach to control of aircraft systems whilst improving supportability, reusability, modifiability, obsolescence management and weight characteristics of controller hardware.

The improvements are achieved by segregation of provision and management of the shared "computation" and "communication" resources into a dedicated aircraft level function and, thus, a dedicated resource or infrastructure system (similar to electrical and hydraulic power generation and distribution systems). This system, IMA, provides a standard interface (API) to application software (organised into "partitions") and contains both computer hardware and operating system software. The specification of the operating system for civil aviation IMA is contained in ARINC 653 standard [8] which, in addition to the API layer, recognises the hardware interface layer or "Core Executive" (CO-EX) as shown in Figure 111.

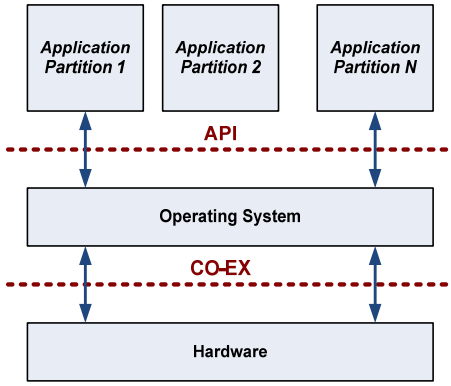


Figure 111 - Layers of the IMA

The key concept of the IMA is *partitioning, which* provides conceptual bridge between the traditional "federated systems" approach (mentioned above) and that of the IMA. Partitioning is the IMA mechanism used to ensure that each application is allocated sufficient resources (e.g., memory and processing time) that no other application can access. Two types of partitioning are traditionally identified:

- *Spatial partitioning* concerned with allocation and segregation of data storage resources and communication channels. This is achieved by pre-allocating each major application software component with a unique area of memory and communication channels at the configuration time. The Memory Management Unit detects and prevents any call from a partition to the memory it has not been allocated.
- *Temporal partitioning* concerned with policing access to a shared processing resource. This is currently achieved by the fixed periodic scheduling with the precise schema determined at the configuration time (thus, by construction, preventing partition overruns).

From the IMA perspective the application software is organised into major components (*partitions*) and elementary components (*processes*). Whilst protection described above is applied to partitions, processes within the same partition are not that strongly segregated. For example, scheduling of the processes is not necessarily fixed and, thus, malfunctions (e.g., exposed design errors) in one process may affect the timely execution of other processes.

In terms of physical architecture, the principal elements of the IMA are *CPIOM*s. Each CPIOM is typically organised as a cabinet of electronic boards with a dedicated CPU board, a number of input and output boards (including a dedicated network interface board), as well as a shared power supply unit and a bus backbone that connects all boards. In addition to the CPIOMs, the IMA architecture often contains a number of generic Input Output Modules (IOMs) for interfacing with the "non-IMA world", as well as interfaces to system-specific modules (referred here as LRUs or Line Replaceable Units); the latter are not considered to be part of the IMA system (beyond their network interface). In line with the standard aerospace practice, the CPIOMs and the IOMs are organised into two "sides" which are typically independently powered and physically located in different parts of the aircraft in order to minimise incidence of common modes of failure and exposure to common threats.

As was mentioned above, the key goal of the IMA is to provide application software with a common and stable abstraction layer for the hardware and network implementations. As such, the IMA may in principle utilise a wide range of network technologies. In practice, today, the IMA is most closely associated with the AFDX network [7] – an Ethernet-based switched redundant network designed specifically for avionics applications. Similarly to the IMA emulating the federated processing, the AFDX emulates a deterministic point-to-point network through the concept of *Virtual Link* (VL). Each VL emulates a point-to-point communication channel between one source *End Node* (EN) and a number of destination ENs. Similarly to partitions, the runtime parameters of each VL (e.g., bandwidth) are determined off-line by the configuration of the ENs and the network switches.

To achieve high reliability, the physical architecture of the AFDX is based on full duplex interconnections and two redundant networks. Each network connection consists of two separate twisted pairs for transmit and receive channels respectively. The dual networks (to an extent analogous to the two IMA "sides") consist of two sets of switches and cables and provide redundant communication paths between any two ENs. The end nodes themselves, however, are not duplicated. The data sent over a virtual link is transmitted on both networks by the source EN; the "receiving" EN accepts the first valid frame and discards the second valid frame (if any).

C2.2 Architecture of the Infrastructure

In this case study, a simplified architecture of the IMA is assumed (Figure 112).

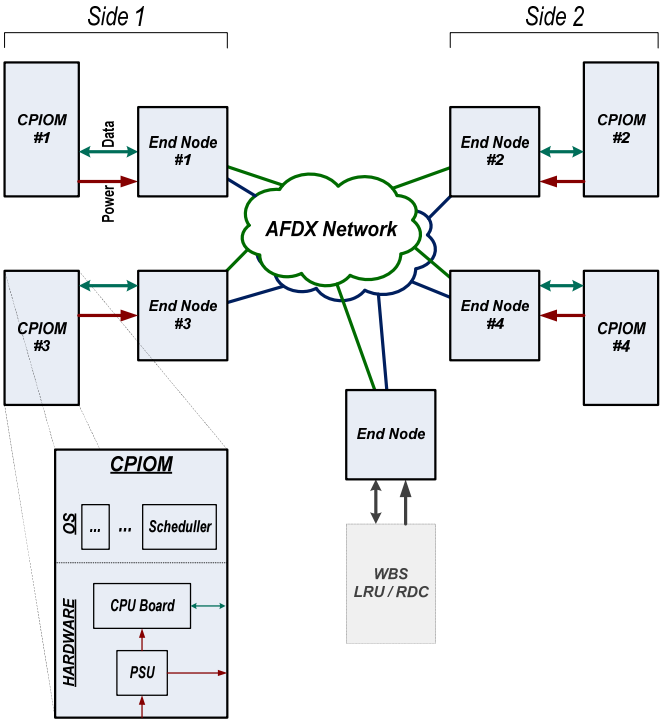


Figure 112 - Simple IMA Architecture

The system consists of four identical CPIOMs. These modules have no I/O cards except for an AFDX end node (one per each CPIOM); the latter is considered external to the CPIOM and is connected to it through a data bus as well as a DC power supply. The CPIOM contains three major hardware sub-components: a Power Supply Unit (PSU), a CPU board, and a Memory Management Unit (MMU). It is also recognised that the CPIOM "contains" operating system software that can be organised into a number of key conceptual components (e.g., *Scheduler*). Overall the CPIOMs are grouped into two "sides" powered from different AC busbars: the CPIOMs 1 and 3 form side 1 and the CPIOMs 2 and 4 – side 2.

Each CPIOM's End Node has seven ports: one (mentioned above) for power provision from the CPIOM's PSU, as well as three pairs of communication ports (one for communication with the CPIOM via data bus, and two pairs for communication with two redundant AFDX switches). Inside the EN, each communication port is connected to a dedicated "port interface" component and to a data processing "pipeline" that consists of a number of memory units and redundancy management components. There are two redundancy management components: one concerned with duplication of output data over two redundant networks (*WriteRedundancy*), and another – with the consolidation of redundant input data streams (*ReadRedundancy*). Both redundancy management components are driven by an internal clock, whereas port interfaces are assumed to

be asynchronous. Between every port and its adjacent redundancy management component is a memory unit. All components are powered from the same DC power input port.

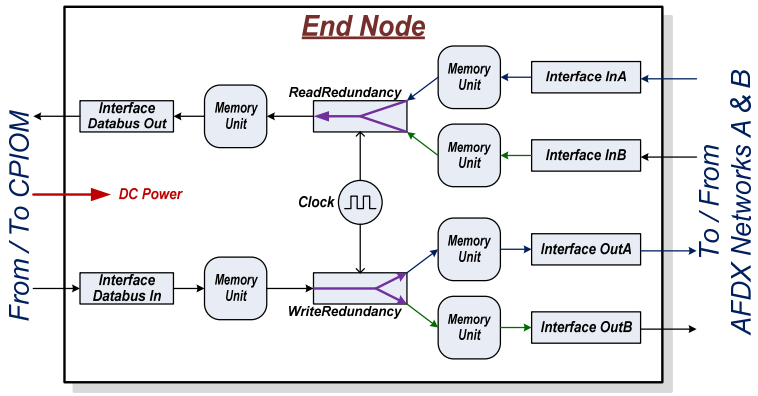


Figure 113 - Internal Structure of an End Node

As overcurrent conditions in the AFDX network are a known threat to continuous operation of the network components as well as the CPIOMs, each EN port interface component is protected from dangerously high current through an integrated circuit breaker. However, the memory units and redundancy management components do not include such protection. Consequently, in presence of external overcurrent, if interface protection fails, memory units and redundancy management components will both fail and propagate the threat further to the adjacent components (including the CPIOMs or the network switches if applicable).

In addition to the four CPIOM's End Nodes, the IMA contains an additional EN connected to a WBS LRU. As this is a bespoke hardware unit that hosts the BSCU Switch and the Validity Monitor components, it is considered to be part of the WBS rather than the IMA system; however, the LRU's EN interface with the IMA *is* clearly part of the AFDX network.

The CPIOMs' and the LRU's end nodes are connected to two redundant switching networks – Network A and Network B (Figure 114). Each network consists of two cascaded switches and a number of duplex twisted pair cables. Switches are externally powered with two redundant networks powered by the same busbars as the CPIOMs' sides 1 and 2 respectively.

In addition to the power port, each switch has four pairs of data ports. Similarly to the ENs, each port is connected to a dedicated port interface component and a memory unit. The eight memory units are connected to the switching fabric component that forms the "core" of the switch. The main function of the fabric is to copy the data between appropriate input and output ports' memory units (as determined by the configuration of the switch and the addressing portion of the AFDX frame concerned); the fabric also performs traffic policing and frame filtering duties (to ensure compliance with the Virtual Links parameters). Finally, the switching fabric is not

protected from overcurrent threats and its behaviour, when exposed, is similar to that of memory units described above.

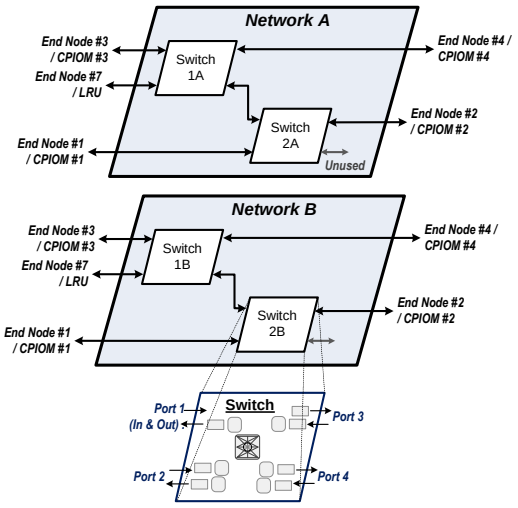


Figure 114 - Redundant AFDX Networks

C3. Model Description

The architecture of the IMA DSFM is shown in Figure 115. It identifies key components (*CPIOMs*, *End Nodes*, *AFDX cables* and *Switches*), internal FM flows between these components, and "external" IMA output FMs. As the *EndNode7* is connected to the WBS LRU that is considered outside the scope of the IMA (and instead – part of the WBS domain), the data input flows of this component are permanently set to *ok* (by the IMA equipment-level assertion). The power input is bound (again by assertion) to outputs of both *AC1* and *AC2* "stubs" so that the flow takes value of *PowerOmiss* if, and only if, both stubs produce *PowerOmiss* on their outputs[66].

