



## **Model-Connected Safety Cases**

being a Thesis submitted in fulfilment of the  
requirements for the degree of  
Doctor of Philosophy  
in the University of Hull

by

Athanasios Retouniotis BSc., MSc.

October 2020

*For my family...*

## Abstract

---

Regulatory authorities require justification that safety-critical systems exhibit acceptable levels of safety. Safety cases are traditionally documents which allow the exchange of information between stakeholders and communicate the rationale of how safety is achieved via a clear, convincing and comprehensive argument and its supporting evidence. In the automotive and aviation industries, safety cases have a critical role in the certification process and their maintenance is required throughout a system's lifecycle. Safety-case-based certification is typically handled manually and the increase in scale and complexity of modern systems renders it impractical and error prone.

Several contemporary safety standards have adopted a safety-related framework that revolves around a concept of generic safety requirements, known as Safety Integrity Levels (SILs). Following these guidelines, safety can be justified through satisfaction of SILs. Careful examination of these standards suggests that despite the noticeable differences, there are converging aspects. This thesis elicits the common elements found in safety standards and defines a pattern for the development of safety cases for cross-sector application. It also establishes a metamodel that connects parts of the safety case with the target system architecture and model-based safety analysis methods. This enables the semi-automatic construction and maintenance of safety arguments that help mitigate problems related to manual approaches. Specifically, the proposed metamodel incorporates system modelling, failure information, model-based safety analysis and optimisation techniques to allocate requirements in the form of SILs. The system architecture and the allocated requirements along with a user-defined safety argument pattern, which describes the target argument structure, enable the instantiation algorithm to automatically generate the corresponding safety argument. The idea behind model-connected safety cases stemmed from a critical literature review on safety standards and practices related to safety cases. The thesis presents the method, and implemented framework, in detail and showcases the different phases and outcomes via a simple example. It then applies the method on a case study based on the Boeing 787's brake system and evaluates the resulting argument against certain criteria, such as scalability. Finally, contributions compared to traditional approaches are laid out.

## Acknowledgements

---

In this section, I would like to express my gratitude to all of my allies who offered me their support during these long years towards the completion of this thesis. I would also like to share some of my thoughts.

I always thought of life as a rhombus-shaped diagram with interconnected nodes. Each node may lead directly to new opportunities, a dead end or may just offer an interconnection to other nodes. Starting from the top of the rhombus, as infants we do not have that many choices; however, while growing up, we meet more and more people who act as nodes. It is up to us to speculate the path each of these nodes might lead us to. Based on our choices, it is possible to discover nodes with a short route towards achieving our goals or nodes that broaden our horizon and help us find new directions in life. I feel I have been lucky enough to have chosen wisely so far.

During my second year as a BSc student, I found myself fascinated with the idea of doing a PhD for two major reasons. First, for the possibility of one day working as an academic, able to interact with students and help them achieve their dreams by transferring my knowledge and experience. Second, for conducting research with the potential to contribute something to Computer Science, which I adore. When I came to Hull for my MSc, I met Prof. Yiannis Papadopoulos, who was assigned as my advisor by the University. At the time, we discussed about my goals and he introduced me to the world of academic research. Yiannis also taught me about the importance of conducting research that may have immediate impact on the society and guided me through his work on dependability engineering. After my graduation, Yiannis suggested that I apply for a position in his group, as a PhD student. I cannot find enough words to thank him for everything he has done for me ever since. He offered me his guidance throughout my PhD for both my research and personal matters. Every meeting we had felt like a friendly conversation which cleared my mind, gave me confidence to continue this PhD and boosted my morale. Yiannis was one of the major connecting nodes that opened up many paths in my life. I will always be grateful for his support and patience; and unequivocally, I will always be there for him in case of need.

Next, I would really like to thank Dr. Martin Walker, Dr. David Parker and Dr. Septavera Sharvia for their support. Despite their strict schedule, they offered me their technical expertise, via workshops and meetings, on various tools and techniques that I utilised during my PhD. They have also generously offered their feedback on earlier drafts of this thesis. I consider myself lucky that I was able to work with them.

I would also like to mention my father, mother and sister for their moral support. They really stood there for me and sent various invigorating messages over the past few semesters and especially during the pandemic.

Further, I would like to thank my dear friend Yanos. Through good and dark moments, he was there with a smile on his face and willing to mitigate any of my problems. Yanos was a PhD student himself until a couple of years ago; therefore, he was the only person of my inner circle that was able to fully understand me in the matter. I will never forget our all-nighters at the university, both researching for our dissertations whilst enjoying each other's company.

Last but not least, I would like to express my eternal gratitude to my partner, Pia, for everything she has done for me all these years. I cannot even imagine how hard it can be to live with a student for so many years. Her patience, cheer and our insightful discussions have made me a better person.

My deepest appreciation and apologies to everyone who managed to help me in their own way, yet not included in this section.

# Table of Contents

---

<b>Abstract.....</b>	<b>3</b>
<b>Acknowledgements .....</b>	<b>4</b>
<b>Table of Figures.....</b>	<b>11</b>
<b>List of Tables .....</b>	<b>13</b>
<b>List of Abbreviations .....</b>	<b>14</b>
<b>Chapter 1 Introduction.....</b>	<b>16</b>
1.1 Thesis Outline .....	16
1.2 The essentials of Safety and Reliability .....	16
1.3 Evolution of Safety Assurance.....	19
1.4 Significant Changes in Safety Cases.....	21
1.5 Maintenance Issues in Safety Cases.....	22
1.6 Safety Requirements Described in Safety Standards .....	25
1.7 Model-Based Development and Analysis.....	28
1.8 Research Motivation .....	29
1.9 Research Hypothesis .....	29
1.10 Research Objectives .....	30
1.11 Thesis Structure .....	30
1.12 Published Material .....	32
<b>Chapter 2 Background .....</b>	<b>34</b>
2.1 Common Terminology in Safety Assurance .....	34
2.2 Safety-related Processes in System Development .....	39
2.2.1 System Modelling and Specifications .....	39
2.2.2 Functional Analysis of Systems .....	40
2.2.3 Functional Hazard Analysis .....	42
2.2.4 Functional Failure Analysis .....	43
2.2.5 Preliminary Safety Assessment.....	44
2.2.6 Common Cause Analysis .....	45
2.3 Safety Analysis Techniques .....	46
2.3.1 Fault Tree Analysis .....	47
2.3.2 Failure Modes and Effects Analysis .....	51
2.4 Other Safety Analysis Methods .....	52
2.4.1 Markov Modelling and Analysis.....	53

2.4.2	Failure Propagation and Transformation Notation .....	56
2.4.3	Fault Propagation and Transformation Calculus.....	58
2.4.4	Dependence Diagrams .....	59
2.4.5	Dynamic Fault Trees .....	60
2.5	Model-Based Safety Analysis .....	61
2.6	Compositional Safety Analysis .....	62
2.6.1	Component Fault Trees .....	62
2.6.2	State Event Fault Trees .....	64
2.6.3	Hierarchically Performed Hazard Origin and Propagation Studies .....	65
2.6.4	Failure Propagation and Transformation Analysis .....	67
2.7	Behavioural Safety Analysis .....	69
2.7.1	FSAP/NuSMV-SA and xSAP .....	70
2.7.2	AltaRica .....	72
2.8	Other Notable Tools.....	73
2.8.1	Galileo.....	73
2.8.2	DFTCalc.....	74
2.8.3	FaultTree+ .....	74
2.9	Safety Cases .....	75
2.9.1	An Introduction to Safety Cases .....	75
2.9.2	Definition of Safety Cases .....	76
2.9.3	Structure of Safety Cases .....	78
2.9.4	Benefits and Limitations .....	79
2.10	Safety Argument Notations.....	80
2.10.1	Goal Structuring Notation .....	82
2.10.2	Claims Arguments Evidence Notation.....	86
2.11	Methods for Managing Safety Cases .....	87
2.11.1	Argument Patterns.....	88
2.11.2	Modularisation of Safety Cases .....	91
2.12	Safety Integrity Levels .....	92
2.13	Modelling in Safety-Critical Systems .....	94
2.13.1	SACM Metamodel .....	94
2.13.2	SysML.....	98
2.13.3	AADL .....	98
2.13.4	EAST-ADL .....	100

2.14	Summary .....	101
<b>Chapter 3 Safety Assessment in Automotive and Aviation .....</b>		<b>103</b>
3.1	Functional Safety .....	103
3.2	Safety Assessment in Modern Safety Standards .....	104
3.3	Regulations and Certification.....	107
3.3.1	Certification and Regulations for Civil Aviation .....	107
3.3.2	Certification and Regulations for Automotive .....	109
3.4	Development and Safety Lifecycles in Modern Safety Standards .....	110
3.4.1	The Safety Lifecycle in Civil Aviation .....	111
3.4.2	The Safety Lifecycle in Automotive .....	115
3.5	Safety Cases in Automotive and Civil Aviation .....	118
3.6	Safety Integrity Levels in Aviation and Automotive .....	121
3.6.1	SILs in the Civil Aviation .....	121
3.6.2	SILs in Automotive .....	123
3.6.3	A Decomposition Example with DALs .....	126
3.6.4	SIL Allocation as an Optimization Problem .....	129
3.7	Optimisation Methods.....	129
3.7.1	Metaheuristics .....	130
3.7.2	Exact Methods .....	132
3.8	Automatic Allocation of DALs: An Extension to HiP-HOPS .....	133
3.9	Summary .....	136
<b>Chapter 4 Connecting Safety Cases to System Models.....</b>		<b>138</b>
4.1	Complications in Maintaining Safety Cases .....	138
4.2	From Safety Standards to a Pattern for Safety Assurance .....	140
4.3	The Pattern and Automation Infrastructure.....	142
4.4	Method Overview .....	144
4.5	Method Demonstration .....	146
4.5.1	Functional Design .....	146
4.5.2	Hazard Analysis and Requirement Allocation .....	147
4.5.3	Designing the System Architecture.....	148
4.5.4	Failure Behaviour Annotation.....	149
4.5.5	System Safety Analysis.....	151
4.5.6	SILs Decomposition and Allocation .....	152
4.5.7	Creation of Argument Pattern.....	152



4.5.8 Argument Structure Generation .....	153
4.6 Comparison with Relevant Work.....	156
4.6.1 SACM .....	156
4.6.2 D-Case Editor.....	158
4.6.3 ACedit .....	160
4.6.4 ACME .....	161
4.6.5 Weaving Model.....	161
4.6.6 Reusable Safety Case Fragments from Compositional Safety Analysis.....	163
4.6.7 Automated Generation of Safety Cases for Modular Software Product Lines .....	165
4.6.8 Generation of Model-Based Safety Arguments from Automatically Allocated SILs....	165
<b>Chapter 5 Design of Model-Connected Safety Cases.....</b>	<b>167</b>
5.1 The Metamodel .....	167
5.2 The Model Connecting Storage Unit .....	171
5.3 Tool Framework.....	175
5.3.1 The Implementation .....	175
5.3.2 The Design .....	175
5.4 Pattern Generation and Processing .....	184
5.5 Pattern Instantiation Algorithm.....	187
5.6 Summary .....	191
<b>Chapter 6 Case Study .....</b>	<b>192</b>
6.1 A brief history of brake technologies.....	192
6.1.1 Operation Part .....	193
6.1.2 Transmission/Actuating Part.....	193
6.1.3 Brake Units .....	196
6.2 Auxiliary Braking Mechanisms .....	197
6.2.1 Spoilers .....	197
6.2.2 Reverse Thrust .....	198
6.3 Additional Functions.....	199
6.3.1 Anti-skid Protection .....	199
6.3.2 Temperature Indication and Fuse Plugs.....	199
6.3.3 Autobrake.....	200
6.4 Boeing 787 Dreamliner - Brake System .....	200
6.5 Function Modelling.....	205
6.6 Hazard Identification and Analysis.....	206