The following two sub-sections present failure modes of the IMA, which can be organised into two groups:

a) "*Internal" Failure Modes and FM flows*: that is, flows between IMA components shown as connectors in Figure 115

b) *"External" failure modes*: which form the IMA interface to "subscribing" systems and are related to the high-level functions of the IMA (shown as unconnected output ports of the CPIOMs, the ENs and the Switches in Figure 115)

Subsequent sections C3.3 and C3.4 outline key modelling principles with respect to all major components of the infrastructure DSFMs.

[66] In other words, it is assumed that this end node is powered by both busbars (with a single busbar being sufficient for correct functioning of the component). The justification for this can be found in the WBS description contained in Chapter Three of the Thesis (where the *Validity Monitor* of the BSCU is connected to both power inputs)

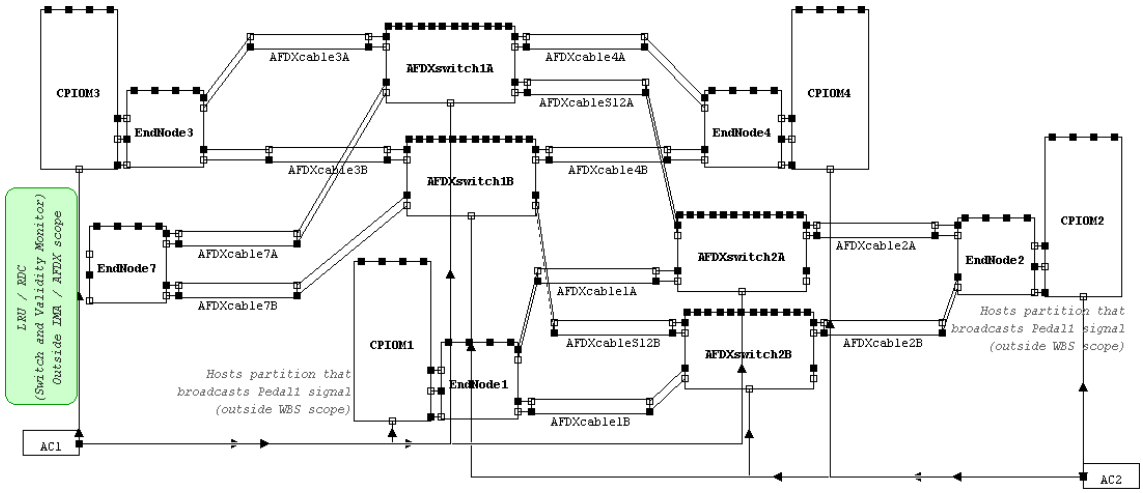


Figure 115 - Architecture of the IMA DSFM

C3.1 Infrastructure Failure Modes: "External" Output FMs and IMA Functions

With respect to the second group of FMs, whilst it is believed that it is unrealistic to expect WBS (and other "subscriber" systems') engineers to model the dependence of their system(s) failure behaviour *on* the infrastructure explicitly (i.e. as input failure modes of software- and passive dataflow- components), the services offered by the IMA *to* subscribers can clearly be considered part of the IMA engineering domain. However, specification of such "external" output FMs is not a trivial matter for the IMA system. In the author's experience the current industrial practice is to consider a small number of failure modes of the entire IMA platform such as total loss of the IMA system, partial loss (typically – loss of one "IMA side") as well as partial and total loss of the AFDX network.

This approach essentially assumes a single interaction between the IMA and subscriber systems in terms of "provision of the IMA service" and seems too coarse in the context of failure logic modelling. At the other extreme, operating systems response for each of the service requests identified in ARINC 653 (such as *RECEIVE_QUEUING_MESSAGE* and *STOP_SELF*) can be considered as an interaction in its own right and associated with a number of failure modes. However, service requests appear to be too detailed as a basis for construction of manageable failure logic models. Furthermore, they only identify intended interactions, so failure modes associated with unintended interactions – such as corruption of a partition's memory – would not be identified.

To achieve a reasonable trade-off between too coarse and too detailed views of the computational platform, in this case study the approach proposed by Conmy [34-36] is adopted in a simplified form. Conmy proposes six higher-level IMA functions:

1. Provision of secure and timely data flow to and from applications and input/output devices
2. Controlled access to processing facilities
3. Provisions of secure data storage and memory management
4. Provision of consistent execution state
5. Provision of health monitoring and failure management
6. General provision of computing capability

Each of Conmy's functions can be considered as an interaction initiated by the IMA system (or more specifically, by each CPIOM) and can be associated with the set of failure modes. Since the goal of this case study is not the construction of an accurate failure logic model of the IMA system, but rather the demonstration and evaluation of the DSFM composition approach, Conmy's approach is simplified by omitting functions (4) and (5) from consideration. In terms of health monitoring and failure management (5), this is a pure simplification of the IMA system, whereas omission of the "provision of consistent execution state" function (4) can be further justified by the model's focus on run-time behaviour of the system. Furthermore, as the latter function is concerned with loading of the applications data into IMA modules from the datastore (typically, following an upgrade or modification [34, 35]), it may be more efficiently treated as a separate domain (with failures of loading process as well as corruption of the datastore modelled in a different DSFM).

Each of the remaining four functions is associated with a simplified set of failure modes (predominantly identified by application of the *Omission* and *Value* keywords) and modelled as an enumerated output flow of *CPIOM* components in AltaRica model (so that each *CPIOM* has four enumerated output flows). Table 14 lists such *CPIOM* outputs, their enumeration symbols along with a brief description of the failure modes they denote.

Table 14 - Classes of the "External" IMA Output Failure Modes (IMA DSFM)

Relevant IMA Components	AltaRica Output Flow	Enumeration Symbol	FM description
CPIOMs	Communications	TotalLoss	CPIOM fails to provide support to any of the communication channels it is responsible for
		Partial Loss	CPIOM fails to provide support to one or more (but not necessarily all) communication channels it is responsible for
		AddressingFailure	CPIOM directs data to the wrong software component (partition) or output port (this may affect one or more channels)
		ok	Failure Mode Privative (function is provided as intended)
	Computing	Incorrect	CPIOM performs computation required by one or more partitions incorrectly
		Stuck	CPIOM halts computation of one or more partitions or processes (e.g. due to corruption of process execution state or CPU failure)
		ok	FM privative
	DataIntegrity	TotalLoss	Data stored or transmitted by, to or from all partitions supported by the CPIOM is lost/ unusable
		PartialLoss	As above but for a subset of partitions
		Corruption	Data is corrupted (affects one or more partitions)
		ok	FM privative
	Scheduling	TotalLoss	CPIOM fails to schedule any of its partitions appropriately (including providing too little execution time)
		PartialLoss	As above for a subset of partitions
		PrioritiesLost	CPIOM is incapable of scheduling one or more processes as intended (e.g. due to corruption or loss of priorities data)
		ok	FM privative
End Nodes, AFDX Switches	NetworkInfrastructure*XXX*[67]	TotalLoss	All VLs' data sent along this path is lost (not transmitted)
		PartialLoss	Data sent along this path by one or more VLs is lost
		DataCorruption	Data sent along this path by one or more VLs is corrupted (in such a way that corruption cannot be detected by the network component(s))
		AddressingCorruption	Data sent along this path by one or more VLs is misrouted to wrong destination
		ok	FM privative

[67] Components have a number of *NetworkInfrastructure* output flows (related to all paths through the component) – "*XXX*" is replaced by the path identifier. For instance, output flows form every End Node components include *NetworkInfrastructureOutA*, *NetworkInfrastructureOutB*, *NetworkInfrastructureInA* and *NetworkInfrastructureInB*.

Of course IMA subscribers can also be affected by the malfunction of network components (switches and end nodes); consequently, these components also have "external" output failure modes. It is assumed that they interact with subscribers in terms of provision of a single function – provision of network infrastructure – associated with four failure mode classes (*TotalLoss*, *PartialLoss*, *AddressingCorruption* and *DataCorruption*). However, these failure modes are associated with each *path* through a network component. So, for example, AltaRica characterisation of an end node has four enumerated output flows (with the above enumeration symbols): two for propagation of data from the CPIOM to the redundant AFDX networks and two for the reception of data (namely, *NetworkInfrastructureOutA*, *NetworkInfrastructureOutB*, *NetworkInfrastructureInA* and *NetworkInfrastructureInB* respectively); similarly, network switches have 12 sets of network infrastructure output FMs (and, thus, 12 enumerated output flows[68]).

C3.2 Infrastructure Failure Modes: "Internal" FMs

Most of the internal FM flows within the IMA system are related to interactions in terms of data signals between CPIOMs, End Nodes, AFDX cables and switches (and their respective sub-components). In the simplified DSFM, only physical aspects of these interactions – in terms of electrical current – are considered. Consequently, each network or CPIOM bus signal link is associated with two failure mode classes – modelled as *SignalOmis* and *HighCurrent* enumeration symbols in AltaRica. The former denotes total omission of the electrical signal and the latter an overcurrent condition that potentially threatens the physical integrity of the IMA's hardware. So, for example, an end node component has three groups of such output failure modes – associated with two AFDX output ports and the CPIOM bus port – as well as three "symmetrical" groups of input FMs. Note also that AFDX cables have two sets of input and two sets of output FMs since they model two twisted pairs dedicated to propagation of signals in both directions.

In addition to the dependence on interactions over data signal links, all hardware components (with exception of cables) also depend on the power supply, which, in this DSFM, is associated with the single omission FM (*PowerOmiss*). Note that the CPIOMs and the Network Switches are powered from two AC busbars external to the system (but modelled in AltaRica as two trivial "stub" components (since the language does not permit free system-level inputs), whereas End Nodes are powered indirectly by the DC current provided by "their" CPIOM (or, in case of *EndNode7* – LRU)

[68] These are all called *Infrastructure<X>to<Y>*, where *<X>* is an incoming port of the switch and *<Y>* - outgoing (e.g. "*Infrastructure 3to2*").

C3.3 Failure Logic Model: Network Components

The IMA DSFM contains three types of top-level network components: AFDX cables, End Nodes and Switches; the latter two are complex components, whereas the cables are modelled as trivial basic components.

The failure logic of complex network components is illustrated on an End Node. Each node contains a number of basic components that represent "physical" sub-components of the node as well as four "virtual" components that determine *NetworkInfrastructure* failure modes for each of the paths through the node respectively. Figure 116 shows the former group of basic components within a node organised, according to node design, into two splitting / merging pipelines. Each path through the node passes through two port interface components, two memory units and either a *ReadRedundancy* or a *WriteRedundancy* component (depending on the path's orientation). All of these components are sensitive to *PowerOmiss* input FM of the node, dataflow signal output FMs of their "neighbours" and – in case of the redundancy management components – *ActivationOmiss* output FM of the shared *Clock*.

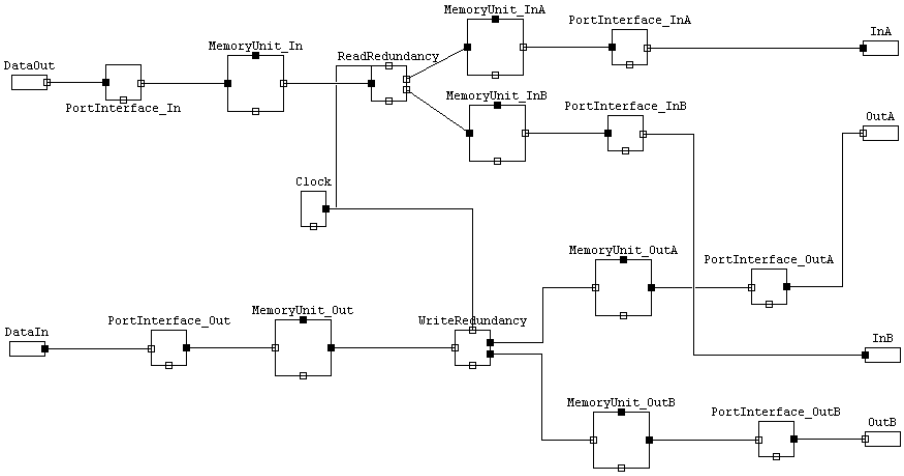


Figure 116 - Failure Logic Model of an End Node (Data Propagation "Pipelines" Only)

All port interface components are modelled by identical AltaRica components in this simplified model. Each such component is capable of exhibiting a *SignalOmiss* FM as a result of the similar input FM, *PowerOmmis* FM, or an internal failure (*FailLost*). Furthermore, when exposed to a *HighCurrent* input FM the component moves to a failure handling state (through an instantaneous *Update* event that models the void state transition trigger) and, as a result, also exhibits a *PowerOmiss* FM. However, the internal protection from overcurrent may fail (*ProtectionFail* event) which will obviously result in propagation of the *HighCurrent* FM. Finally, an internal failure of the component (*FailHighCurrent*) may also result in this output FM. Figure 117 shows the complete characterisation of the component's failure logic in AltaRica.

The failure logic of *MemoryUnit* components is conceptually different. These components only exhibit *SignalOmiss* following an internal failure (*TotalFailure* which moves the component into a *Lost* failure state) or an exposure to either *PowerOmiss* or *HighCurrent* input FMs (the latter also moves the component into a *Lost* state since memory is assumed to be unprotected from overcurrent conditions).

```
node IMA_PortInterface
   flow
     Out : {ok,SignalOmiss,HighCurrent} : out ;
     In : {ok,SignalOmiss,HighCurrent} : in ;
     Pwr : {ok,PowerOmiss} : in ;
   state
     FailSt : {OK,Lost,Threatens,ProtectionLost} ;
     FHState : {OK,Tripped} ;
   event
     Update, FailLost, FailHighCurrent, ProtectionFail ;
   init
     FailSt := OK ;
     FHState := OK ;
   trans
     // Failures & Failure States
     FailSt != Lost |- FailLost -> FailSt := Lost;
     FailSt != Lost and FailSt != Threatens |- FailHighCurrent -> FailSt := Threatens;
     FailSt = OK |- ProtectionFail -> FailSt := ProtectionLost;
     //Failure Handling State: Protection against dangerous network current/voltage
     In = HighCurrent and FailSt != ProtectionLost and FHState = OK |- Update -> FHState := Tripped;
   assert
     // Propagation conditions (electrical network)
     Out = ( case {
               FailSt = Lost or Pwr = PowerOmiss or
               (FHState = Tripped and FailSt != Threatens) or
               (In = SignalOmiss and FailSt != Threatens)          : SignalOmiss,
               Pwr = ok
               and ( FailSt = Threatens or
                    (In = HighCurrent and FailSt = ProtectionLost
                    and FHState != Tripped))                       : HighCurrent,
               else ok})
   extern
     law <event Update> = Dirac(0) ;
edon
```

Figure 117 - AltaRica Characterisation of *PortInterface* Component

Notably, the *SignalOmiss* input failure mode (in absence of the previously stated conditions) does not result in an output omission since the loss of input data would not result in deviations of the memory units' *electrical* interface. However, in this case, the correctness of the data held by the memory will clearly be affected. To account for this effect, the memory unit can exhibit a standard set of *NetworkInfrastructure* output failure modes. When the unit is in *Lost* state or exposed to signal or power omissions it will exhibit a *TotalLoss* FM. Internal failures of component can also result in *PartialLoss*, *DataCorruption* and *AddressingCorruption* FMs (these failures, however, will have no effect on any of the signal output FMs of the component). *MemoryUnits* are the only components in an End Node capable of exhibiting infrastructure failure modes. Figure 118 shows the AltaRica characterisation of the *MemoryUnit* (identical for all units in end nodes and switches).

```
node IMA_MemoryUnit
  flow
    In : {ok,SignalOmiss,HighCurrent} : in ;
    Out : {ok,SignalOmiss,HighCurrent} : out ;
    Pwr : {ok,PowerOmiss} : in ;
    NetworkInfrastructure : {ok,TotalLoss,PartialLoss,DataCorruption,AddressingCorruption} : out ;
  state
    FailSt : {OK,Lost,PartialLoss,DataCorrupted,AddressingCorrupted} ;
  event
    TotalFailure, PartialFailure, AddressingCorruption, DataCorruption, Update ;
  init
    FailSt := OK ;
  trans
    // Failure & Failure States
    FailSt != Lost |- TotalFailure -> FailSt := Lost;
    (In = HighCurrent) and FailSt != Lost |- Update -> FailSt := Lost;
    FailSt = OK |- PartialFailure -> FailSt := PartialLoss;
    FailSt = OK |- DataCorruption -> FailSt := DataCorrupted;
    FailSt = OK |- AddressingCorruption -> FailSt := AddressingCorrupted;
  assert
    Out = (case {
              In=HighCurrent                          : HighCurrent,
              FailSt = Lost or Pwr = PowerOmiss       : SignalOmiss,
              else ok});
    NetworkInfrastructure = (case {
              FailSt = Lost or Pwr = PowerOmiss or In = SignalOmiss   : TotalLoss,
              FailSt = PartialLoss                                    : PartialLoss,
              FailSt = DataCorrupted                                  : DataCorruption,
              FailSt = AddressingCorrupted                            : AddressingCorruption,
              else ok});
  extern
    law <event Update> = Dirac(0) ;
edon
```

Figure 118 - AltaRica Characterisation of the *MemoryUnit* Component

Returning to the complex *EndNode* component, as was mentioned in section C3.1, in addition to "physical" FM interfaces (i.e. signal and power FMs), the component can exhibit four sets of *infrastructure FMs* related to four conceptual dataflow channels through the node. To determine which of these output FMs are exhibited by the *EndNode,* four "virtual" components are introduced (*ChannelInA*, *ChannelInB, ChannelOutA* and *ChannelOutB* – i.e. one for each set of FMs). Each of these components is essentially a stateless propagation condition that determines the deviation of the channel's behaviour based on the FMs exhibited by the relevant memory and port interface components. Namely, the status of each channel depends on the infrastructure FMs of two memory units and the data signal FMs of one interface port component. Any output FM of the *PortInterface* results in the *TotalLoss* of the relevant channel(s); in terms of channels' sensitivity to the memory units' infrastructure FMs, "virtual" components essentially implement an "or" gate. Figure 119 shows the complete OCAS representation of the model architecture of an End Node (for clarity Power FMs are not shown).

The failure logic models of AFDX switches (Figure 120) are conceptually similar to those of end nodes where the logic of the only new basic component – *SwitchingFabric* – can be seen as *a multi-port extrapolation of the memory unit*. The "virtual" channel components of the switch consolidate four groups of failure modes of switch components (the infrastructure FMs of two *memory units* and the *switching fabric*, as well as the signal FMs of one *port interface*

component). The regular structure of virtual components and their FM dependencies makes it possible (and the sheer number of these components and their FM flows makes it efficient) to automatically generate part of the switch model in AltaRica[69].

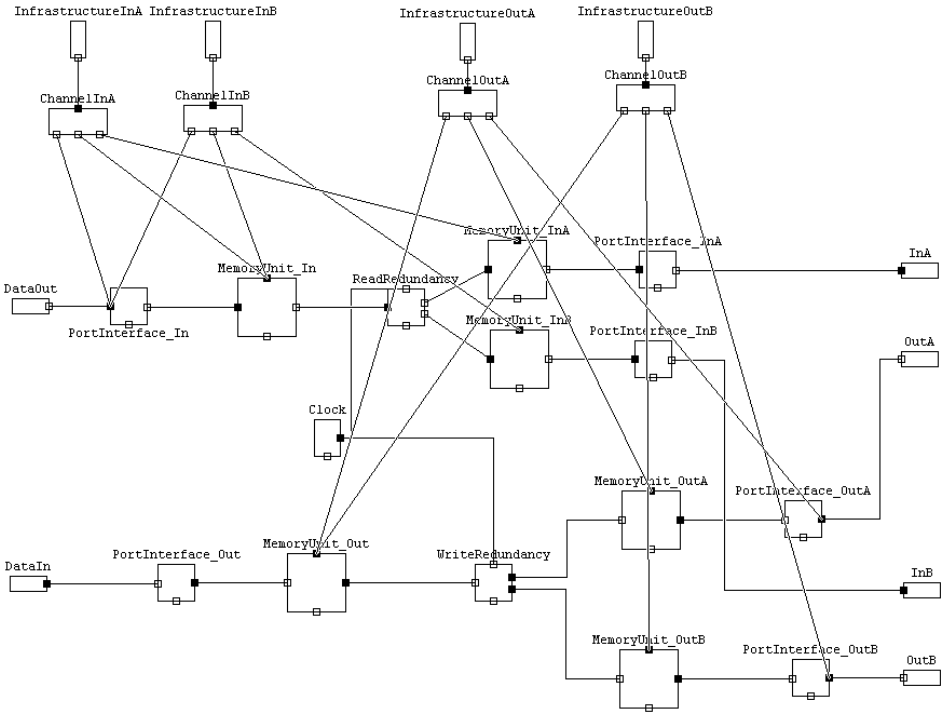


Figure 119 - Structure of the *EndNode* Complex Component

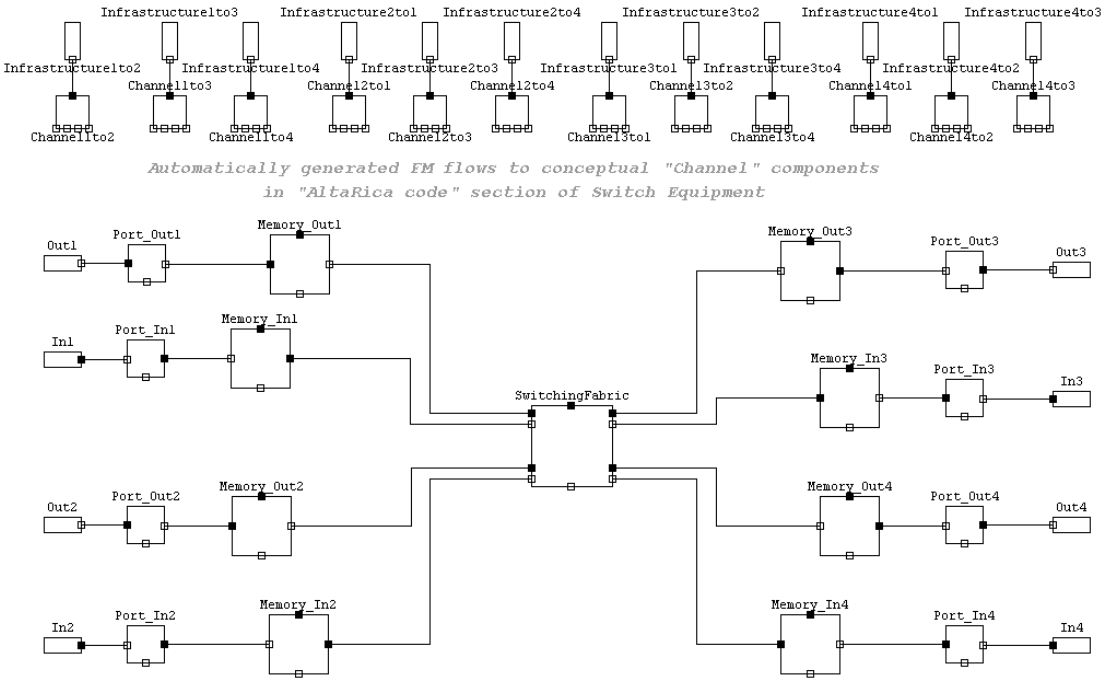


Figure 120 - Structure of the *AFDXswitch* Complex Component Model

[69] The flows were generated (in equipment assertions- rather than graphical flow- form) from a trivial application of *concatenate* function in MS Excel worksheet. Each switch contains 48 automatically generated flows/assertions (four for each of the 12 possible channels through the switch)

C3.4 Failure Logic Model: CPIOMs

The failure logic model architecture of CPIOMs (modelled as complex components – see Figure 121) is intentionally simplified to the extreme. In terms of hardware, only the *CPU board* and the *power supply unit* (*PSU*) are recognised. The *memory management unit* (*MMU*) FLM component represents a *combination* of the hardware MMU and the operating system services related to data storage and retrieval. Both MMU and CPU board are sensitive to the *PowerOmiss* failure mode of the *PSU* and to the *HighCurrent* input FMs of the CPIOM (ultimately exhibited by the CPIOM's end node). The latter causes both components to enter *Lost* failure states, whereas the former propagates out as *Lost* and *TotalLoss* output FMs of the *CPUboard* and *MMU* respectively.