6.7	Modelling of System Architecture .....	207
6.8	Failure Analysis and Annotation.....	210
6.9	DAL Allocation and Decomposition .....	213
6.10	Pattern Preparation.....	214
6.11	Argument Structure.....	218
6.12	Handling Modification.....	222
6.13	Method Evaluation and Framework Testing.....	226
6.14	Summary .....	230
<b>Chapter 7</b>	<b>Conclusions.....</b>	<b>231</b>
7.1	Synopsis .....	231
7.2	Contributions.....	234
7.3	Limitations and Future Work.....	235
<b>References</b> .....		<b>239</b>
<b>Appendix I:</b>	<b>Images from the Implemented Tool.....</b>	<b>252</b>
<b>Appendix II:</b>	<b>Metrics Implementation.....</b>	<b>254</b>

## Table of Figures

Figure 2.1: The Bathtub Curve (Wilkins, 2002) .....	36
Figure 2.2 Functional Analysis flow process .....	41
Figure 2.3: Abstract Example of FHA on aircraft autopilot system .....	43
Figure 2.4: Abstract Example of FFA on a car brake function inspired by (Johannessen et al., 2001:511) .....	44
Figure 2.5: Zone map of a small aircraft (Linzey, 2006:9) .....	46
Figure 2.6: The architecture of a simple system .....	50
Figure 2.7: The resulting Fault Tree .....	51
Figure 2.8: A simple MA model .....	53
Figure 2.9: Markov model of an abstract safety-related system .....	54
Figure 2.10: An example of FPTN diagram (Fenelon & McDermid, 1992:24) .....	57
Figure 2.11: FPTC Symbols as summarised in (Wallace, 2005:5) .....	58
Figure 2.12: Example of a simple DD structure .....	59
Figure 2.13: The graphical representation of DTF gates (Faulin et al., 2010:42) .....	61
Figure 2.14: Differences in structure between FTA and CFT (Kaiser, 2003:5) .....	63
Figure 2.15: Basic elements of SEFTs (Kaiser & Gramlich, 2004:9) .....	65
Figure 2.16: Basic workflow in HiP-HOPS .....	66
Figure 2.17: Example of a generic processor (Ge et al., 2010:4) .....	68
Figure 2.18: An example of probabilistic data inclusion in FPTA .....	69
Figure 2.19 A switch representation in AltaRica (Point & Rauzy, 1999:4) .....	73
Figure 2.20: A fault within FaultTree+ (FTDSolutions, 2019) .....	75
Figure 2.21: Basic GSN Elements (Wei et al., 2019:5) .....	83
Figure 2.22: Example of an argument in GSN (Joba, 2015) .....	84
Figure 2.23: Strategy choice example .....	86
Figure 2.24: A simple structure in CAE notation .....	87
Figure 2.25 A Safety Argument Pattern for Software contribution (Hawkins & Kelly, 2013: 12) ..	89
Figure 2.26: Example diagrams for Element abstraction and Optionality extensions (Kelly, 1998:170; ACWG, 2018:26) .....	90
Figure 2.27: The Away Goal and the Module reference elements (ACWG, 2018:29) .....	91
Figure 2.28 Replacement of justification by away goals .....	92
Figure 2.29: Safety lifecycle in IEC 61508 (Bell, 2014:10) .....	93
Figure 2.30: The SACM's basic components (Wei et al., 2019:7) .....	95
Figure 2.31: The SACM's AssuranceCasePackage Class Diagram (OMG, 2019:29) .....	97
Figure 2.32: A Summary of the AADL Elements (Feiler et al., 2006:253) .....	99
Figure 2.33: The basic EAST-ADL model (EAST-ADL,2017) .....	101
Figure 3.1: Aircraft system guidelines relationship diagram (SAE, 2010:6) .....	109
Figure 3.2 The V-model of system development (Kelly, 1998:67) .....	111
Figure 3.3: ARP Development lifecycle (SAE, 2010:20) .....	111
Figure 3.4: Development lifecycle under the V-model (Kelly, 1998:67) .....	113
Figure 3.5: Simplified Version of ISO 26262 lifecycle (ISO, 2011:42) .....	115
Figure 3.6: The V-model adaptation in ISO 26262 (ISO,2011:107) .....	116
Figure 3.7: Diagram of System Development (ISO, 2011:113) .....	117
Figure 3.8: Versions of the Safety Case mapped on 'V-model' .....	119
Figure 3.9: ASIL decomposition diagram (ISO, 2011:378) .....	126

Figure 3.10: Abstract system model .....	127
Figure 3.11: Optimisation methods (Talbi, 2009:18).....	130
Figure 4.1: Transition from standards to the pattern and metamodels.....	141
Figure 4.2: Representation of the pattern.....	143
Figure 4.3: Method Overview .....	145
Figure 4.4: The control flow of the example system .....	149
Figure 4.5: Failure Propagation of brake system .....	150
Figure 4.6: Failure behaviour of the target system via a fault tree structure.....	151
Figure 4.7: Minimal cut sets produced for the example.....	152
Figure 4.8: Pattern for the example.....	153
Figure 4.9: Produced argument from the abstract example .....	155
Figure 4.10: V-model combined with COTS-driven development (Sljivo et al., 2016:4).....	164
Figure 5.1 (Left) Simplification of core HiP-HOPS metamodel and (right) illustration of system model design strategies .....	169
Figure 5.2: Overview of metamodel relationships .....	170
Figure 5.3 Example of the Composite mechanism .....	172
Figure 5.4: Core elements of the MCSU storage structure .....	173
Figure 5.5 Tool high-level classes .....	176
Figure 5.6 The ‘ModelEditor’ design .....	179
Figure 5.7 The ArgumentEditor design .....	180
Figure 5.8 The PatternEditor design .....	182
Figure 5.9 The FTViewer design .....	183
Figure 5.10: Abstract example of a pattern in the XML-based syntax .....	184
Figure 5.11: Graphical representation of the pattern example.....	186
Figure 5.12: Resulting Argument Structure.....	186
Figure 6.1 Wing Spoilers .....	197
Figure 6.2 Reverse Thrust (Aerospacweb.org, 2018) .....	198
Figure 6.3 Generic Segmented Rotor-Disc Brake (Academic, 2020).....	201
Figure 6.4 Pedal Motion Command.....	202
Figure 6.5 Base Architecture of 787 Brake System(Maré, 2017:187).....	203
Figure 6.6 High-level Aircraft Functions.....	205
Figure 6.7 Wheel Brake System Model .....	207
Figure 6.8 Abstract BSCU model in detail .....	209
Figure 6.9 Overall aircraft failure behaviour .....	210
Figure 6.10 Analysis Loss of Wheel Brake System.....	212
Figure 6.11 DAL allocation for Wheel Brake.....	214
Figure 6.12 Pattern Overview .....	215
Figure 6.13 Function Validation Claim .....	216
Figure 6.14 DAL Validation Argument Overview .....	217
Figure 6.15 Top-level Argument - Aircraft safety due to compliance with safety standards .....	219
Figure 6.16 Left BSCU’s supporting systems .....	221
Figure 6.17 Altered Argument.....	224
Figure 6.18 Altered Pattern Overview .....	225
Figure 1: Example model from Chapter 4 .....	252
Figure 2: Pattern from the example in Chapter 4.....	252
Figure 3: Argument structure from example in Chapter 4.....	253

## List of Tables

---

Table 2.1 A definition of safety from various sources .....	37
Table 2.2: Failures and Immediate effects .....	50
Table 2.3: FMEA table for a generic item (E.P.Plans, 2006) .....	52
Table 2.4 Previous example in tabular form .....	54
Table 3.1 Example of methods for correct implementation of hardware-software level (ISO, 2011: 127) .....	118
Table 3.2: DAL per severity classification (RTCA, 2011:13) .....	122
Table 3.3 Classes of Severity (ISO, 2011:83).....	124
Table 3.4 Classes of probability of exposure based on operational situation (ISO, 2011: 83) .....	124
Table 3.5: Classes of controllability (ISO, 2011:84) .....	124
Table 3.6: ASIL determination table (ISO, 2011:84) .....	125
Table 3.7: DAL Decomposition Options for Components .....	128
Table 3.8: Indicative cost catalogue for DALs .....	128
Table 4.1 Hazard Analysis Information .....	148
Table 6.1 Ground Deceleration FHA page 1 .....	206
Table 6.2 Ground Deceleration FHA page 2 .....	206
Table 6.3 Framework Execution Time .....	229

## List of Abbreviations

Acronym	Description
AADL	Architecture Analysis & Design Language
AD	Airworthiness Directive
ACME	Assurance Case Modelling Environment
ALARP	As Low As Reasonable Practicable
ARM	Argumentation Metamodel
ARP	Aerospace Recommended Practice
BBN	Bayesian Belief Network
BSA	Behavioural Safety Analysis
CAE	Claims Arguments Evidence
CCA	Common Cause Analysis
CEG	Cause Effect Graph
CFT	Component Fault Tree
CSA	Compositional Safety Analysis
DALs	Development Assurance Levels
DDs	Dependence Diagrams
DFT	Dynamic Fault Tree
DIFTree	Dynamic Innovative Fault Tree
EAST-ADL	Electronic Architecture and Software Technology – Architecture Description Language
ESACS	Enhanced Safety Assessment for Complex Systems
ESM	Extended System Model
FAA	Federal Aviation Administration
FBK	Fondazione Bruno Kessler
FF	Functional Failure
FFA	Functional Failure Analysis
FFS	Functional Failure Set
FHA	Functional Hazard Analysis
FLAR2SAF	Failure Logic Analysis Results to Safety case Argument Fragments
FMEA	Failure Modes and Effects Analysis

FPTA	Failure Propagation and Transformation Analysis
FPTC	Failure Propagation and Transformation Calculus
FPTN	Failure Propagation and Transformation Notation
FSAP	Formal Safety Analysis Platform
FSR	Functional Safety Requirements
FTA	Fault Tree Analysis
GSN	Goal Structuring Notation
GUI	Graphical User Interface
HiP-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies
HTML	Hyper Text Markup Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standardisation Organisation
MA	Markov Analysis
MBSA	Model-Based Safety Analysis
MCSU	Model Connecting Storage Unit
NuSMV	New Symbolic Model Checker
OMG	Object Management Group
PASA	Preliminary Aircraft Safety Assessment
PSSA	Preliminary System Safety Assessment
QM	Quality Management (ISO 26262 term)
SACM	Structured Assurance Case Metamodel
SAE	Society of Automotive Engineers
SAEM	Structured Assurance Evidence Metamodel
SAM	Safety Argument Manager
SEFT	State Event Fault Tree
SG	Safety Goal (ISO 26262 term)
SIL	Safety Integrity Level
SPLs	Safety Product Lines
SysML	Systems Modelling Language
UML	Unified Modelling Language

# Chapter 1 Introduction

---

## 1.1 Thesis Outline

Safety-critical systems need to be certified before they can be put in operation. Certification provides confidence that a given system can operate reliably while satisfying the safety requirements set by industry standards. Assessment activities, evidence and the final assurance argument that rationally communicates how safety is achieved, can be expressed in a safety case. Construction and maintenance of safety cases remains a tedious procedure and manual approaches can be impractical for modern complex systems. Additionally, current practice requires a detailed architecture before assessing the system for safety and proceeding with the identification of mitigation means. As a result, safety cases fail to follow the evolutionary development cycle and addressing safety is often critically delayed until the later stages of development. Realising a hazard is omitted may lead to system redesign, with rapidly increasing development cost when undertaken during the implementation or production stages. To safeguard the public against accidents while also managing these costs, it is important to improve our procedures. This thesis proposes a method that is compatible with modern standards and links the safety case with the system architecture in order to integrate safety assessment activities during the early development stages and facilitate the production and maintenance of safety arguments.

## 1.2 The essentials of Safety and Reliability

Systems can be viewed as collections of components, each having a singular purpose, but with their parent system delivering more than the sum of the constituent components' functionality. Engineered systems have shaped contemporary society, permeating virtually every aspect of it. From systems in the energy production domain, governing the production and distribution of power, to systems controlling the behaviour of individual vehicles, humankind has grown dependent on this infrastructure. Governments and regulatory bodies do not neglect this reliance on technology, and to prevent significant societal danger, they have classified systems, whose failure to operate can be



catastrophic, as **safety-critical systems**. Examples of such systems can be found in the automotive, aerospace, nuclear, healthcare and railway transportation industries. Based on the system type, scale and phase of operation, the potential consequences of a failure differ and may involve significant property damage, environmental damage or loss of life.

To rationalise the repeated utilisation of a system, engineers need to examine whether it is sufficiently reliable or not. **Reliability** refers to the ability of a system to perform its intended function under certain conditions and failing to do so may lead to unpredictable behaviour; thus, making it unreliable (SAE, 2010:13). Reliability engineering is the discipline that studies the behaviour of systems and aims to improve their reliability. Systems are often designed with a minimum level of reliability, which essentially sets an acceptable probability for failure occurrences, and to accomplish this goal potential causes of failure need to be decreased. Therefore, to design a reliable system, it is crucial to understand how it interacts with its components or other integrated systems as well as its environment, and carefully examine under what circumstances the system might fail. Then, it is important to estimate the consequences of each failure and apply countermeasures accordingly. In the past, systems were examined for reliability only at later stages of development or even during their deployed lifetime, but nowadays it is common to consider and analyse system reliability during the early stages of development. This information allows the engineers to identify any flaws in the design and change the system appropriately with decreased costs compared to later stages of development.

During a system's normal operation, any condition that may solely or partly lead to an undesirable event is specified as a **hazard** and is required to be addressed. Examples of hazards in the aforementioned industries include an aircraft crash due to the failure of the flight control system during flight, or a patient's loss of life during surgery as a result of equipment failure. Hazards may vary significantly, and appropriate management is determined based on the following factors: a) the probability of occurrence b) the associated impact of consequences and c) the availability or access to mitigation measures during operation. The severity of consequences and the likelihood of occurrence combined allow the assessment of the hazard's **risk** (IEEE, 1994). Changes to system

design and dependencies might affect the risk, and consequently mitigate the hazard to an acceptable level, by reducing its probability of occurrence or reducing the impact. However, the identification of hazards and their associated risk for each system function, especially at early design stages, is challenging due to the complexity of modern systems and lack of contextual data. For instance, a system with multiple constituent subsystems, often originated from third-party suppliers with their own complex architectures, responsible for just one function still requires a detailed analysis in order to design mitigation means for all the hazards associated with that function. That being said, with enough information about the target system and its architecture as well as sufficient experience from the engineer, it is possible to examine the system for hazards even during the design phase.

Once identifying the hazards, it is important to analyse the system architecture and understand its failure behaviour. This type of analysis falls into the category of **failure analysis**, which encompasses all the actions necessary to determine how failures propagate in the system architecture. Over the years, many analysis techniques have been introduced to assist in understanding system behaviour. Two classical, yet still widely used, examples of such techniques are the Fault Tree Analysis (FTA) and the Failure Mode and Effects Analysis (FMEA). With the information from these analyses at hand, reliability engineers are able to establish measures that can potentially prevent that failure. In certain industries, assessment processes help the developers to set safety requirements for high-level system functions and subfunctions, during design, in order to mitigate the associated hazards. Once the architecture is set, it is possible to distribute these requirements to lower-level systems or components and evaluate whether or not that specific implementation meets the initial high-level requirements. Assessment processes are often directly connected to the structure of the system's architecture; thus, the increasing scale and complexity of modern systems makes their application in a manual fashion, arduous, inefficient and prone to errors.

Analysis techniques and safety assessment processes can be helpful in understanding the system and allowing for the design and implementation of hazard mitigation means. However, poor design choices, errors during the development or improper application of processes may still occur due to

the human factor. To assure that such mistakes are minimised to an acceptable level, the methodology of development needs to be evaluated and documented along with the corresponding reasoning. The latter involves a set of activities that are commonly referred to as **safety assurance** and often their progress is encapsulated within a set of documents known as the **safety case**. This documentation is also responsible for providing a set of clear claims regarding a system's safety – commonly established as a **safety argument** – and their supporting evidence. Regulatory authorities suggest, and for some industries require, that a safety case is maintained as a 'living-document' during a system's life span. Examples of such requirements can be found in the railway industry (HSE, 1994) and the defence sector (MoD, 1996a).

### 1.3 Evolution of Safety Assurance

In the past, safety assurance was largely governed by a rules-based, prescriptive approach (Leveson, 2011). That approach was mainly supported via domain-specific standards, which prescribed in detail all the procedures, analyses, approved tests and other routines that could assist in achieving higher system safety. This model for managing safety was centralised around coverage of previously encountered incidents and the application of practices established from experience in the relevant field. However, with the succession of large-scale accidents during the second half of the 20<sup>th</sup> century, it became apparent that both the concept and the application of that paradigm was bearing weaknesses. Albeit functional with simplistic systems, the approach was not performing well as system integration and complexity grew. Additionally, the 'checklist-mentality' was limiting the consideration of system contextual details or environmental factors as the causes of potential hazards and in many cases restrained the engineers from detecting obvious development errors. Another issue with the previous model was the potential of the regulations to be misinterpreted and lead to incorrect actions. In response to those issues and their role in the accidents, regulatory bodies and industries reformed the model of safety assurance. Modern safety standards have been reshaped towards a more descriptive regime and therefore typically provide guidelines and suggestions instead of strict rules. This allows for flexibility regarding use of processes, methods or tools during the development; however, the

responsibility of managing safety has been placed on the developers. Under this scope, the safety case serves as the developers' medium to communicate an argument of safety regarding the target system. Based on the industry-specific standard, the system application or the developer's expertise, the processes for safety assessment and assurance may differ and the safety case has to be adjusted appropriately. For instance, engineers may have a different interpretation of the standard and employ alternative methods to produce the evidence necessary for the safety argument. The decision-making and any assumptions regarding the standard or safety activities should be included in the safety case to diminish ambiguity and help reveal possible errors. Similarly, system features added later in the development cycle should be incorporated in the original argument with additional claims and evidence. This effort to maintain the safety case is necessary and, once again, it is the developers' responsibility to carry out the task.

The framework for safety analysis and assurance is typically highly complex, considering various regulations, guidelines and safety standards that need to be accounted for during development. It is essential for most safety-critical systems to prove that they satisfy certain requirements in order to receive the certification that authorises their operation. In cases where certification is not compulsory, it is still beneficial for a company as a means to establish good reputation and demonstrate high integrity. Certification is generally administered by independent government or industry regulatory organisations and provides confidence to the public that the system in question is compliant with the appropriate regulations and is acceptably safe to operate.

The nuclear industry has significantly influenced the development approach related to safety assessment and assurance. In fact, the concept of safety cases originated from that industry and has been widely adopted by other safety-critical domains ever since (Bloomfield and Bishop, 2010). Initially, the production of safety cases was an activity that would take place only at the very end of the development lifecycle. However, it became evident that the development of the safety case in early stages can be more valuable and help guide later decisions. For instance, the UK's safety standard for safety management requirements for defence systems, DS 00-56 (MoD, 1996a:10),

suggests that in order to better identify and mitigate hazards, while the opportunity exists, the safety case should be initiated as early as possible. Following this principle, a communication cycle between the safety case and the system development is created, allowing for dynamic feedback. When weaknesses are identified through the safety case, it is possible to go back and change the design, which will consequently create the need to update the safety case. Naturally, the continuous assessment of the system and readjustment of the safety case can reveal weaknesses within the architecture and result in a safer system backed up by a thorough safety argument.

#### 1.4 Significant Changes in Safety Cases

As mentioned earlier, a safety case was originally introduced as a set of documents – even now this can be true. Initial approaches to constructing safety cases expressed argumentation using only unstructured plain text. That form harboured a number of weaknesses. First of all, maintaining a cohesive structure with the excessive volume of information that normally characterises a safety case through plain text was difficult. That became clearer during the approval and maintenance stages, when the stakeholders of the project read and endeavour to understand a convoluted document with heavy use of cross referencing. Further, it was challenging even for experts to provide clear and well-structured arguments and appropriately connect them with the evidence within the documentation. To overcome these obstacles, a new approach for structuring and presenting a safety case was developed, referred to as **safety argument notations**. Introduced in the early '90s, the Goal Structuring Notation (GSN) and the Claims Arguments Evidence (CAE) are graphical notations that provide a new layout for structuring safety arguments. Both these notations provide a set of graphs that represent the argument claims, logic and evidence as well as other auxiliary elements. Note that these notations feature elements of common meaning under different names. GSN uses goals, strategies and solutions, whereas CAE refers to those elements as claims, arguments and evidence, respectively. Goal or claim nodes serve as requirements, objectives or other properties the target system is argued to meet. Strategy or argument nodes represent the rationale that connects claims or subclaims between them or with the appropriately generated evidence. Solution or evidence nodes

act as references to information, typically derived from safety analyses, to support the claims. Additionally, the notations provide a set of rules that define the validity of connections in order to help prevent vague structures. The graphs aid the comprehension and concise illustration of the argument logic and preserve a hierarchy between claims. This hierarchy allows the engineer to better understand the connection between higher and lower-level claims about the system.

The increasing interest in software design patterns, especially since the work published in (Gamma et al., 1994), has also inspired reliability engineers to adapt the concept and incorporate it in safety cases as another layer of structural management. Specifically, argument patterns allow the generalisation of successful argument structures which can be re-employed when suitable. The latter promotes re-use of high-quality argument structures and facilitates the development of safety cases. Argument modules are another concept that enables the segmentation of the overall argument into smaller subgraphs and encapsulates them within new element types, known as modules. Under this segmentation, the analysts gain access to a concrete and summarised overview of the safety argument. This is really helpful when managing safety cases of complex systems and benefits are discussed in Section 2.11.

## 1.5 Maintenance Issues in Safety Cases

Construction and maintenance of safety cases typically involves considerable manual work and can be consuming in both time and cost. During the design and development stages, a system undergoes numerous changes. With each iteration, safety engineers are expected to analyse the system for safety concerns, adjust the safety argument, re-evaluate the rationale, formulate a thorough justification for their decisions and provide feedback to other development teams. The argument reconstruction is often handled manually, and rigorous reviews are required to ensure that the rest of the argument remains consistent and valid. Note that feedback is highly important for the synchronization between the architecture and the safety case in order to avoid discrepancies between the two that could result in confusion and mistakes.

Considering all this workload in conjunction with activities such as safety analyses, feasibility studies, team meetings and other internal activities, it becomes apparent that maintenance of a safety case is not a trivial task. Modern systems development is characterised by a set of processes applied in a repetitive manner and any delays in a specific process might result in major development setbacks further down the development lifecycle. This is more likely to happen when the underlying activities, involved in safety case construction, are performed manually. In addition, the dependency between the safety case and the system architecture introduces scalability issues. Specifically, when the system architecture is complex and extended, the safety argument size grows at an increasing rate. A notable example is described in (Denney et al, 2014) where a preliminary safety case for an airport surveillance system reached about 200 pages and it was estimated to increase in size even further. Contemporary safety standards suggest that the safety case should remain a living document, meaning that the developers are required to revise it whenever appropriate during a system's life span. The assessment of whether or not a particular change may have an impact on system safety is already a tedious process and this extends further during safety assurance when the construction and maintenance of safety cases is handled manually. The need for continuous revision of safety cases in conjunction with the aforementioned problems make the traditionally manual approaches infeasible for modern safety case construction and maintenance.