The "amalgamated" MMU is the sole provider of the *DataIntegrity* function, and thus the associated CPIOM's failure modes (see Figure 121). With exception of *TotalLoss* mentioned above, these output FMs can only be caused by internal failures of the *MMU*.

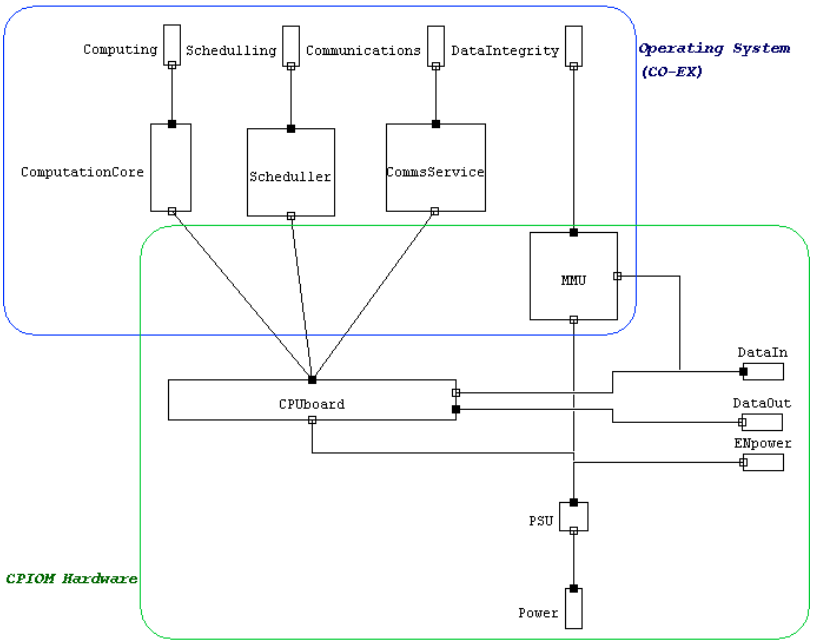


Figure 121 - Structure of the CPIOM Complex Component Model

In contrast, *CPUboard* does not directly cause any of the CPIOM FMs associated the IMA functions; instead it influences three operating system conceptual components concerned with provision of scheduling, communications service and core computation. The board component can exhibit three "*hardware services*" FMs: *Lost*, *PartiallyLost* and *Untrustworthy* (with the exception of *Lost*, mentioned above, these FMs can only be generated through *internal failure* of the *CPUboard*). The *Lost* FM propagates through the operating system components and causes *TotalLoss* FMs on all three functions. However, other FMs of the CPU board do not have a "fixed"

effect. Instead, *normal events*[70] (and normal states) internal to OS components determine the effect. To illustrate, Figure 122 shows AltaRica characterisation of the *Scheduling* component.

To conclude the description of CPIOM OS components it is important to note that, with the exception of the *ComputationCore*, OS components may exhibit failure modes as a result of an *internal failure* (e.g. due to *O/S software errors*).

Whilst the CPIOM model may be considered as unrealistically simplistic and can be significantly improved, for example, through a systematic application of LISA method [123] and/or incorporation of Conmy's IMA analysis results [34], a more detailed and internally accurate model is not essential (and even value-adding) in terms of the demonstration of the approach to the DSFM integration.

```
node IMA_CoEx_Scheduller
  flow
    SchedulingFunction : {ok,TotalLoss,PartialLoss,PrioritiesLost} : out ;
    HWservices : {OK,Lost,PartiallyLost,Untrustworthy} : in ;
  state
    FailSt : {OK,Lost,PartitionLost,ProcessLost} ;
    NormSt : {UnaffectedByHW,HWaffectsPartition,HWaffectsProcess} ;
  event
    FatalFailure, DeadlinesCorruption, PrioritiesCorruption,
    CONDITION_ProcessAffected, CONDITION_PartitionAffected ;
  init
    FailSt := OK ;
    NormSt := UnaffectedByHW ;
  trans
    // Scheduler failures (Failure State)
    FailSt = OK |- FatalFailure -> FailSt := Lost;
    FailSt = OK |- DeadlinesCorruption -> FailSt := PartitionLost;
    FailSt = OK |- PrioritiesCorruption -> FailSt := ProcessLost;
    // Sensitivity to HW failure (Normal State / Conditioning Event)
    NormSt = UnaffectedByHW |- CONDITION_PartitionAffected -> NormSt := HWaffectsPartition;
    NormSt = UnaffectedByHW |- CONDITION_ProcessAffected -> NormSt := HWaffectsProcess;
      //(Note: Scheduler is *always* sensitive to total loss of HW services)
  assert
    // Scheduler Failure Modes:
    SchedulingFunction = ( case {
                HWservices = Lost or FailSt = Lost      : TotalLoss,
                FailSt = PartitionLost or
                HWservices != OK
                and NormSt = HWaffectsPartition         : PartialLoss,
                FailSt = ProcessLost or
                HWservices = Untrustworthy
                and NormSt = HWaffectsProcess           : PrioritiesLost,
                else ok})
edon
```

Figure 122 - AltaRica Characterisation of the *Scheduler* Component

[70] As described in Chapter 3, to model the equivalent of FTA conditioning events, FLMM's normal events may be assigned probabilistic characteristic. These events are not counted towards cardinality of the minimal cut sets/sequences in model analysis.

Appendix D:

Summary of the Aircraft Fuel System Review

This appendix expands upon a concise summary of the review of the Aircraft Fuel System presented in Section 6.2.2 of Chapter Six.

The key high-level functions of the system include:

- Storage of fuel
- Fuel management (in-flight)
    - Engine feed (including cross-feed)
    - Provision of fuel reserves to feed tanks
    - Airframe protection load relief
    - Drag reduction (maintaining adequate Centre of Gravity)
    - Aircraft weight reduction (fuel jettison)
- Provision of cockpit interface and indications / annunciation; the later includes
    - Fuel Quantity & Distribution (each tank, overall fuel on board, centre of gravity)
    - Fuel Temperature (in selected tanks)
    - Failure and system status annunciation
- On-ground Refuel / Defuel

In order to complete the review within the set timeframe it was decided not to subject system functionality related to fuel jettison, venting, temperature indication and on-ground operations to a detailed examination. The review has identified the following key characteristics of the fuel system (each described in more detail in the following sections of this Appendix):

(1) Complex mode logic
(2) Intentional architectural limitations
(3) Complexity of scale and design decomposition
(4) Circular dependencies and loops
(5) Time-dependency and reliance on consumable resource

D1. Complex Mode Logic

The fuel system is highly reliable and designed to provide continuous operation in presence of failures. The system can be operated in a large number of modes with predominantly automatic reconfiguration upon detection of failure. A key example of such reconfiguration can be derived

from a consideration of the liquid-mechanical architecture of the system and its fuel transfer paths.

The system consists of eleven tanks: 4 feed tanks (one per engine), three reserve tanks on each wing (Outer Tank, Mid Tank and Inner Tank) and a Trim Tank. All wing tanks are connected by two transfer galleries – Gallery A and Gallery B (see Figure 123 for a schematic representation); two additional galleries are used for connecting the wing tanks' galleries to the trim tank and for cross feed.

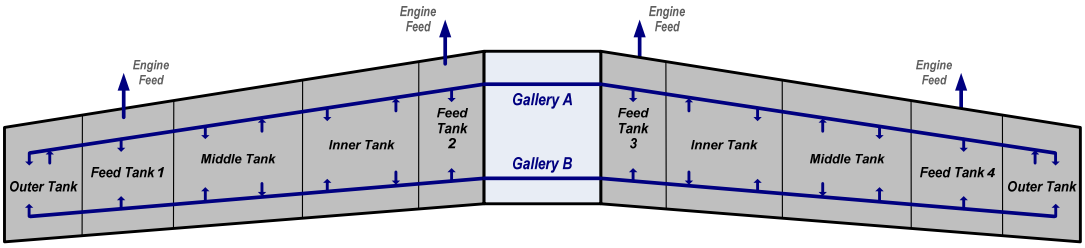


Figure 123 - Schematic of Wing Tanks Architecture

Overall, the system must support three types of fuel transfer: main transfers (from the reserve tanks to the feed tanks), load alleviation transfers (from the inner or mid tanks to outer tanks at take-off and in the reverse direction before landing) and centre of gravity transfers (from the trim tank to the wing reserve tanks). In a simplified view the system can be seen as progressing through a number of modes (see Figure 124). In practice, however, some of the transfers may take place in parallel and in normal operation each transfer is allocated to a particular transfer gallery (and the corresponding valves and pumps).

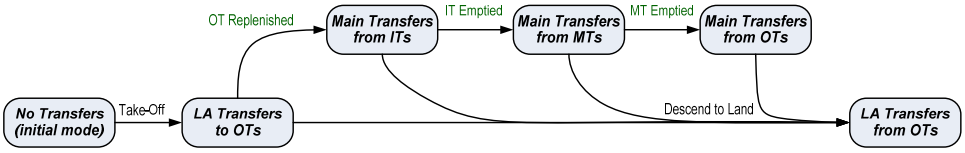


Figure 124 - Simplified Phases of Operation

If a single failure of pumps or valves is detected and if that failure affects the feasibility of the transfers the system enters one of nine alternate modes (termed "workarounds") reallocating transfers to different galleries or, in some cases, inhibiting certain non-essential transfers. Workarounds are also used for some cases of double failure. Figure 125 shows a significantly simplified schematic of a mode model. First, it is important to note that the shape of the model suggests that a hierarchical mode modelling approach proposed by Papadopoulos [115] (see section 5.7 of Chapter Five) is unlikely to yield significant value for this system (with transitions between sub-states of different states, the hierarchy is likely to be a purely decorative facility).

Second, it should be noted that the model shows a *degree* of independence between the phase-of-flight and the workaround modes.