To tackle these problems, it is essential to investigate the necessary procedures advocated by safety standards regarding the assessment processes and structure of safety arguments and to consider any possible means to facilitate the development of safety cases. Firstly, a thorough investigation was conducted on the ISO 26262 and the ARP4754-A safety standards that provide guidelines for the automotive and aviation industries, respectively. The main reason for choosing these standards was their utilisation of safety cases as a means of certification and the people's familiarity with associated systems on a daily basis. This investigation unveiled that safety assessment activities and results from safety analyses can be directly utilised for the synthesis of argument structures. Moreover, the assessment activities are recommended to be conducted in an iterative manner. Finally, the standards

also suggest that safety assessment activities should be performed in a top-down approach following the system architecture; thus, it is only logical to follow the same approach when mapping the related information to the argument structures. These observations allow the assumption that parts of the safety case can be systematically determined. Standardising the way to map the information from assessment activities to the argument structure and automatically conducting safety analysis on the system architecture would allow to largely automate the construction of the safety argument. This line of thought leads to a semi-automatic approach for safety case development that supports the mitigation of the aforementioned issues, such as the scalability. The implementation of such an approach seems promising for the integration of safety assurance in early stages of development that promotes safety-driven design processes while complying with contemporary standards. The automotive and civil aviation industries are valid candidates for this concept since they support a top-down process, similar to what was described earlier. Specifically, their process centralises system certification around allocation and satisfaction of safety requirements known as Safety Integrity Levels (SILs). More details on SILs and their role within the process described here are presented in Section 1.6.

Since this section discusses the maintenance of safety cases during a product's lifecycle, it is crucial to examine how safety cases adapt to the different phases during the development lifecycle and what the underlying arguments communicate. The development lifecycle consists of three main phases, where the first phase involves all requirement validation processes, the second is the implementation of the system design and the last phase includes the requirement verification tasks. Validation of requirements is applied in the early design stages and aims to confirm that the requirements identified at any given level in the architecture contribute to the top-level safety objectives. The implementation stage typically covers both software and hardware, and the output can be procedures, detailed designs, source code and prototypes. Finally, requirements verification is conducted after the implementation and evaluates whether or not the current implementation properly meets the defined requirements. The safety case differs across these phases in terms of what the corresponding arguments attempt to



communicate. During the validation stage, focus is given on the correctness and suitability of the requirements chosen. When the implementation is established and during the verification phase, the arguments are centralised around the correctness of the system behaviour with respect to the requirements. Following this logic, it is concluded that the system has correctly identified requirements and has been implemented correctly to meet these requirements.

For a complete safety case, it is also important to examine the types of arguments available. Safety certification often requires two types of arguments, the process-based and the product-based. Product-based arguments have been at the centre of this introduction and involve the particular properties a given system must fulfil to be acceptably safe. Process-based arguments are directed towards the correctness of the processes involved as well as the planning and management. They also aim to address issues with the integrity of utility tools and personnel competence. It has been argued in (Habli & Kelly, 2006) that proper linking of process-based arguments in the evidence, supporting the product-based arguments, can help build more confidence on the evidence's provenance.

## 1.6 Safety Requirements Described in Safety Standards

For the electrical, electronic and programmable electronic safety-related systems, the International Electrotechnical Commission (IEC) has initiated a new type of safety requirement. **Safety Integrity Levels** (SILs) were introduced in the international IEC 61508 standard, which revolves around the safety lifecycle and has been customised to generate industry-specific standards. In this generic safety standard, safety integrity is defined as the probability of a safety-critical system to acceptably perform the required safety functions under stated conditions within a stated period of time. In IEC 61508, safety functions are described as functions of a system or equipment whose operation affects the overall safety (IEC, 1998). SILs are considered meta-requirements since they are basically deployed as a common mechanism to represent abstract safety requirements for distinct system elements. This means that the same SIL assigned in two different components does not necessarily lead to same actions or cost required for the development of that entity. SILs are generally categorised in different levels based on the level of stringency and can be allocated to different layers in the architecture.

Typically, SILs are initially assigned to system-level functions according to the risk of the associated hazards. Then, they can be progressively refined and distributed to subsystems and components by following a set of standard-specific rules. Modern standards advocate certification via SILs, where the main objective is to determine whether or not system-level SILs have been met via satisfaction of component-level SILs.

SILs have been instrumental in the development of software and hardware systems due to the nature in which such systems fail. A failure is the inability of a system to perform its intended function and can generally be classified into **systematic** and **random**. Traditionally, mechanical systems are characterised by finite reliability since random failures are bound to happen due to physical degradation and deterioration from continuous use. On the other hand, software and partially hardware components suffer from systematic failures, which can be introduced during the specification, design, manufacturing or installation. This type of failure occurs deterministically when certain conditions that trigger the error are met. In this context, errors can be any embedded flaws in the system (e.g. software bug). Random failures typically follow probabilistic distributions over time and can be computed using statistical analysis methods. Systematic failures can be determined via testing and formal methods. Testing is usually deployed to examine the behaviour of a program and identify how thoroughly it follows the specification. Unfortunately, the input domain is usually immense and prohibitive for exhaustive testing; therefore, only a part of the software is tested for correctness. Formal methods prove program correctness via mathematical reasoning. They normally require the program specification of inputs, the specification of outputs and with this information aim to show that a post-condition is true after program execution. Formal methods might be exceptional for proving safety-related properties; however, they can be complex to deploy and prohibitively time-consuming to apply for large system architectures. Frequently, a balanced combination between testing and formal methods is both feasible and practical. SILs act as a key factor for deciding the level of rigour of the assessment activities applied on individual subsystems and developers can target specific elements in the architecture based on the SIL assigned. Higher-level SILs suggest higher

stringency, thus any subsystem assigned a high-level SIL requires rigorous assessment and meticulous documentation. Elements in the architecture with the lowest SIL may even be omitted from the evaluation process as not significant for safety; thus, saving development time.

Safety standards prescribe that once SILs for high-level system functions have been assigned, they need to be allocated throughout the system architecture in a top-down fashion. This process follows the development lifecycle in the sense that every new element added in the architecture at any point, requires re-allocation of requirements (SILs). Furthermore, industry-specific standards have also provided a set of rules for addressing SIL allocation in lower-level components. These rules may apply depending on the system architecture or when failure countermeasures, such as redundancy, are introduced. Specifically, in the scenario where a system element is solely responsible for the failure of its parent-system, it will inherit the SIL of its parent-system. If more than one lower-level elements are responsible for the occurrence of a hazard, these elements are assigned lower SILs based on the rules. This decomposition mechanism, in practice, can influence the development cost since systems with lower-level SILs require less rigorous assessment. The decomposition rules prescribed by standards are simple, but as the system architecture grows, the number of possible combinations of SILs across the system architecture increases drastically.

In complex systems, the design space is vast and features numerous different SIL combinations. A competent practitioner is required to meet the appropriate SILs in order to ensure acceptable system safety while minimizing development costs. However, manual navigation towards the optimal solution can be an intractable problem for large systems. Optimal allocation of SILs can be viewed as an optimisation problem, where the goal is the minimization of development costs and the constraints are the decomposition rules prescribed by the standard. There is a substantial body of work around methods that use optimisation techniques in an attempt to automatically and optimally allocate SILs. More details regarding optimisation techniques able to tackle this problem as well as the specific technique employed in the proposed method are discussed in Chapter 3.

## 1.7 Model-Based Development and Analysis

Classical safety analysis techniques are often employed on informal system models to evaluate the architecture at hand for safety concerns. The correct application of these techniques and other assessment activities are highly dependent on the experience and view of the practitioner. Moreover, following a manual approach might be adequate for small and trivial systems; however, as systems grow more complex, the probability of human errors increases, and the required workload accumulates and becomes limiting for large scale applications. As a result, both the effort and the associated costs required with traditional methods do not allow to conduct safety assessment activities as often as required in modern systems development.

An alternative paradigm was introduced to address the issues discussed earlier related to the development of safety-critical systems. This paradigm, known as Model-Based Safety Analysis (MBSA) was inspired by the pre-existing model-based development. MBSA rejected the asynchronous development of conventional methods and enabled the linking of design and development processes with safety assessment activities via the introduction of formal models. Under this notion, safety analyses can be incorporated iteratively in the development cycle and support, to an extent, safety assessment activities from the early design stages. Prior research has investigated the concept of exploiting MBSA methods to assist in the generation of safety arguments for the aerospace industry. This thesis further explores how MBSA can be integrated with safety assessment and assurance and provides a framework for the semi-automatic synthesis of safety argument structures with cross-sector application. The method described in this thesis links the safety cases with system design models and supports a safety-driven development process.

Towards this direction, safety analysis methods that employ the MBSA paradigm, which is supported by modern safety standards, are utilised as a base for conducting system analysis. Notably, Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) is a mature body of work that follows the MBSA paradigm and, among other features, is capable of automating FTA and FMEA analysis techniques. Further, through recent extensions, it allows for the automatic and

optimal allocation of safety requirements in the automotive and aerospace domain. Thus, by exploiting and extending the capabilities of HiP-HOPS, the operationalisation of the proposed method can be facilitated.

## 1.8 Research Motivation

Modern safety standards help guide the developers in the processes involved in the design and safety assessment of safety-critical systems. They advocate that systems need to be assessed for safety throughout their lifecycle and provide a series of tasks that can be applied during the development stages. Despite the evolution of guidelines and tool support, a large amount of the tasks involved are still manually performed. Also, considering the increased complexity of contemporary systems, the certification process can be susceptible to errors and costly in time and effort. This thesis argues that automation and tool support can be key factors in mitigating these issues. Moreover, it presents common elements found in safety standards and defines a general pattern suitable for the development of safety cases across different industrial domains.

## 1.9 Research Hypothesis

Current practice indicates that the production and maintenance of safety cases remains largely a manual process. Safety cases are not connected to the models of the systems which they represent, and synthesis of safety arguments is often omitted in the early stages of development. Maintaining a safety case as a ‘living document’ requires significant effort and time reassessing the system for safety and updating the safety argument. Modern standards have introduced the concept of safety integrity levels and provided a set of rules for decomposing and allocating these requirements throughout the architecture. This thesis examines these standards and defines a pattern that represents the converging common features. It also integrates this pattern with model-based safety analysis methods and safety argument patterns under a common framework which connects the safety cases with system models. The hypothesis is that this concept of a model-connected safety case improves model-based safety assessment in the following dimensions:

- Enables semi-automatic synthesis of safety arguments
- Enhances design iteration by directly providing feedback from safety analysis back to the design team
- Promotes safety assurance in early development stages

## 1.10 Research Objectives

The following objectives were defined with the aim to explore the hypothesis:

1. To examine safety standards for converging common elements and propose a pattern for safety assessment and assurance across the relevant domains
2. To operationalise the pattern with the development of a metamodel for connecting safety arguments with system models
3. To implement the metamodel in a software supporting tool to demonstrate the feasibility of the method
4. To evaluate the approach with a case study

The first objective involves the investigation of safety standards and how they propose safety assessment and assurance. This process is important for the identification of the converging elements from which a pattern for assessment and assurance in these domains is derived. The second objective involves the design of a metamodel suitable for representing all pattern elements and the inter-relationships between the system model and safety arguments. Next, the implementation of the metamodel in a software supporting tool that materialises the concept and facilitates the demonstration of the approach comprise the third objective. The final objective concerns the use of a case study for the evaluation of the method and the potential for cross-sector application as well as reuse of information models (system design, analyses results and argument patterns) towards the synthesis and maintenance of safety arguments.

## 1.11 Thesis Structure

The remaining chapters of this thesis along with a brief description are provided below:

## **Chapter 2. Background**

The background chapter presents the foundations of the research area. Specifically, it begins with the fundamental terminology found in reliability and safety engineering as well as other terms frequently used in this thesis. Next, it focuses on the and assessment aspects of the research area such as safety analysis techniques, the MBSA paradigm, derived methods and tool support. Then, it guides the reader into the field of safety assurance by introducing the notion of safety cases, the corresponding challenges, and safety requirements as described in various safety standards. Finally, it presents established metamodels and specification languages in safety assurance that could have been useful when designing the method or during the implementation of the supporting tool.