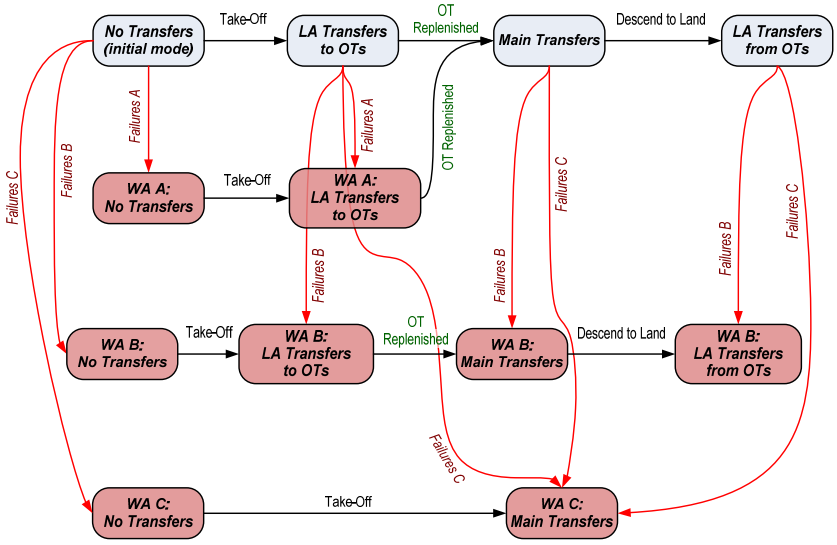


Figure 125 - Simplified Mode Model of the Fuel System

In terms of independence it should also be noted that workarounds and phases-of-flight are not the only modes found in the fuel system. The Fuel Quantity Management (Sub-) System (FQMS), consisting of two specialised Remote Data Concentrators (RDCs) and a number of software partitions (implemented on aircraft IMA), is organised in the standard “two-side” aviation architecture. Upon detection of abnormal FQMS behaviour (either through direct monitoring or through detection of abnormal fuel distribution) the control of the Fuel System is handed over to the passive side. This yields a set of FQMS modes that is largely *orthogonal* to the workarounds and phases discussed above.

D2. Intentional Architectural Limitations

In addition to the modes described in the previous section, when system operation is fully automated, the fuel system can also be operated in a *manual mode* with pilots controlling transfers directly through the cockpit interface. The manual mode can be initiated by the pilots at any time regardless of the system status; in addition it is entered when FQMS detects an unhandled combination of two or more failures.

What makes manual mode remarkable is that the pilots do not have full control of the system. For example, whilst they can initiate any main, load alleviation or centre of gravity transfers, these transfers will only take place on their respective pre-allocated galleries; if the gallery equipment (a pump or a valve) is affected or if the gallery itself is suspected of leaking, the pilots *cannot* utilise redundancy of the liquid-mechanical architecture. This limitation cannot be justified in terms of the system model: the justification comes from a trade-off between fuel transfer

availability in rare circumstances (when manual mode is required) and the need to minimise the likelihood of pilot error (that is increased by complex control tasks). The effect of the limited mode on a failure logic model of the fuel system would be that certain fuel transfer paths (via alternate routes) are unavailable in Manual Mode even in the absence of any relevant equipment failures.

D3. Complexity of Scale and Design Decomposition

As was indicated above the fuel system can be divided into two distinct parts:

- A hydro-mechanical subsystem comprising 11 fuel tanks, four fuel galleries (including cross-feed), and the associated valves and pumps
- A Fuel Quantity Management Sub-System (FQMS) consisting of in-tank probes, pump & valve sensors, two specialised remote data concentrators (RDC) and various software partitions implemented on the IMA platform.

FQMS controls the pumps and valves of the liquid-mechanical sub-system either through discrete signals (via dedicated electronic interfaces on FQMS CPIOMs) or via AFDX broadcasts.

The two sub-systems are designed in relative isolation, suggesting that any effective safety analysis process should reflect this decomposition. Simplified partial failure logic models of the two subsystems have been constructed during the review process. These experiments have indicated that at early stages of development two sub-systems can be meaningfully analysed in their own right. The analysis of the liquid-mechanical architecture can identify failures that may lead to fuel becoming unusable (either by being isolated in the reserve tanks or through leaks). Analysis of the FQMS could identify failure scenarios that lead to the management subsystem inadvertently commanding transfers, erroneously commanding transfers on compromised paths or failing to command transfers when needed. The models of the subsystems, however, could not be effectively directly composed through input and output FM flows. Instead, the composition would require the definition of complex “conceptual” or “virtual” components to explicitly represent *transfers as model entities*. Every transfer component would have to contain a sub-component for each redundant “path”. Each path, in turn, would contain a representation of valves, pumps and a fuel gallery. The model of paths and transfers could be integrated with the liquid-mechanical failure logic model as two views of the same system elements (i.e. through direct links between failures of the corresponding components in two models) and with the FQMS failure logic model through failure mode flows.

Finally, review of the fuel system documentation has uncovered a case when the same set of components (the liquid-mechanical architecture) is considered by *separate safety analyses* from two different viewpoints: an ability to support effective transfers and the loss of fuel through

leaks[71]. This is a clear case of two engineering domains having been defined over the same scope but from different viewpoints (see Table 5 in Section 4.3.1, page 129). This observation has provided some further *confidence in necessity of a flexible conceptual framework for platform decomposition* (advocated in Chapter Four)

D4. Circular Dependencies and Loops

Whilst the problem of loops in failure logic models is not new, reviews of the fuel system along with preliminary partial experiments have provided a new insight into this issue. The fuel system models contained two types of loops: model loops due to control feedback (e.g. closed-loop control schemas) and strong circular dependencies in abstract models.

With respect to the first type, it is important to note that control loops in design models in fact do not necessarily result in loops in the failure logic models. If the sensing strategy is perfect – that is, if failure-free sensors can be assumed to provide correct measurement of physical phenomena regardless of the value of the characteristic being measured – there is generally no failure mode dependency between the "primary" control channel and the feedback / monitoring channel. FM flows are only established by:

- Imperfect sensing strategy (e.g. when some failure modes of the primary channel may cause an unintended behaviour of the sensor)
- The failure logic modelling process if safety engineers wish to reflect "incidental correctness" scenarios accurately (e.g. failure of the sensors combined with coincidental failure modes of the primary channel resulting in apparently correct overall behaviour of the system)

Regardless of the cause this type of failure logic model loops is well recognised and only poses a challenge to "backward search" approaches to model analysis or transformation (such as fault tree synthesis). In contrast, forward search techniques, such as that proposed by Wallace [163] or a sequence generation approach utilised throughout this thesis, are significantly less susceptible to feedback loops and resolution strategies are often trivial (e.g. see Section 5.6.1).

However, failure logic modes may contain strong circular dependencies even in absence of control feedback. The problem is particularly acute for models of reconfigurable and redundant systems which abstract from the details of the actual reconfiguration schema (for example, to assess the safety of the hardware architecture in its own right).

[71] Whilst it was possible to construct a partial simplified failure logic model that combined these two aspects, the resultant model became exceptionally complex (in terms of review and maintenance as well as in terms of computational complexity of the analysis)

This type of loops are best illustrated on a modified simple example of a fuel system routinely used on University of York continuous professional development courses on safety engineering (Figure 126). The system consists of a single feed tank and two reserve tanks. The fuel can be transferred from each reserve tank to the feed tank or, alternatively, across the reserve tanks (through a single dedicated transfer gallery).

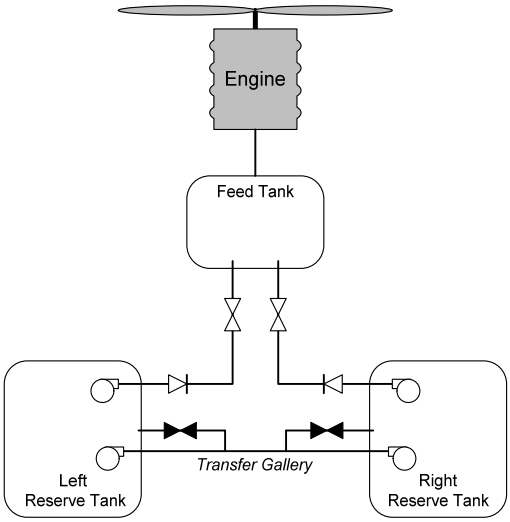


Figure 126 - Schematic of a Trivial Aircraft Fuel System

Considering the failure logic model of the liquid-mechanical architecture of the system (without the details of the controller and, thus, the details of the reconfiguration algorithm) the key failure mode of the reserve tanks is an inability to provide fuel to the feed tank. Each reserve tank exhibits this FM if it is both empty (e.g. through normal exhaustion of the fuel) and incapable of receiving fuel from the 'peer tank'. This logic clearly establishes a circular dependency between the two tanks. What is noteworthy about this dependency is that the naïve loop resolution strategy, based on a delay in FM propagation, results in the incorrect overall behaviour of the system. In particular, analysis of such model would suggest that in the absence of failure of the transfer gallery (and associated valves and pumps) the fuel is never exhausted. In other words, whilst the delay in FM propagation allows analysis and simulation to converge to a stable state, *this equilibrium is erroneous*.

Such strong circular failure mode dependencies can only be correctly resolved by enriching failure logic models with a more complex loop resolution behaviour (that has no direct engineering interpretation). For the above example this behaviour will inject "virtual" transient failure modes (not detectable by the analysis engine) into all FM flows between tanks upon either transfer tank's becoming empty. The resultant model stabilises in the correct state upon the expiry of the 'temporary' FM injection.

Returning to the actual aircraft fuel system – the failure logic model of the liquid mechanical architecture (see previous section) exhibited such strong circular dependencies; the problem was further exacerbated by a number of fuel tanks and presence of two alternative galleries for most fuel transfers.

D5. Time-Dependency and Reliance on Consumable Resource

The naïve system-level failure condition of the aircraft fuel system, implied in some of the above sections, is an inability to provide fuel to the engines. However, this failure condition is obviously reachable in a "nominal case" (i.e. in the absence of failures of any components) – on-board fuel reserves can be depleted through a normal aircraft engine burn. In general, normal consumption of fuel (along with consideration of the leaks) *renders timing a key aspect of the system behaviour*. In fact, even at the level of functional failure condition any realistic specification requires explicit reference to timing characteristics (e.g. "inability to provide fuel to the engines within less than 4 hours of a warning given to the flight crew [necessary for a normal diversion]")

Furthermore, the time dependency as well as the consumable and finite nature of fuel as a resource introduce a strong coupling between flight time, the exact amount of reachable fuel and, even, the rates of fuel flow and leakage. Whilst the FLM Framework provides some facilities for describing deviation from intent in timing and value domains the approach is likely to be too coarse for systems that exhibit this degree of dependence on detailed timing and quantity characteristics. Overall, it is not clear whether the abstraction inherent in any failure logic modelling approach is applicable to a realistic aircraft fuel system.

However, whilst this characteristic of the reviewed system is noteworthy and clearly poses challenges to the FLM Framework presented in this thesis, it was adjudged to be unique to the fuel system. In particular, no other aircraft system relies on a *consumable* resource and precise timing characteristics to this degree: whilst other systems – such as electrical power generation & distribution system and hydraulically powered systems – may rely on some consumable resources (e.g. electrical batteries and hydraulic accumulators) this reliance is either transient (only necessary for successful reconfiguration) or is considered to be the last resort option (that can be pessimistically abstracted from in safety assessment and/or, if necessary, addressed by isolated safety analyses).

Abbreviations

AADL	Architecture Analysis and Design Language
AFDX	Avionics Full-Duplex Switched Ethernet
ARP	Aerospace Recommended Practice (typically refers to the ARP-4761 document)
BSCU	Braking System Control Unit
CCA	Common Cause Analysis
CCF	Common Cause Failure (a type of synchronisation in AltaRica OCAS)
CPIOM	Core Processing and Input/Output Module
DSFM	Domain-Specific Failure Logic Model
EMF	Eclipse Modelling Framework
EN	End Node
EPDS	Electrical Power Distribution System
ETA	Event Tree Analysis
EVL	Epsilon Validation Language
FC	Failure Condition
FDEP	Functional Dependency [gate]
FHA	Functional Hazard Assessment
FLM	Failure Logic Model
	Failure Logic Modelling
FLMM	Failure Logic Metamodel
FM	Failure Mode
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes Effects and Criticality Analysis
FPTC	Failure Propagation and Transformation Calculus
FPTN	Failure Propagation and Transformation Notation
FT	Fault Tree
FTA	Fault Tree Analysis
GUI	Graphical User Interface
HAZOP	Hazard and Operability Studies
HiP-HOPS	Hierarchically Performed Hazard and Origin and Propagation Studies
ICBM	Inter-Continental Ballistic Missile
IF-FMEA	Interface-Focussed Failure Modes and Effects Analysis
IMA	Integrated Modular Avionics
ISAAC	Improvement of Safety Activities on Aeronautical Complex systems [project]
LRU	Line Replaceable Unit
MCS	Minimal Cut Set
MISSA	More Integrated and Cost-Effective Systems Safety Assessment [project]

PAND	Priority AND [gate]
PASA	Preliminary Aircraft Safety Assessment
PSSA	Preliminary System Safety Assessment
SEFT	State/Event Fault Tree
SEI	Software Engineering Institute (of Carnegie Mellon University, USA)
SHARD	Software Hazard Analysis and Resolution in Design
SSA	System Safety Assessment
TLE	Top-Level Event
VL	Virtual Link
WBS	Wheel Braking System

References

1. *Avionics on the A380 (Workshop)*. 2005, Royal Aeronautical Society: London, UK.
2. *Report on the Serious Incident to Airbus A319-111, Registration G-EZAC near Nantes, France on 15 September 2006*, 2009. Aircraft Accident Report: Nr. *4/2009*, Air Accidents Investigation Branch (Department of Transport): London, UK.
3. Abdulla, P.A., et al., *Designing Safe, Reliable Systems Using Scade*, in proceedings of *1st International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, 2004. LNCS-4313, pp. 115-129. Springer-Verlag.
4. Actel Corporation, *Developing AFDX Solutions*. Application Note AC221, 2005. Available from: http://www.actel.com/documents/AFDX_Solutions_AN.pdf [Last accessed: 1 March 2010]
5. Airbus France and ONERA, *Common Cause Analysis: Coupling of Functional and Geometrical Models*. Dissemination Presentation, 2005. Available from: http://www.cert.fr/isaac/doc/EYDLT_PR0503970_v2.pdf [Last accessed: 1 March 2010]
6. Airbus Operations & MISSA Project Consortium. *MISSA Project Website*. Available from: http://www.missa-fp7.eu/ [Last accessed: 1 March 2010]
7. Airlines Electronics Engineering Committee (ARINC), *Specification 664, Part 1: Aircraft Data Network, Systems Concepts and Overview (ARINC 664-1)*. 2002, Aeronautical Radio Inc.: Annapolis, MD.
8. Airlines Electronics Engineering Committee (ARINC), *Avionics Application Software: Standard Interface (ARINC 653-1)*. 2003, Aeronautical Radio Inc.: Annapolis, MD.
9. Åkerlund, O., et al., *ISAAC, a Framework for Integrated Safety Analysis of Functional, Geometrical and Human Aspects*, in *3rd European Congress on Embedded Real Time Systems (ERTS)*. Toulouse, France, 2006.
10. Andrews, J.D., *To Not or not to Not!*, in *18th International System Safety Conference (ISSC)*. Fort Worth, TX, 2000. System Safety Society.
11. Andrews, J.D., *The Use of Not Logic in Fault Tree Analysis*. Quality and Reliability Engineering International, 2001(17): p. 143-150.
12. Andrews, J.D. and J. Dugan, *Dependency Modelling using Fault Tree Analysis*, in proceedings of *17th International System Safety Conference (ISSC)*, 1999, pp. 67-77. System Safety Society.
13. Arnold, A., G. Point, A. Griffault, and A. Rauzy, *The AltaRica Formalism for Describing Concurrent Systems*. Fundamenta Informaticae, 2000. **34**: p. 109-124.
14. ASSERT Project Consortium. *ASSERT Project Website*. Available from: http://www.assert-project.net/ [Last accessed: 1 March 2010]

15. Atkinson, C. and T. Kühne, *Model-Driven Development: A Metamodeling Foundation*. IEEE Software, 2003. **20**(5): p. 36-41.
16. Banach, R. and M.R. Poppleton, *Retrenchment, Refinement and Simulation*, in proceedings of *1st International Conference of B and Z Users*, 2000. LNCS-1878, pp. 304-323. Springer-Verlag.
17. Banach, R. and M. Bozzano, *Retrenchment, and the Generation of Fault Trees for Static, Dynamic and Cyclic Systems*, in proceedings of *25th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, 2006. LNCS-4166, pp. 127-141. Springer-Verlag.
18. Bass, L., P. Clements and R. Kazman, *Software Architecture in Practice*. (2nd ed). SEI Series in Software Engineering. 2003, Boston, MA: Addison-Wesley.
19. Bernard, R., et al., *Experiments in Model-Based Safety Analysis: Flight Controls*, in *IFAC Workshop on Dependable Control of Discrete Systems*. Paris, France, 2007. The International Federation of Automatic Control.
20. Bieber, P., C. Castel and C. Seguin, *Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System*, in proceedings of *4th European Dependable Computing Conference*, 2002. LNCS 2485, pp. 19-31. Springer Verlag.
21. Bieber, P., et al., *Integration of Formal Fault Analysis in ASSERT: Case Studies and Lessons Learnt* in *4th European Congress on Embedded Real-Time Software (ERTS)*. Toulouse, France, 2008.
22. Boiteau, M., Y. Dutuit, A. Rauzy, and J.-P. Signoret, *The AltaRica Data-Flow Language in Use: Modelling of Production Availability of a Multi-State System*. Reliability Engineering and System Safety, 2006. **91**(7): p. 747-755.
23. Bozzano, M. and A. Villafiorita, *Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform*, in proceedings of *22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2003. LNCS-2788, pp. 49-62. Springer-Verlag.
24. Bozzano, M., et al., *ESACS: an Integrated Methodology for Design and Safety Analysis of Complex Systems*, in proceedings of *European Safety and Reliability Conference (ESREL)*, 2003, pp. 237-245. Balkema Publishers.
25. Bretschneider, M., *Lessons Learnt About System Safety Assessment Based on Scade Models (Presentation)*, in *Model-based Safety Assessment (Journées MISSA)*. Toulouse, France, 2010. CISEC / MISSA Project Consortium. Available from: http://cisec.enseeiht.fr/index.php?option=com_content&view=article&id=53:news&catid=37:past-events&Itemid=61 [Last accessed: 1 March 2010]
26. Briones, J.F., M.Á. de Miguel, J.P. Silva, and A. Alonso, *Application of Safety Analyses in Model Driven Development*, in proceedings of *5th IFIP WG 10.2 International Workshop*, 2007. LNCS 4761, pp. 93-104. Springer Berlin / Heidelberg.

27. Caspi, P., D. Pilaud, N. Halbwachs, and J.A. Plaice, *LUSTRE: A Declarative Language for Programming Synchronous Systems*, in proceedings of *ACM Symposium on Principles of Programming Languages (POPL)*, 1987, pp. 178-188. ACM.
28. Cassez, F., C. Pagetti and O. Roux, *A Timed Extension for AltaRica*. Fundamenta Informaticae, 2004. **62**(3-4): p. 291-332.
29. Cavallo, A., *System Safety Assessment Based on Formal Models: Lessons Learnt by Alenia Aeronautica (Presentation)*, in *Model-based Safety Assessment (Journées MISSA)*. Toulouse, France, 2010. CISEC / MISSA Project Consortium. Available from: http://cisec.enseeiht.fr/index.php?option=com_content&view=article&id=53:news&catid=37:past-events&Itemid=61 [Last accessed: 1 March 2010]
30. Cheng, B.M.W., J.H.M. Lee and J.C.K. Wu, *A Constraint-Based Nurse Rostering System Using a Redundant Modeling Approach*, in proceedings of *8th International Conference on Tools with Artificial Intelligence*, 1996, pp. 140-148. IEEE Computer Society.
31. Cimatti, A., et al., *NuSMV2: An OpenSource Tool for Symbolic Model Checking*, in proceedings of *14th International Conference on Computer Aided Verification (CAV)*, 2002. LNCS-2404, pp. 241-268. Springer-Verlag.
32. Clements, P., *Comparing the SEI's Views and Beyond Approach for Documenting Software Architectures with ANSI-IEEE 1471-2000*, 2005. Technical Note: Nr. *CMU/SEI-2005-TN-017*, Software Engineering Institute, Carnegie Mellon University (SEI/CMU): Pittsburgh, PA.
33. Clements, P., et al., *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. 2002, Boston, MA: Addison -Wesley.
34. Conmy, P., *Performing Failure Analysis for IMA as a Separate System*, 2001. DCSC Technical Note: Nr. *DCSC/TN/2000/20*, The University of York.
35. Conmy, P., *Safety Analysis of Computer Resource Management Software* (PhD Thesis), 2006. Dep't of Computer Science, The University of York.
36. Conmy, P. and J.A. McDermid, *High Level Failure Analysis for Integrated Modular Avionics*, in *6th Australian Workshop on Industrial Experience with Safety Critical Systems and Software*. Brisbane, Australia, 2001.
37. Contini, S., G.G.M. Cojazzi and G. Renda, *On the Use of Non-Coherent Fault Trees in Safety and Security Studies*. Reliability Engineering and System Safety, 2008. **93**: p. 1886-1895.
38. Damm, W., B. Josko and T. Peikenkamp, *Contract Based ISO CD 26262 Safety Analysis*, in *2009 SAE World Congress*. Detroit, MI, 2009. SAE International.
39. Dassault Aviation, *Cecilia Workshop (c) OCAS Module: System Design and Analysis (v4.3)*. User Manual, Paris, France, 2007.
40. Dawkins, S.K., et al., *Issues in the Conduct of PSSA*, in *17th International System Safety Conference (ISSC)*. Unionville, VA, 1999. System Safety Society.

41. Döhmen, G., *SPEEDS Methodology – a White Paper*, 2008. Technical Report, SPEEDS Project.
42. Domis, D. and M. Trapp, *Integrating Safety Analyses and Component-Based Design*, in *27th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*. Newcastle upon Tyne, UK, 2008. Springer Verlag.
43. Domis, D. and M. Trapp, *Component-Based Abstraction in Fault Tree Analysis*, in proceedings of *28th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2009. LNCS 5775, pp. 297-310. Springer-Verlag.
44. Dugan, J., S.J. Bavuso and M.A. Boyd, *Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems*. IEEE Transactions on Reliability, 1992. **41**(3): p. 363-377.
45. Eclipse Foundation. *Epsilon Home Page*. Available from: http://www.eclipse.org/gmt/epsilon/ [Last accessed: 2 March 2010]
46. Eclipse Foundation. *Eclipse Modelling Framework web pages*. [Web Page]; Available from: http://www.eclipse.org/modeling/emf/ [Last accessed: 2 March 2010]
47. Eclipse Foundation. *Emfatic Article in Eclipse Wiki*. Available from: http://wiki.eclipse.org/Emfatic [Last accessed: 2 March 2010]
48. Ericson, C., *Fault Tree Analysis - A History*, in *17th International System Safety Conference (ISSC)*. Orlando, FL, 1999. System Safety Society.
49. Ericson, C., *Hazard Analysis Techniques for System Safety*. 2005, Hoboken, NJ: John Wiley & Sons.
50. ESACS Project Consortium. *ESACS Project Website*. Available from: www.esacs.org [Last accessed: 1 March 2010]
51. Esterel Technologies. *SCADE Design Verifier Webpage*. Available from: http://www.esterel-technologies.com/products/scade-suite/design-verifier [Last accessed: 1 March 2010]
52. Esterel Technologies. *SCADE Suite Webpage*. Available from: http://www.esterel-technologies.com/products/scade-suite/ [Last accessed: 1 March 2010]
53. European Aviation Safety Agency, *Paragraph 1309: Equipment, Systems and Installations (CS-25 (Amendment 4))*. 2007, EASA: Cologne, Germany.[Certification Requirements]
54. European Committee for Electrotechnical Standardization, *Railway Applications - Safety Related Electronic Systems for Signalling (CENELEC ENV 50129)*. 1998, CENELEC: Brussels, Belgium.[European Standard]
55. European Organisation for the Safety of Air Navigation (EUROCONTROL), *Air Navigation System Safety Assessment Methodology (SAF.ET1.ST03.1000-MAN-01)*. 2006, EUROCONTROL: Brussels, Belgium.[Standard]
56. Feiler, P. and A. Rugina, *Dependability Modeling with the Architecture Analysis & Design Language (AADL)*, 2007. Nr. *CMU/SEI-2007-TN-043*, Software Engineering Institute, Carnegie Mellon University (SEI/CMU): Pittsburgh, PA.