## **Chapter 3. Safety Assessment in Automotive and Aviation**

This chapter describes how safety assessment is generally guided in modern standards for safety-critical systems. It then explains in more detail how assessment is governed under the guidelines of the ARP4754-A and ISO 26262 safety standards for the aerospace and automotive domains, respectively. Next, the rules for decomposition and the challenges for an optimal allocation of safety requirements (e.g. ASILs/DALs) are presented along with a summary of some of the techniques available for tackling this issue. Finally, the chapter concludes with an abstract example of how requirements allocation is managed in the implemented framework, through a HiP-HOPS extension.

## **Chapter 4. Connecting Safety Cases to System Models**

This chapter begins with a review on the persisting issues in safety assurance and then presents an overview of the proposed method. Specifically, it provides all the major phases and demonstrates how these are performed via a simple example. The chapter concludes with an introduction to and comparison with earlier approaches and tools that have common elements, to a certain extent, with the work of this thesis.

## **Chapter 5: Design of Model-Connected Safety Cases**

Chapter 5 presents the software engineering elements, structures and primary algorithms that constitute the implemented software environment, presented in this thesis, which serves as the proposed method's supporting tool. It also explains the connection between the various metamodels under a unique framework and argues about the choice of the development environment and language.

## **Chapter 6: Case Study**

This chapter presents the application of the method in a more detailed case study, based on the brake system of the Boeing 787, to better demonstrate its applicability and practicality.

## **Chapter 7: Conclusions**

This final chapter begins with a brief review of the issues related to safety assurance and presents the contributions of this thesis on the matter. Following, a set of evaluation criteria are introduced which are used for assessing the method outcomes. Finally, areas for further work are also discussed.

## **References**

A list of references used in this thesis.

## **Appendix**

In the Appendix there are a few images from the example in Chapter 4 as it was tested on the software tool and a brief, yet detailed, explanation of the means utilised for the measurement of the execution time of the automatic processes within the implemented framework.

### **1.12 Published Material**

The initial idea and basic elements of the method described in this thesis have been presented in the following publication:

- Retouniotis, A., Papadopoulos, Y., Sorokos, I., Parker, D., Matragkas, N. & Sharvia, S. (2017) Model-Connected Safety Cases. 5<sup>th</sup> International Symposium, IMBSA 2017. Trento, Italy, September



2017, Lecture Notes in Computer Science 10437:50-66, Springer, ISBN 978-3-319-64118-8, ISSN  
0302-9743, Available online @ [https://doi.org/10.1007/978-3-319-64119-5\\_4](https://doi.org/10.1007/978-3-319-64119-5_4)

## Chapter 2 Background

---

The proposed method incorporates safety analysis and other standard-related activities along with argument patterns and system modelling to enable semi-automatic safety assurance. Moreover, for operationalisation purposes, a metamodel and a supporting tool were designed, implemented and are presented later in this thesis. Before proceeding, it was essential to examine the research area and current practices. Specifically, what the well-known safety analyses techniques are, understand how these are conducted, what their outcomes are and how they can be utilised within the proposed method. It was also critical to comprehend what safety standards advocate in order to ensure that the method is compliant and therefore usable in the certification process. Prior to the implementation, it was also important to review previous approaches and tools that can be employed directly, where appropriate, or help with the design process. Hence, this chapter presents the key findings related to safety analysis and assurance as well as available tools for this purpose. Initially, it establishes useful definitions and common terms being used throughout this thesis to help the reader familiarise themselves with the research area. Then, it presents a more in-depth view of safety assessment methods and analysis techniques as well as brief descriptions of notable analysis tools and examines whether or not these could be used in the proposed approach. Finally, the chapter proceeds with the assurance aspect of certification and provides a review on safety cases and related issues, graphical notations as well as existing metamodels and languages for safety engineering applications. The literature review presented in this chapter is connected with the first two research objectives, the examination of safety assessment techniques and tools directly usable by the method and the understanding of related issues that the method and tool should try and resolve. Note that this chapter is quite extensive since the proposed method has a wide scope and integrates a large portion of the safety-related tasks during the development lifecycle of safety-critical systems.

### 2.1 Common Terminology in Safety Assurance

Before leaping into the details of safety assurance, it should be helpful to explain common terms and review some of the key underlying procedures. First off, this discussion should begin with one of the

most significant properties of systems, whether mechanical or hardware or even software – **reliability**. In most cases, systems that are widely used must be highly reliable, which means that they are less likely to fail. In (IEC, 2020), reliability is more accurately defined as “the ability of a system or component to perform its required functions, without failure, under stated conditions for a specified time interval”. Based on this term, any component, no matter its significance in the architecture, can be tested for reliability as long as it is suitable for individual analysis and testing. The word “conditions” refers to any sort of environmental parameters, stress levels or different operation states. For example, when examining an aircraft for reliability, environmental parameters may refer to weather conditions, such as strong winds, the air traffic or the aircraft’s physical state that can be estimated from maintenance monitoring and obsolescence metrics. Another useful description in the (IEC, 2020) states that reliability can be viewed as “the degree to which a system, product or component performs specified functions under specified conditions for a specified period of time”. Following this definition, it is possible to interpret reliability as the probability that an item will not fail to perform its intended function for a given period of time ( $t_0$  to  $t_1$ ) (Elsayed, 2013). Assuming a system may be responsible for multiple functions, it is reasonable to consider that it also has an equal number of reliability evaluations that correspond to those functions. Based on the second definition provided, a generic formula for calculating reliability is derived and shown below:

$$R(t) = \text{Probability}(\text{system does not fail during the period of } [t_0, t_1])$$

In the formula above,  $R$  represents the reliability and  $t_0$  and  $t_1$  the stated time period. Another useful term in reliability engineering is **unreliability**. In simple words, it is the probability of failure for a system to perform its intended function. That being said, there are basically only two situations during the operation of a system: success or failure. These states are mutually exclusive, and their probabilities are the reliability and unreliability, respectively. The sum of the probabilities for those two states are always equal to unity (i.e. 1). Systems with high failure rates are characterised as unreliable and it may be preferable to measure their unreliability instead. The relationship between reliability and unreliability is given by the following formula:

$$U(t) = 1 - R(t)$$

The formulas presented here are in simplified forms to help understand how the concept of reliability can be quantifiable; originally, they derive from probability and cumulative density functions.

In general, reliability, for mechanical and electric systems, follows a Weibull ‘bathtub’ curve as seen in the adaptation from (Wilkins, 2002) shown in Fig. 2.1.

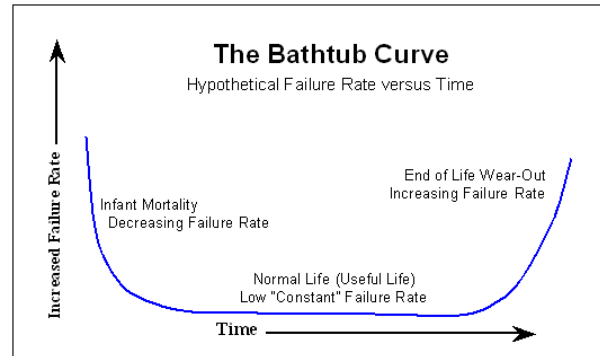


Figure 2.1: The Bathtub Curve (Wilkins, 2002)

During the early stages in a system’s lifetime, the failure rate is high due to inherent deficiencies in design and defect components, but it decreases monotonically as the system matures. The second period (i.e. normal life) is typically characterised by low failure rates, which mostly comprise random failures. Towards the end of the lifetime, the failure rate increases due to material degradation and wear, and as a result the reliability of the system decreases monotonically. Electronic systems are also subject to physical degradation; however, engineers are more concerned with systematic failures that happen during operation due to design flaws and system interactions, as well as specification errors. Despite all the effort for reliable systems, they may still be susceptible to failures. Taking this fact into consideration, it is crucial to design with safety in mind so that system failures will not cause unacceptable harm to the community or the environment. **Safety** as a notion has neither a universal definition nor a specific quantitative form and its meaning typically relies on the domain. An investigation into safety standards and other sources might help us understand how safety is defined in relevant industries and highlight any similarities. Table 2.1 below presents these findings.

Table 2.1 A definition of safety from various sources

Definition of Safety	Source	Source description
“expectation that a system does not, under defined conditions, lead to a state in which human life, health, property, or the environment is endangered” (IEC, 2010)	IEC 61508	The cross-sector standard for functional safety
A state where “all identified risks to life are reduced to levels that As Low As Reasonably Practicable (ALARP) and tolerable”, unless a more stringent policy or regulation applies (HSW, 1974:2)	Defence Standard 00-56	The UK’s Ministry of Defence (MoD) standard
“the state in which risk is acceptable” (SAE, 2010:13)	ARP4754-A	The US aerospace industry standard
“absence of unreasonable risk” (ISO, 2011:14)	ISO 26262	The automotive safety standard
“the ability of a system not to lead to dangerous or catastrophic events under a set of conditions”	(Villemeur, 1992)	Book on reliability & safety assessment
“freedom from unacceptable losses”	(Leveson, 2009)	Journal paper on safety & reliability

The phrasing might differ, however a closer look at the definitions allows us to understand that safety is directly correlated to the concepts of **risk** or **harm**. Investigation for the definition of risk in various sources leads to the conclusion that risk is related to harm. Specifically, risk is described “as the combination of probability of occurrence of harm and the severity of that harm” (IEC, 2020). Based on the IEC 61508 safety standard, **harm** is conceived as any “physical injury or damage to property or the environment” (IEC, 2020) that happens directly or indirectly. Risk can also be expressed as the probability of a failure occurring (metric for reliability) and the severity of its consequences (metric for safety). This indicates how reliability and safety are closely related to risk and why the two properties are often, but incorrectly, used interchangeably. Undeniably, improving the reliability of a system will reduce its probability of failure and consequently will lessen the chances of an accident occurring as a result of that failure. Nevertheless, that is not always the case since a system can be reliable, yet unsafe, and vice versa. Safety is not solely connected to architectural decisions and requirements, but it is also associated with external factors, environmental interactions and the intent

behind system functions. Therefore, a system might behave correctly based on the specification for a given period of time, thus being reliable; however, it is not safe if any component interactions or environmental conditions were unaccounted for during the design of system functions or if the system was designed to cause harm (e.g. weaponry). For instance, an aircraft's main function is to transport passengers and by achieving this function with no failures, it is considered to be reliable. However, if the air quality is poor and negatively affects the passenger's health in the process, that makes the aircraft unsafe. Risk is certainly undesirable and during reliability and safety assessment activities, effort is made to minimize it. Risk management tasks typically involve an investigation for sources of risk, an evaluation of their probability to occur and their potential impact. Then, the engineers assess mitigation means that can reduce the likelihood of occurrence or at least weaken the impact of failures. Through the investigation of safety standards, it was shown that any properties, conditions or events associated with system functions and having the potential to cause harm are referred to as **hazards**. Hence, identification of hazards and provision of suitable mitigation means or lowering their probability of occurrence would be expected to reduce risk.

Reliability engineering is the process employed that attempts to increase the reliability of an entity by minimising its associated risks, constrained by production costs and time required for assessment. Initially in the process, analysis identifies a number of flaws in the system and actions towards their potential solutions are decided. For instance, components that are likely to fail can be: a) replaced with more reliable and higher quality counterparts, b) replicated for redundancy purposes or c) entirely excluded due to problematic behaviour affecting the encompassing architecture. There is a variety of analysis techniques, commonly referred to as **systems analysis techniques**, which help examine system behaviour and yield useful information. The results provided can be utilised as proof towards safety or reliability. Although this data collected might be adequate for developers, it is certainly insufficient for the certification of systems. Regulatory authorities and project stakeholders typically require well-structured and comprehensive arguments that logically establish confidence in the identification of hazards, the employed methodology, the decisions made and the implementation.