57. Fenelon, P. and J.A. McDermid, *New Directions in Software Safety: Causal Modelling as an Aid to Integration*, 1992. HISE Technical Report, High Integrity Systems Engineering Group, Dep't of Computer Science, The University of York: York, UK.
58. Fenelon, P. and J.A. McDermid, *An Integrated Toolset for Software Safety Analysis*. The Journal of Systems and Software, 1993. **21**(3): p. 279-290.
59. Ge, X., R. Paige and J.A. McDermid, *Probabilistic Failure Propagation and Transformation Analysis*, in proceedings of *28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, 2009. LNCS 5775, pp. 215-228. Springer Verlag.
60. Giese, H., M. Tichy and D. Schilling, *Compositional Hazard Analysis of UML Components and Deployment Models*, in *23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP)*. Potsdam (Germany), 2004. Springer-Verlag.
61. Griffault, A. and G. Point, *On the Partial Translation of Lustre Programs into the AltaRica Language and Vice Versa*, 2006. Research Report: Nr. *RR-1415-06*, Laboratoire Bordelais de Recherche en Informatique (Université Bordeaux): Bordeaux, France.
62. Grunske, L. and B. Kaiser, *Automatic Generation of Analyzable Failure Propagation Models from Component-Level Failure Annotations*, in *5th International Conference on Quality Software (QSIC)*. Melbourne, Australia, 2005. IEEE Computer Society.
63. Habli, I. and T. Kelly, *Achieving Integrated Process and Product Safety Arguments*, in *15th Safety-critical Systems Symposium (SSS)*. Bristol, UK, 2007. Springer.
64. Harel, D., *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 1987. **8**(3): p. 231-274.
65. Harel, D., et al., *Statemate: a Working Environment for the Development of Complex Reactive Systems*, in proceedings of *10th International Conference on Software Engineering*, 1988, pp. 396-406. IEEE Computer Society.
66. Heimdahl, M.P.E., Y. Choi and M. Whalen, *Deviation Analysis Through Model Checking*, in proceedings of *17th IEEE International Conference on Automated Software Engineering (ASE)*, 2002, pp. 37-46. IEEE Computer Society.
67. Hofmeister, C., R. Nord and D. Soni, *Applied Software Architecture*. Object Technology Series, G. Booch, I. Jacobson, and Rumbaugh series ed. 1999, Reading, MA: Addison Wesley Longman.
68. Hubka, V. and W.E. Eder, *A Scientific Approach to Engineering Design*. Design Studies, 1987. **8**(3): p. 123-137.
69. IBM. *IBM Rational Statemate Webpage*. Available from: http://www-01.ibm.com/software/awdtools/statemate/ [Last accessed: 1 March 2010]
70. Institute of Electrical and Electronic Engineers, *Draft Recommended Practice for Architectural Description (IEEE P1471/D4.1)*. 1998, IEEE: New York, NY.[Draft Standard]

71. International Electrotechnical Commission (IEC), *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA) (IEC 60812)*. 2006, IEC: Geneva, Switzerland.[International Standard]
72. International Electrotechnical Commission (IEC) and I.S.O. (ISO), *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model (ISO/IEC 7498-1)*. 1994, ISO/IEC: Geneva, Switzerland.[International Standard]
73. ISAAC Project Consortium. *ISAAC Project Website*. Available from: http://www.isaac-fp6.org/ [Last accessed: 1 March 2010]
74. Iwu, F., et al., *Compositional Analysis and Verification Approaches to Safety Analysis and Systems Modelling*, 2005. Airbus Dependability Network White Paper (D1.1), Airbus DepNet Consortium: Filton, UK.
75. Johnston, B.D. and R.H. Matthews, *Noncoherent Structure Theory: A Review of its Role in Fault Tree Analysis*, 1983. Nr. *SRD R245*, United Kingdom Atomic Energy Authority - Safety and Reliability Directorate.
76. Joshi, A. and M.P.E. Heimdahl, *Behavioral Fault Modelling for Model-based Safety Analysis*, in proceedings of *10th IEEE High Assurance Systems Engineering Symposium (HASE)*, 2007, pp. 199-208. IEEE Computer Society.
77. Joshi, A., M.P.E. Heimdahl, S.P. Miller, and W.W. Whalen, *Model-Based Safety Analysis*, 2006. Contractor Report: Nr. *NASA/CR-2006-213953*, US National Aeronautics and Space Administration (NASA): Hampton, VA.
78. Josko, B., Q. Ma and A. Metzner, *Designing Embedded Systems Using Heterogeneous Rich Components*, in *18th INCOSE International Symposium*. Utrecht, Netherlands, 2008. International Council on Systems Engineering (INCOSE).
79. Kaiser, B., *A Fault-Tree Semantics to Model Software-Controlled Systems*. Softwaretechnik-Trends, 2003. **23**(3).
80. Kaiser, B. and C. Gramlich, *State-Event-Fault-Trees – A Safety Analysis Model for Software Controlled Systems*, in proceedings of *23rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, 2004. LNCS 3219, pp. 195-209. Springer Berlin / Heidelberg.
81. Kaiser, B., P. Liggesmeyer and O. Mäckel, *A New Component Concept for Fault Trees*, in proceedings of *8th Australian workshop on Safety critical systems and software*, 2003. 33, pp. 37-46. Australian Computer Society.
82. Kehren, C., et al., *Advanced Simulation Capabilities for Multi-Systems with AltaRica*, in *24th International System Safety Conference (ISSC)*. Providence, RI, 2004. System Safety Society.
83. Kelly, T., *Arguing Safety - A Systematic Approach to Managing Safety Cases* (PhD Thesis), 1999. Dep't of Computer Science, The University of York.
84. Kelly, T., *A Systematic Approach to Safety Case Management*, in *SAE 2004 World Congress*. Detroit, MI, 2004. Society of Automotive Engineers.

85. Kletz, T., *HAZOP and HAZAN: Identifying and Assessing Process Industry Hazards*. (4th ed). 1999, Rugby, UK: Institution of Chemical Engineers (IChemE).
86. Kolovos, D., R. Paige and F. Polack, *Eclipse Development Tools for Epsilon*, in *Eclipse Summit Europe, Eclipse Modelling Symposium.* Esslingen (Germany), 2006.
87. Laboratoire Bordelais de Recherche en Informatique. *AltaRica Project Website*. Available from: http://altarica.labri.fr/ [Last accessed: 1 March 2010]
88. Ladkin, P.B. *The Why-Because Analysis Homepage*. Available from: http://www.rvs.uni-bielefeld.de/research/WBA/ [Last accessed: 2 March 2010]
89. Lawrence, B., *A380 Aircraft Safety Process*, in proceedings of *1st IET International Conference on System Safety*, 2006, pp. 95-115 (Keynote Presentation). Institute of Engineering and Technology (IET).
90. Leveson, N., *Safeware: System Safety and Computers*. 1995: Addison-Wesley.
91. Leveson, N., *White Paper on Approaches to Safety Engineering*, 2003. Submission to the Columbia Accident Investigation Board, Massachusetts Institute of Technology: Cambridge, MA.
92. Leveson, N.G., M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, *Requirements Specification for Process Control Systems*. IEEE Transactions on Software Engineering, 1994. **20**(9): p. 684-707.
93. Lewis, B. and P. Feiler, *Multi-Dimensional Model Based Engineering for Performance Critical Computer Systems Using AADL*, in *3rd European Congress on Embedded Real Time Systems (ERTS).* Toulouse, France, 2006.
94. Lisagor, O., *Work Package 4 Development Description Report - Issue A*, 2009. MISSA Project Deliverable: Nr. *D4.10*, University of York & MISSA Project Consortium: York, UK.
95. Lisagor, O. and T. Kelly, *Incremental Safety Assessment: Theory and Practice*, in *26th International System Safety Conference (ISSC).* Vancouver, 2008. System Safety Society.
96. Lisagor, O., J.A. McDermid and D.J. Pumfrey, *Towards a Practicable Process for Automated Safety Analysis*, in *24th International System Safety Conference (ISSC).* Albuquerque, NM, 2006. System Safety Society.
97. Lisagor, O., et al., *Towards Safety Analysis of Highly Integrated Technologically Heterogeneous Systems – A Domain-Based Approach for Modelling System Failure Logic*, in *24th International System Safety Conference (ISSC).* Albuquerque, NM, 2006. System Safety Society.
98. Liu, S. and R. Adams, *Limitations of Formal Methods and an Approach to Improvement*, in proceedings of *2nd Asia Pacific Software Engineering Conference*, 1995, pp. 498-521. IEEE Computer Society.
99. Lutz, R. and R. Woodhouse, *Bi-directional Analysis for Certification of Safety-Critical Software*, in *1st International Software Assurance Certification Conference (ISACC'99).* Chantilly, VA, 1999.

100. Majdara, A. and T. Wakabayashi, *Component-Based Modelling of Systems for Automated Fault Tree Generation.* Reliability Engineering and System Safety, 2009. **94**(6): p. 1076-1086.

101. Malhotra, M. and K.S. Trivedi, *Power-Hierarchy of Dependability-Model Types.* IEEE Transactions on Reliability, 1994. **43**(3): p. 493-502.

102. Mason, P.A.J., *MATrA: Meta-modelling Approach to Traceability for Avionics* (PhD Thesis), 2002. Dep't of Computer Science, University of Newcastle.

103. Mauri, G., *Integrating Safety Analysis Techniques, Supporting Identification of Common Cause Failures* (PhD Thesis), 2000. Dep't of Computer Science, The University of York.

104. Mauri, G., J.A. McDermid and Y. Papadopoulos, *Extension of Hazard and Safety Analysis Techniques to Address Problems of Hierarchical Scale.* IEE Digest, 1998. **98/249**: p. 4.1-4.6.

105. McDermid, J.A. and D.J. Pumfrey, *A Development of Hazard Analysis to aid Software Design*, in proceedings of *9th Conference on Computer Assurance (COMPASS '94)*, 1994, pp. 17-25. IEEE.

106. McDermid, J.A. and D.J. Pumfrey, *Software Safety: Why is there no Consensus?*, in *19th International System Safety Conference (ISSC).* Huntsville, AL, 2001. System Safety Society.

107. McDermid, J.A., M. Nicholson, D.J. Pumfrey, and P. Fenelon, *Experience with the application of HAZOP to computer-based systems*, in proceedings of *10th Conference on Computer Assurance (COMPASS '95)*, 1995, pp. 37-48. IEEE.

108. Mokos, K., et al., *Towards Compositional Safety Analysis via Semantic Representation of Component Failure Behaviour*, in proceedings of *8th Joint Conference on Knowledge-Based Software Engineering*, 2008. 180, pp. 405-414. IOS Press.

109. Motor Industry Software Reliability Association, *Guidelines for Safety Analysis of Vehicle Based Programmable Systems* 2007, MISRA: Nuneaton, UK.[MISRA Guidance]

110. Object Management Group Inc, *Human-Usable Textual Notation (HUTN) Specification* 2004, OMG: Needham, MA.

111. Pagetti, C., F. Cassez and O. Roux, *Hierarchical Modelling and Verification of Timed Systems in Timed AltaRica*, in *1st Workshop on Formal Aspects of Component Software (FACS).* Pisa, Italy, 2003.

112. Pahl, G. and W. Beitz, *Engineering Design: A Systematic Approach*. 1997, London: Springer-Verlag.

113. Palshikar, G.K., *Temporal Fault Trees.* Information and Software Technology, 2002. **44**: p. 137-150.

114. Papadopoulos, Y., *Hierarchically Performed Hazard Origin and Propagation Studies*, in proceedings of *18th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, 1999. LNCS-1698, pp. 139-152. Springer-Verlag.

115. Papadopoulos, Y., *Safety-Directed System Monitoring Using Safety Cases* (PhD Thesis), 2000. Dep't Computer Science, The University of York.

116. Papadopoulos, Y. and M. Maruhn, *Model-Based Synthesis of Fault Trees from Matlab-Simulink Models*, in *International Conference on Dependable Systems and Networks (DSN)*. Goteborg (Sweden), 2001. IEEE Computer Society.

117. Papadopoulos, Y., J.A. McDermid, R. Sasse, and G. Heiner, *Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in Conditions of Failure*. Journal of Reliability Engineering and System Safety, 2001. **71**(3): p. 229-247.

118. Parnas, D.L., *On a 'Buzzword': Hierarchical Structure*, in proceedings of *6th IFIP Congress on Information Processing*, 1974, pp. 336-339. North-Holland.

119. Peikenkamp, T., et al., *Towards a Unified Model-Based Safety Assessment*, in proceedings of *25th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2006. LNCS-4166, pp. 275-288. Springer-Verlag.

120. Pentti, H. and H. Atte, *Failure Modes and Effects Analysis of Software-Based Automation Systems*, 2002. STUK Technical Report: Nr. *STUK-YTO-TR-190*, Radiation and Nuclear Safety Authority: Helsinki, Finland.

121. Pidcock, W. *What are the Differences Between a Vocabulary, a Taxonomy, a Thesaurus, an Ontology, and a Meta-model?* ; On-line Article. Available from: http://infogrid.org/wiki/Reference/PidcockArticle [Last accessed: 1 March 2010]

122. Point, G. and A. Rauzy, *AltaRica: Constraint Automata as a Description Language*. Journal Européen des Systèmes Automatisés, 1999. **33**(8-9): p. 1033-1052.

123. Pumfrey, D.J., *The Principled Design of Computer System Safety Analyses* (D.Phil Thesis), 1999. Dep't Computer Science, The University of York.

124. Radio Technical Committee for Aeronautics (RTCA), *Software Considerations in Airborne Systems and Equipment Certification (DO-178b (also ED-12B))*. 1994, RTCA / EUROCAE: Washington, DC / Malakoff, France.[Means of Compliance Guidance]

125. Rauzy, A., *The AltaRica DataFlow Language: Syntax*, 2002. Technical Note: Nr. *AltaRica/TN02-1*, ARBoost Technologies: Marseilles, France.

126. Rauzy, A., *Mode Automata and Their Compilation into Fault Trees*. Reliability Engineering and System Safety, 2002. **78**(1): p. 1-12.

127. Reese, J.D., *Software Deviation Analysis* (PhD Thesis), 1996. University of California, Irvine.

128. Reese, J.D. and N.G. Leveson, *Software Deviation Analysis*, in proceedings of *19th International Conference on Software Engineering*, 1997, pp. 250-260. ACM.

129. Rossi, F., P. van Been and T. Walsh, *Handbook of Constraint Programming*. Foundations of Artificial Intelligence, J. Hendler, H. Kitano, and B. Nebel series ed. 2006, Oxford, UK: Elsevier.

130. Sagan, S., *The Limits of Safety: Organizations, Accidents and Nuclear Weapons*. Princeton Studies in International History and Politics, J.L. Gaddis, J.L. Snyder, and R.H. Ullman series ed. 1993, Princeton, NJ: Princeton University Press.

131. Sartor, V., *Plateforme CECILIA AltaRica-OCAS: L'Atelier de Sûreté de Fonctionnement de Dassault Aviation (Presentation)*, in *2èmes Journées AltaRica (2003 AltaRica Workshop)*. Bordeaux, France, 2003. Available from: http://altarica.labri.fr/pub/slides/workshop-20031022/jalta_sartor.ppt [Last accessed: 1 March 2010]

132. Seguin, C., P. Bieber, C. Castel, and C. Kehren, *Formal Assessment Techniques for Embedded Safety Critical Systems*, in *2nd National Workshop on Control Architectures of Robots (CAR'2007)*. Paris, France, 2007. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.109.2586 [Last accessed: 1 March 2010]

133. Selic, B., *The Pragmatics of Model-Driven Development*. IEEE Software, 2003. **20**(5): p. 19-25.

134. Shaikh, T., *Assessment of Safety Critical Systems: New Model Based Safety Analysis Technique* (Industrial Project Report), 2009. Bristol Institute of Technology, University of the West of England.

135. Sharvia, S. and Y. Papadopoulos, *Non-coherent Modelling in Compositional Fault Tree Analysis*, in proceedings of *17th IFAC World Congress*, 2008, pp. 4138-4143. The International Federation of Automatic Control.

136. Society of Automotive Engineers, *Architecture Analysis and Design Language (AADL), Annex E: Error Model Annex (Annex Volume 1) (AS5506/1)*. 2006, SAE Aerospace: Warrendale, PA.[Aerospace Standard]

137. Society of Automotive Engineers, *Potential Failure Mode and Effects Analysis in Design (Design FMEA), Potential Failure Mode and Effects Analysis in Manufacturing and Assembly Processes (Process FMEA) (SAE J1739)*. 2009, SAE International: Warrendale, PA.[Standard]

138. Society of Automotive Engineers, *Architecture Analysis and Design Language (AADL) (AS5506A)*. 2009, SAE Aerospace: Warrendale, PA.[Aerospace Standard]

139. Society of Automotive Engineers (SAE), *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (ARP4761)*. 1996, SAE International: Warrendale, PA.[Aerospace Recommended Practice]

140. Society of Automotive Engineers (SAE), *Certification Considerations for Highly-Integrated or Complex Aircraft Systems (ARP 4754 / ED-79)*. 1996, SAE International / EUROCAE: Warrendale, PA.[Aerospace Recommended Practice]

141. SPEEDS Consortium / Airbus Deutschland GmbH. *SPEEDS Project Website*. Available from: http://www.speeds.eu.com [Last accessed: 13 March 2010]

142. Steinberg, D., F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modelling Framework*. (2nd ed). The Eclipse Series, E. Gamma, L. Nckman, and J. Wiegand series ed. 2008: Addison-Wesley.

143. The Chemical Industry Safety and Health Council of the Chemical Industries Association Ltd., *A Guide to Hazard and Operability Studies* 1977, CISHEC: London, UK.

144. The MathWorks Inc. *Stateflow 7.5: Design and Simulate State Machines and Control logic*. Available from: http://www.mathworks.co.uk/products/stateflow/ [Last accessed: 1 March 2010]

145. The MathWorks Inc. *Simulink: Simulation and Model-Based Design*. Available from: http://www.mathworks.co.uk/products/simulink/ [Last accessed: 1 March 2010]

146. UK Ministry of Defence (MoD), *HAZOP Studies on Systems Containing Programmable Electronics (DefStan 00-58)*. 1996, MoD: Glasgow, UK.[Interim Defence Standard (Cancelled)]

147. UK Ministry of Defence (MoD), *Safety Management Requirements for Defence Systems; Part 1: Requirements (DefStan 00-56 / 1; Issue 2)*. 1996, MoD: Glasgow, UK.[Defence Standard (Superseded)]

148. UK Ministry of Defence (MoD), *Requirements for Safety Related Software in Defence Equipment; Part 1: Requirements (DefStan 00-55 / 1; Issue 2)*. 1997, MoD: Glasgow, UK.[Defence Standard (Superseded)]

149. UK Ministry of Defence (MoD), *Safety Management Requirements for Defence Systems; Part 1: Requirements (DefStan 00-56 / 1; Issue 4)*. 2007, MoD: Glasgow, UK.[Defence Standard]

150. UK Ministry of Defence (MoD), *Safety Management Requirements for Defence Systems; Part 2: Guidance on Establishing a Means of Complying with Part 1 (DefStan 00-56 / 2; Issue 4)*. 2007, MoD: Glasgow, UK.[Defence Standard]

151. US Air Force Safety Agency, *Airforce System Safety Handbook* 2000: Kirtland AFB, NM.

152. US Department of Defense, *Procedures for Performing, a Failure Mode, Effects, and Criticality Analysis (MIL-STD-1629A)*. 1980: Washington, DC.

153. US Department of Defense (DoD), *System Safety Program Requirements (MIL-STD-882C)*. 1993, AFMC/SE: Wright Patterson AFB, OH.[Military Standard (Superseded)]

154. US Department of Defense (DoD), *Standard Practice for System Safety (MIL-STD-882D)*. 2000, AFMC/SES: Wright Patterson AFB, OH.[Military Standard]

155. US Nuclear Regulatory Commission, *PRA Procedures Guide: A Guide to the Performance of Probabilistic Risk Assessments for Nuclear Power Plants (NUREG/CR-2300)*. 1983, NUREG: Washington, DC.[Guide]

156. van der Meulen, M., *Definitions for Hardware and Software Safety Engineers*. 2000, London: Springer-Verlag.

157. Vesely, W.E., F.F. Goldberg, N.H. Roberts, and D.F. Haasl, *Fault Tree Handbook (NUREG-0492)*. 1981, US Nuclear Regulatory Commission,

158. Vesely, W.E., et al., *Fault Tree Handbook with Aerospace Applications*, 2002. NASA Office of Safety and Mission Assurance.

159. Vestal, S., *MetaH Support for Real-Time Multi-Processor Avionics*, in proceedings of *Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS / OORTS)*, 1997, pp. 11-21. IEEE Computer Society.

160. Villemeur, A., *Reliability, Availability, Maintainability and Safety Assessment: Methods and Techniques*. Vol. 1. 1991, Chichester: John Wiley & Sons.

161. Walker, M. and Y. Papadopoulos, *Synthesis and Analysis of Temporal Fault Trees with PANDORA: The Time of Priority AND Gates*, in proceedings of *International Conference on Hybrid Systems and Applications*, 2006. Vol. 2, pp. 368-382. Elsevier Science.

162. Walker, M., L. Bottaci and Y. Papadopoulos, *Compositional Temporal Fault Tree Analysis*, in proceedings of *26th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2007. LNCS-4680, pp. 106-119. Springer Verlag.

163. Wallace, M., *Modular Architectural Representation and Analysis of Fault Propagation and Transformation*, in proceedings of *2nd International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2005)*, 2005, pp. 53-71. Elsevier.

164. Whewell, W., *Of Analytical Mathematics as an Educational Study (Ch.1, Sect.5)*, in *Of a Liberal Education in General; and With Particular Reference to the Leading Studies of the University of Cambridge*. 1845, JOHN W. PARKER: London. p. 38-62.

165. Wu, W. and T. Kelly, *Failure Modelling in Software Architecture Design for Safety*, in proceedings of *Workshop on Architecting Dependable Systems (WADS'05)*, 2005, pp. 1-7. ACM.

166. Wu, W. and T. Kelly, *Combining Bayesian Belief Networks and the Goal Structuring Notation to Support Architectural Reasoning About Safety* in *26th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*. Nuremberg, Germany, 2007. Springer-Verlag.