In (SAE, 2010:10), the concept that encapsulates all the associated tasks responsible for building confidence and producing the supporting evidence, that a product or process satisfies a set of given requirements, is referred to as **safety assurance**.

Finally, the term “**system**” has been repeatedly used and it is necessary to explain its meaning under the context of this thesis. A system implies a number of individual elements that can work collectively towards a specific purpose. The ARP4754-A safety standard provides us with a more formal definition:

“a collection of inter-related [elements] arranged to perform a specific function(s).” (SAE, 2010:13)

The term “elements” is assigned to any unique hardware or software components with precisely described functionality and interfaces (i.e. input and output). The next sections investigate safety assessment and assurance activities in the development of **safety-critical systems**. As mentioned in the Introduction, failures of this type of systems can potentially result in harmful or even catastrophic events (Rausand, 2014).

## 2.2 Safety-related Processes in System Development

Before delving into safety cases and domain-specific practices, it would be helpful to briefly introduce some of the procedures involved in safety assurance and assessment during system development.

### 2.2.1 System Modelling and Specifications

Early in development, engineers typically decide the broad requirements and context of the system. This phase results in an informal specification that sets bounds of both the expected and unexpected system behaviour under various conditions. Over time, the context becomes more specific and relevant dependencies are defined which help shape a preliminary architectural **model**. The engineers gradually refine the system architectural model before its use during the implementation and production phases. Apart from the architectural model, a system can be described in other types of models that represent different angles of view (Sommerville, 2011). With the appropriate modelling notation, it is possible to highlight different aspects of a system. For example, with a UML activity

diagram, it is possible to depict all the activities related to a system process and examine the order in which these activities are performed. Considering primary perspectives, models can be categorised into (Sommerville, 2011: 121-138):

- a) **Context models** that describe system functionality under given constraints and its usefulness
- b) **Interaction models** that demonstrate interactivity with any sort of input or output
- c) **Structural models** responsible for exhibiting the underlying subsystems and components as well as their interconnections
- d) **Behavioural models** that show the dynamic behaviour during execution in response to internal or external events

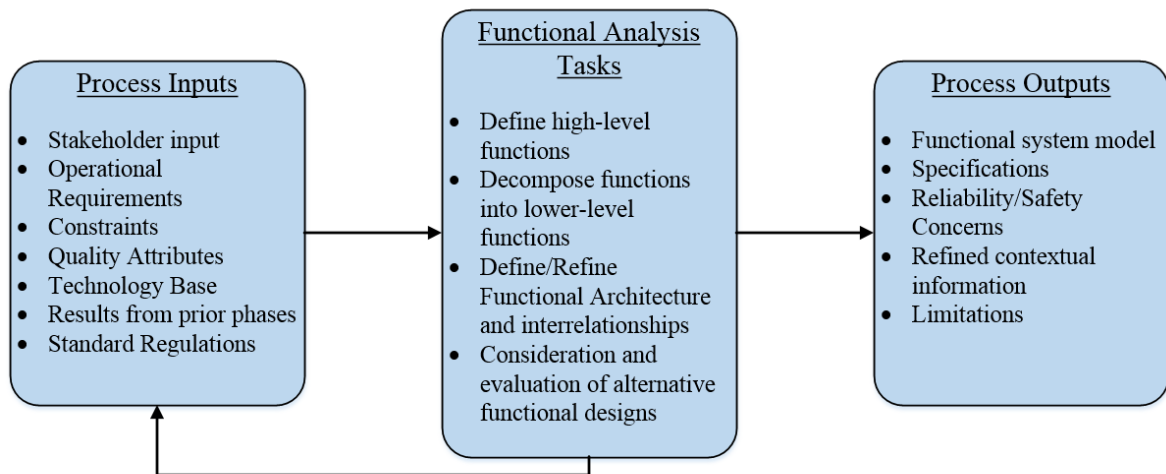
According to their format, models can also be classified into conceptual and mathematical. Conceptual models are typically abstract and encapsulate high-level design elements. They are often employed to help communicate the ideas behind the subject which they represent. Mathematical models, in general, describe systems with the use of mathematical terms and analytical formulations. Further classification exists and depends on their structure (i.e. linear models, dynamic, probabilistic etc). Any quantifiable system parameters are expressed as variables whereas system interrelationships are specified with the use of operators (e.g. algebraic operators or functions). In addition, mathematical models allow the construction of detailed specification models suitable for formal analysis and verification.

## 2.2.2 Functional Analysis of Systems

Prior to the design of the architecture, it is common practice to identify base system functionality towards its environment, goals and objectives that lead to the functional model. Each function represents a system requirement that comes with a specific purpose and possibly various constraints. It is typically executed by one or more system components, either software or hardware, or by the user. Initially, the identification process focuses on high-level functions rather than the details of operation. As time progresses, through the iterative cycle of design and analysis, the system model is refined and the originally defined high-level functions can be decomposed and allocated throughout



the architecture. The identification of functionality, especially early in the development, can be beneficial as it allows the engineers to build a conceptual framework which makes it easier to comprehend the objectives or restraints and help guide the architectural design. Specifically, understanding the functions and any associated elements often results in more detailed requirements responsible for further architectural adaptations. In addition, this knowledge is extremely useful when considering design alternatives, as these can be rejected when they do not meet key requirements. All this process described is performed iteratively until a detailed hierarchical functional model is composed. Fig. 2.2 provides a summary of the functional analysis flow process based on information found in (DoD, 2001; FAA, 2006:4-4).



*Figure 2.2 Functional Analysis flow process*

In this abstract diagram, the element on the left includes any requirements, constraints or other necessary information gathered from requirements analysis and stakeholder instructions. The second element utilises all this as input for the functional analysis tasks, responsible for the definition and decomposition of functions. Finally, the right element presents the basic output, which often comes in the form of the functional model, detailed requirements, concerns and limitations.

### 2.2.3 Functional Hazard Analysis

Functional Hazard Analysis (FHA), also known as Fault Hazard Analysis, is a process first introduced in the aerospace recommended practice document ARP4761 (SAE, 1996:16) and earlier comparable approaches can be found in (IEEE, 1994). FHA enables the investigation of system functions for failure conditions and related safety hazards. Traditionally, in FHA failure conditions are classified into functional losses or malfunctions (USA Department of Defense, 2012:208); albeit more classifications have emerged to expand the notion of failures types. The process is applicable immediately after the identification of major high-level functions. With this information, it is possible to consider various hazards (e.g. loss of function) and any associated effects. Once safety hazards have been established, for each system function, the developers estimate their consequences based on the probability of occurrence, during normal operation, and criticality. From the latter, the functions receive a severity classification that varies from catastrophic to no-safety-effect. Note that the FHA is performed iteratively in a top-down manner starting from higher-level functions towards lower-level; thus, following the current system architecture. Given the results from this process, it is possible to decide appropriate safety objectives and set requirements as mitigation means. Safety guidelines and regulatory bodies advocate that FHA should be conducted as early as possible to enable other safety analysis techniques and encourage safety-driven development. Any assumptions and generic requirements during this process are usually documented and provided as complementary information in the results. Fig. 2.3 provides a partial, abstract example of the FHA output for the fuel control of an aircraft.

Functional Failure Ref.	Function Name	Hazard Name	Hazard Description	Failure Effect	Operational Phase	Criticality Classification	Requirements
2.1.1	Fuel Tank	FT.H1	Fuel Tank Breach	Leak of gas, Fire potential	Flight Planning	Minor	Increased material quality control, Flexible material
2.1.2	Fuel Tank	FT.H2	Tank's Seal Split	Leak of gas, Fire potential	Landing	Hazardous	Double Seal, Isolate fuel tank from spark/high-heat sources
2.2.1	Fuel Pump	FP.H1	Fire	Spread of Fire, Explosion, Accident	Taxiing	Catastrophic	Place fuel pump inside fire resistant material to reduce spread, Increase safety countermeasures

*Figure.2.3: Abstract Example of FHA on aircraft autopilot system*

#### 2.2.4 Functional Failure Analysis

Functional Failure Analysis (FFA) is a safety analysis technique introduced in (Papadopoulos, 1999) and later utilised in (Johannessen et al., 2001). FFA, conceptually, is almost identical to the FHA; however, it expands by including failure classes responsible for the consistent description of failure modes. These failure classes were inspired by the guide words discussed in (Kletz, 1997) which were used as an extension to HAZOP for distinguishing deviations in specialised applications. Hazard and Operability (HAZOP) study is a structured technique that was initially published in (Lawley, 1974). The main objective of HAZOP is to evaluate a relatively detailed system architecture for hazards and causes of operational disruption (British Standard, 2001). In FFA, every system function is methodically examined for possible failure modes which can be further analysed for the identification of potential effects, severity and mitigation means. The results from such analysis can be utilised in other dependability methods towards the safety assurance of the overall system. Under FFA, failures can be classified into one of the following categories:

- **Omission**, where signals/data are inadvertently missing
- **Commission**, where signals/data are inadvertently propagated
- **Timing**, where the data has arrived too early or too late

- **Value**, where the value of data might be out of range or stuck

Note that new categories can be defined and used based on the project needs. Finally, the results of FFA are often represented in a tabular form. Fig. 2.4 provides an abstract example of an FFA output.

Failure Ref.	Function Name	Hazard Name	Hazard Description	Failure Class	Failure Effect	Phase	Severity Class	Requirements
2.1.1	Brake Force Distribution	BD.H1	Greater force on rear wheels than expected	Omission	False distributed of braking, Possible wear	Drive on straight dry road	Hazardous	Better distribution controller
2.2.1	Anti-lock Braking	AB.H1	Release of Brake Actuators	Commission	Loss of brakes	Drive on straight dry road	Catastrophic	Re-evaluation of design and part quality

*Figure 2.4: Abstract Example of FFA on a car brake function inspired by (Johannessen et al., 2001:511)*

## 2.2.5 Preliminary Safety Assessment

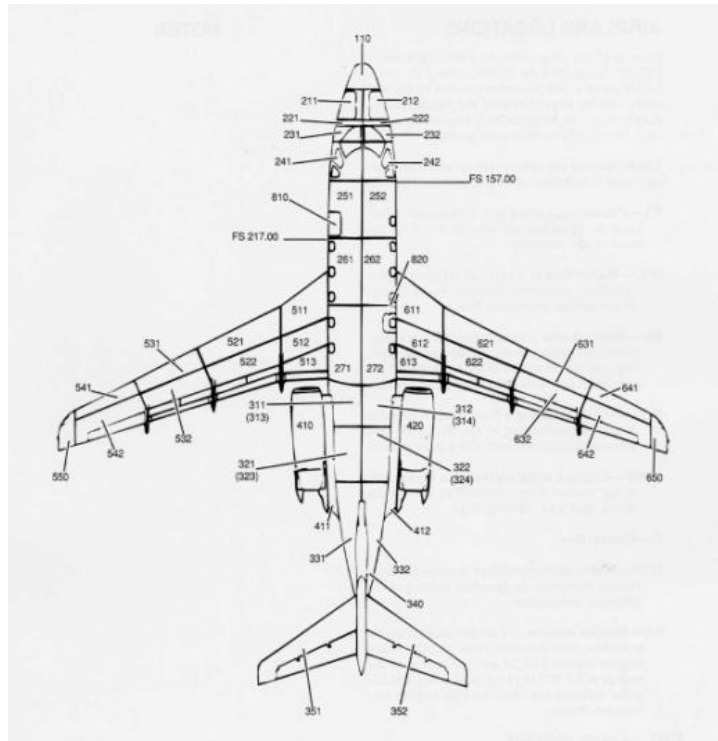
Preliminary Aircraft Safety Assessment (PASA) and Preliminary System Safety Assessment (PSSA) are safety assessment processes, proposed in ARP4754-A safety standard, that are applicable from the early design stages and are iteratively updated throughout development. Both processes aim to establish a complete list of failure conditions and the related safety requirements. They are used for evaluating how the architecture in question will meet safety requirements for the hazards identified during the FHA. PASA addresses the architecture responsible for the aircraft-level functionality whereas the PSSA focuses on the internal architectures of specific system components and lower-level items. Further, these two processes in conjunction with safety analysis techniques (e.g. Markov Analysis) allow the examination of failures and how they could lead to hazards. Based on this information, engineers are able to formulate the appropriate strategies (e.g. fail-safe architecture with redundant elements) required for meeting the safety objectives (SAE, 2010:31). Based on (SAE, 1996:17), results from both the PSSA and PASA are typically documented and used in other assessment activities or help formulate some of the argument strategies.

### 2.2.6 Common Cause Analysis

Common Cause Analysis (CCA) is a set of safety analysis techniques firstly introduced in (Mosleh et al., 1989) and later employed in assessment procedures found in various industries, such as the nuclear and aerospace. In general, CCA allows to investigate a system architecture for common cause failure events, which are considered to be any component failures or external events that act as root causes for multiple failures occurred in other system components. These failures typically occur due to the dependencies between the architectural elements and may lead to catastrophic or hazardous events. Therefore, safety engineers utilise CCA to explore system dependencies and verify either independence between system elements (e.g. functions, systems, items) or ensure that the risk associated with the identified dependencies is considered acceptable. CCA techniques are applicable in every development stage; although, in practice, they are often used in the early stages to reduce costs since appropriate mitigation means may affect the architecture greatly. CCA is divided in the following categories:

- **Particular Risk Analysis**, which is responsible for evaluating risks related to external events (outside the system architecture) but still violate any independence claims.
- **Common Mode Analysis**, which is used to verify that any failure events, identified in previous analyses (e.g. FTA), remain independent in the final implementation. That is important since the decisions made during manufacturing or human errors during production might have invalidated any claims for independence/dependence.
- **Zonal Safety Analysis**, which separates and analyses the system in specific zones (physical areas) with the goal to assure that the final installation (i.e. physical system) meets both the design and the safety requirements related to system interference (i.e. physical impact of a system failure on other systems or product structures) or issues occurred during maintenance.

CCA techniques are typically performed with the application of checklists and any results or new requirements are stored in various documents (SAE, 1996:160). Fig. 2.5 shows an example of a zone map for a small aircraft, reproduced from (Linzey, 2006:9).



*Figure 2.5: Zone map of a small aircraft (Linzey, 2006:9)*

## 2.3 Safety Analysis Techniques

Safety analysis techniques are often employed in safety assessment activities in order to understand systems behaviour and investigate for potential causes of failure. With this information at hand, it is possible to introduce measures and design architectures that either prevent undesirable events occurred from system failures or at least mitigate their effects. Currently, reliability engineers have a variety of safety analysis techniques at their disposal, suitable for different types of systems and often supported by safety standards. Among those techniques one can find classical techniques, with wide application over the years, techniques derived from the classical and adjusted to address issues in specific situations, and modern techniques that introduce new schemes for examining system behaviour. This section presents some of the traditional techniques, such as the Fault Tree Analysis (FTA), that have been utilised in several variations within the method proposed in this thesis. Despite the usefulness of these techniques, they have demonstrated limitations when dealing with modern complex systems, especially when applied manually.

### 2.3.1 Fault Tree Analysis

Fault Tree Analysis (FTA) is a classical safety analysis technique firstly introduced in the early 60's by H.A. Watson as part of a project under the supervision of the US Air Force Ballistics Systems Division team and in collaboration with the Boeing company. The technique was used to analyse the control mechanism of a new generation weapon system, the Minuteman I intercontinental ballistic missile, which was able to launch at the touch of a button (Ericson, 1999). The project was successful, and since then, many publications and references on FTA have convinced various industrial domains, including the nuclear (Vesely, 1981) and civil aircraft (Eckberg, 1964), to adopt the method.

FTA is applied in a top-down manner (deductive method), typically forming a diagram that resembles a traversed tree, known as the “fault tree”. This diagram comprises a set of graphical entities (e.g. tree nodes) that are structured logically and represent the interrelationships between the top node and other leaf nodes. Fault trees are typically used to map and demonstrate system failure behaviour. The top node (or top event) often presents the undesirable outcome (e.g. system failure) and the leaf nodes underneath portray subsystem failures. The logical combinations between nodes are realised with the use of Boolean logic gates (e.g. AND/OR), often referred to “intermediate events”, and together constitute a network of failure logic propagation.

FTA is a versatile technique that allows the engineers to gather information about the target system and assists in decision-making. Note that usefulness of the results is not restricted only to reliability or safety assessment but may be helpful in safety assurance and other development processes. First of all, FTA highlights the causes of undesirable events and the degree of contribution of each subsystem failure. Application of FTA allows for prioritisation of architectural changes which could lead to an optimised design with minimal resources. Additionally, the data collected regarding specific subsystem interactions can be exploited to improve current and future designs, by avoiding unsuccessful architectures. Finally, the ability to perform FTA during different stages in development allows its utilisation either in a proactive or reactive manner. In the case of the former, it is possible to carry out fault tree analysis on a preliminary or even conceptual system architecture to investigate

for system flaws and improve the design with minimal costs. The reactive approach is still valuable since it helps identify weaknesses in existing products and deliver an enhanced version.

Before looking into FTA any further, it is necessary to comprehend the subtle difference between the terms **fault** and **failure**. Different definitions exist on the matter; however, the most intuitive ones are presented in (Villemeur, 1992:27) and (Villemeur, 1992:22) for fault and failure, respectively. Based on these definitions, fault is characterised as the “inability of an entity to perform a required function”, whereas a failure is the “termination of the ability of an entity to perform a required action”. The authors in (Vesely et al., 2002:26) further clarify the topic by arguing that a failure is always a fault whereas the opposite is not always true. Following the rationale presented in (Vesely et al., 1981:45), it appears that any occurrence of a system not operating based on the specification due to a component malfunction or another abnormal condition can be considered a failure. On the other hand, a fault can occur even when the system is able to execute its function. For instance, a mechanism capable of performing its function, but does so inadvertently due to an untimely command from an operator, who in this context is part of the system, is considered to have faced a fault. In another scenario, a system may not perform its intended function due to design or production flaws. In those cases, the system may still be functional; however, the faults could still lead to a harmful event. This notion of fault enforces the consideration of additional factors, such as human errors and time constraints. Based on this definition, faults can be classified into:

- a) **Primary faults:** The item suffers a fault during normal operation (under conditions accounted for during the specification design).
- b) **Secondary faults:** The item suffers a fault during normal operation (under conditions unaccounted for in the specification).
- c) **Command faults:** The item operates normally, but under undesired circumstances. This could be caused from an incorrect command signal (from a human operator or an electronic controller).

Now that these details are addressed, it has become clearer how to properly use fault trees. However, before examining FTA on a simple example, it is also important to investigate how we can further



benefit from the results. Fault trees can be analysed both quantitatively and qualitatively. During qualitative analysis, also known as logical analysis, it is attempted to identify the combination of basic events that are sufficient and necessary for system failure occurrence (top-level event). The outcome of this analysis, known as **minimal cut sets**, is valuable as it highlights any vulnerabilities related to safety (or reliability) in the architecture. For instance, if the failure of a single component is responsible for the failure of the overall system, it should alarm the developers to reconsider the system architecture. This is an important asset and one of the reasons why this analysis method has been adopted in various domains to support other safety-related activities. With the quantitative (or probabilistic) analysis, the engineers are able to calculate the probability of occurrence of the top event based on the probabilities of the intermediate and basic events. This type of analysis helps raise awareness of the system's likelihood of failure, which makes FTA a helpful tool when evaluating a system for safety or reliability. As mentioned earlier, FTA is traditionally conducted manually, although tool support for the automatic synthesis and analysis of fault trees has emerged and has been utilised in various domains.

Below, follows a simple system architecture and its corresponding fault tree. Note that this example does not represent a complete fault tree analysis on the system, but instead showcases how a system failure behaviour can be graphically described using FT nodes. A more detailed demonstration on FTA is provided in Chapter 4. Specifically, Fig. 2.6 features the architecture of a software program in which two functions A and B receive an input and generate identical results R1 and R2. A third function C uses those results to produce the program's output. Function B serves as a redundant component and its result (R2) is used by function C only when there is an omission of input from function A (i.e. R1 omission).

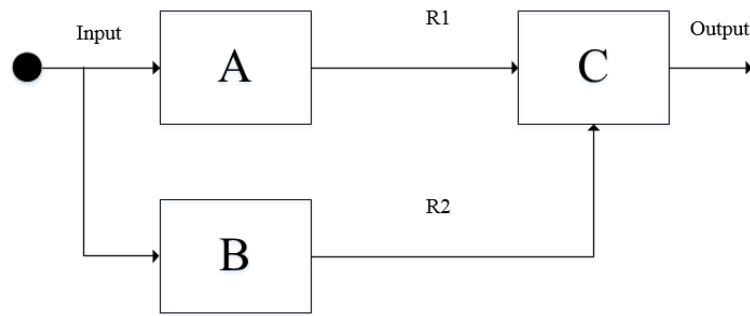


Figure 2.6: The architecture of a simple system

Once the core functionality and dependencies between the elements (i.e. A, B and C) of the program are decided, engineers may proceed with the identification of system behaviour (in this case, failure behaviour). Each component is analysed individually for internal failures (basic events) and other causes that affect its output. Table 2.2 provides the results as failure types and immediate causes.

Table 2.2: Failures and Immediate effects

Component Name	Failures Type	Immediate Causes
A	Omission R1	Failure of A OR Omission-Input
B	Omission R2	Failure of B OR Omission-Input
C	Omission Output	Failure of C OR (Omission-R1 AND Omission-R2)

Finally, Fig. 2.7 depicts the corresponding fault tree of this example. The immediate causes (from the table above) are represented as tree nodes, which are connected with the failures of the corresponding components. The connections are realised with logical gates. The tree is a network of failure propagation that illustrates how individual and relatively minor failures, combined, can lead to system failure. The top event represents the omission of output for function C, which is the system's overall output and the engineer's major concern in this example.

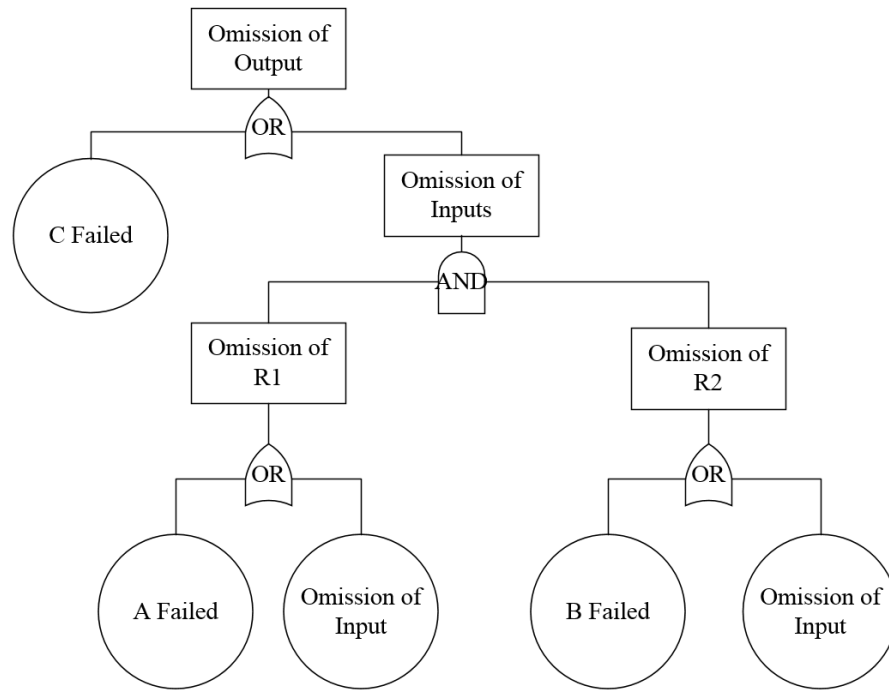


Figure 2.7: The resulting Fault Tree

### 2.3.2 Failure Modes and Effects Analysis

Failure Modes and Effects Analysis (FMEA) is an inductive (bottom-up) safety analysis technique in contrast with FTA. FMEA was introduced as a process for the US military standard MIL-P-1629 (USA Department of Defense, 1949), but it was formally established as an analysis technique during the 70's and published in the up-versioned MIL-STD-1629A standard (USA Department of Defense, 1980). Other versions of FMEA with similar, yet not identical, procedures can be traced back to the 60's as part of the Apollo space program directed by the US National Aeronautics and Space Administration (NASA) (Carlson, 2014).

During FMEA, engineers investigate the architecture for failure modes within lower-level functions or system elements and evaluate how they affect the components in higher architectural levels, up to the system level. Results from FMEA can be further analysed quantitatively. The quantitative analysis is typically realised in cases where the failure modes are provided along with their respective failure rates. FMEA is highly efficient at quickly estimating the overall system safety (USA Department of Defense, 1980). However, it is impractical when determining failure modes for complex

architectures, especially when these arise from failures of multiple components. For example, a system with 1000 components and 2 failure modes would result in almost 500,000 pairs of failures (Xiao et al., 2011). As a result, FMEA is often used as a supplementary analysis technique that supports other processes during the safety assessment. The outcome of FMEA is usually presented in a tabular form; although certain information is often presented in a classical FMEA table, it can be modified to include elements based on specific application needs. Table 2.3 presents the output of an FMEA performed for a generic item, such as a ball-point pen. This example is adapted from (E. P. Plans, 2006) and shows that the severity, detection and occurrence have a specific ranking system that expands between numbers 1 and 10. The criteria for the occurrence are based solely on the failure rates, whereas for the severity are based on the magnitude of the disruption to the production line. Finally, the detection is ranked based on the mechanisms in place and their likelihood to detect failure modes with no specific probabilistic information. The RPN (Risk Priority Number) is the product of these three properties and is helpful for assigning priorities on each failure mode.

*Table 2.3: FMEA table for a generic item (E.P.Plans, 2006)*

Item Part	Function	Failure Mode	Potential Effects of Failure	Severity	Potential Causes of Failure	Occurrence	Detection Mechanism	Detection	RPN	Actions
Outer Tube	Provides grip to the user	Hole gets blocked	Vacuum on ink supply stops flow	7	Debris ingress into hole	3	Check clearance of hole	5	105	Enlarge the tube perimeter

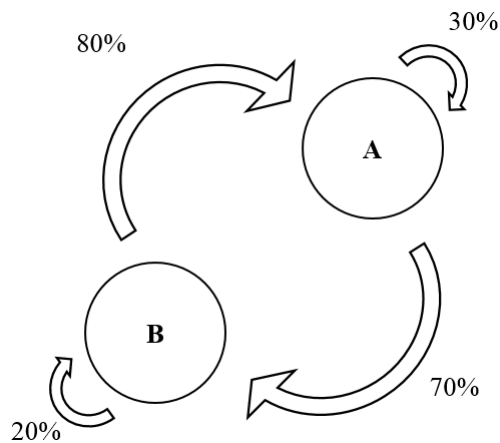
## 2.4 Other Safety Analysis Methods

FTA is not the only option when it comes to system safety analysis; in fact, safety standards suggest that other methods may be used if they are more suitable for the situation (SAE, 1996:35). This section summarises some of the commonly used approaches for system analysis. Note that some of these

have either a primary or a complementary role, whereas others focus on specialised cases such as modelling dynamic failure behaviour.

#### 2.4.1 Markov Modelling and Analysis

Markov Analysis (MA) provides the means necessary for predicting a system's transition between its states, despite the number of these dynamic states, based on a set of probabilistic rules. During the MA, the engineers build a model (i.e. a Markov chain) that captures all the system states, commonly referred to as state space, and connect them with a set of transition relationships (Fosler-Lussier, 1998). With the transition information, knowledge of the current state and, in certain cases, the elapsed time, it is possible to quantify the probability of transition to any particular state; hence, future states depend only on the present state (and sometimes on transition time) and not on the history of transitions. Note that the state space is not restricting in volume or content, meaning that it may have an infinite amount of states and states may be numbers, letters, conditions, performance values and so on. Although any size space is possible, most applications in various disciplines and even the initial theory behind MA focus on finite state spaces. A simple example of MA can be shown in Fig. 2.8.



*Figure 2.8: A simple MA model*

This example features two states, A and B, in the state space and four possible transitions (the system may remain in the same state). In this scenario, the probability of transitioning from state A to B is

70% whereas remaining in state A is 30%. On the other case, moving from state B to A, the probability is 80% whilst preserving state B is 20%.

Analysts frequently prefer to use transition matrices, instead of diagrams, when dealing with systems that involve multiple states. Each state is included, once as a row and once as a column, in the matrix and the resulting cells describe the probability of transitioning from the state of the row to the state the column or vice versa (representing the arrows in a diagram). Table. 2.4 demonstrates the example from Fig. 2.8 in a matrix format (transition is from row to column).

Table 2.4 Previous example in tabular form

States	A	B
A	0.3	0.7
B	0.8	0.2

In safety applications, states often describe the ability of a system to successfully or not (implying failure) perform an operation. Accordingly, transitions may serve as the probabilities of either failure or repair rates. For example, Fig. 2.9 depicts the Markov model of a system with two components, A and B, and 4 different states.

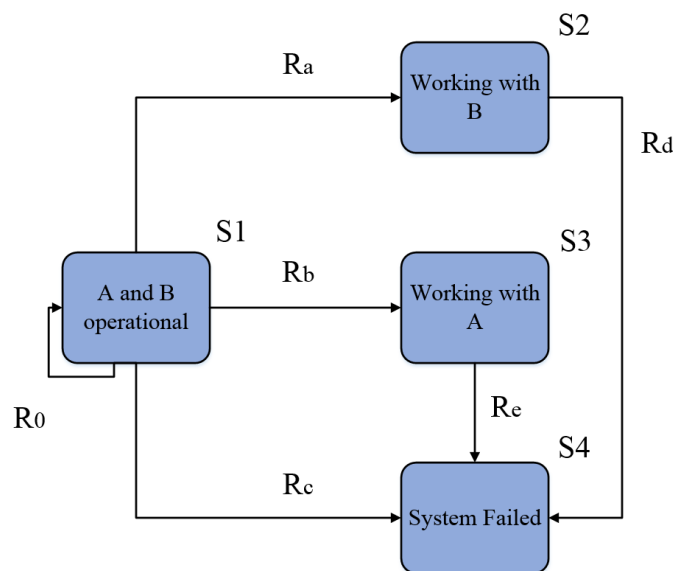


Figure 2.9: Markov model of an abstract safety-related system

State S1 represents the normal operation of the system (i.e. components A and B operable), whereas states S2, S3 and S4 represent the situations where the components A, B and A&B fail, respectively. The arrows indicate the transitions between the states and the parameters  $R_a$  to  $R_e$  are the corresponding probabilities. Finally,  $R_0$  is the probability of the system preserving its normal operation, state S1, which is calculated by subtracting the sum of the remaining scenarios (given that transition rates remain intact).

Generally, the rate of change for each state can be mathematically described and calculated with a differential equation. Complex models often involve numerous states that are dependent on multiple transition rates. As a result, solving a large set of differential equations (typically with Laplace transforms) becomes a difficult and strenuous task for the analysts without the aid of automated tools (Markov solvers) such as HARP or SHARPE (SAE, 1996: 126). Another issue when analysing complex systems with Markov methods is that the diagrams can be too extensive and, apparently, there is no support for a systematic development of the transitions and the model. Hence, the construction of the Markov model is bound to the experience and competence of the analyst, which may end up with inconsistencies from one individual to another. MA's limitations, when compared with other methods in the safety sector, suggest that MA should be used as a complementary method. In fact, MA is very effective when analysing architectures with dynamic characteristics (systems with failure rates dependent on the state) and operator-dependent state transitions (SAE, 1996:109). In these cases, to tackle the issue of MA models being too large and complex, safety standards advocate that the analysts should use different techniques, such as state aggregation or model truncation, for reducing the size. In the former, the analyst is able to aggregate two or more states with equal transition rates into one state. With model truncation it is possible to selectively discard parts of the model where the probabilities tend to diminish (SAE, 1996:123). Finally, there is another technique for reducing the size of a Markov chain known as Hierarchical Modelling. Following this method, the model is divided into separate sub-models; however, the sub-models must be statistically independent (i.e. occurrence of events in one does not influence the other). For example, assume a

system with two components A and B. In this case, the system fails if any of the components A or B fail. It is possible to build Markov models separately for each of the component architectures and calculate their probability of a failed state. The results, for this specific scenario, can be combined using the formula below:

$$P_{systemfailure}(CompA \text{ or } CompB) = P_{compA} + P_{compB} - (P_{compA} * P_{compB})$$

#### 2.4.2 Failure Propagation and Transformation Notation

Failure Propagation and Transformation Notation (FPTN) is a compact graphical notation that allows the modelling of system failure behaviour with the use of standardised entities – the modules. It was developed at the University of York in the 1990s as part of an integrated set of methods used in safety analysis of software systems. The modules, used in FPTN, correspond to functional elements within the target system and can be structured hierarchically to describe system failure behaviour. Modules are characterised by three main attributes; a name, the severity level and the information related to recovery plans. In addition, modules have input and output ports that support the linking with other components and underpin the propagation of failures throughout the architecture. All module-associated failure modes and the relationship between inputs and outputs are encapsulated within the module itself. The failure behaviour is ultimately determined with the use of established analysis techniques, notably FTA and FMEA (Fenelon & McDermid, 1992; Fenelon & McDermid, 1993). In fact, FPTN was created with the intent to act as a bridge between inductive and deductive methods and provide the benefits from both types under a common framework. The fundamental element of FPTN, a module, is shown in Fig. 2.10 below adapted from (Fenelon & McDermid, 1992:24).



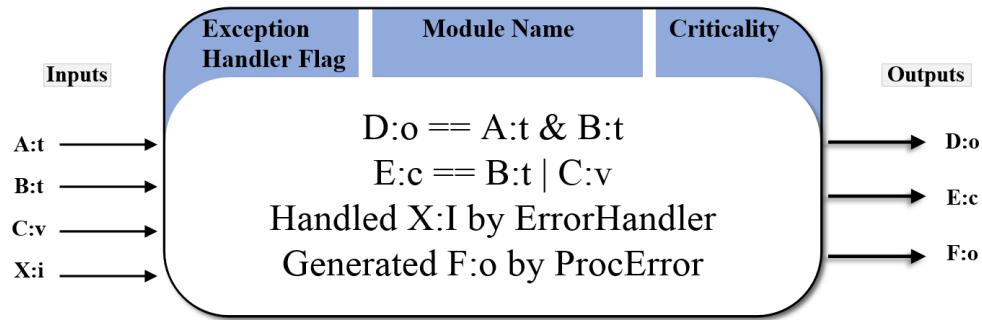


Figure 2.10: An example of FPTN diagram (Fenelon & McDermid, 1992:24)

This is an abstract example of a module to showcase its basic features as prescribed by the FPTN. At the top of the rectangle exist the basic properties of the module, its name, an exception handler (if applicable) and the criticality level. The exception handler is responsible for managing any failure modes that the module can unconditionally prevent from propagating. The arrows represent the information flow (i.e. input, output and propagation) and can be annotated with the corresponding failure type. Specifically, inputs display failure modes that affect the component, whereas the outputs represent the failure modes that the component sends to its connected neighbouring elements. Arrow ‘B’ for example, encapsulates a “timing” failure that will probably cause the input to arrive at the component at an inappropriate time. The main body of the box includes all the failure modes generated internally (independent of external conditions) as well as the relationship equations between inputs and outputs. FPTN is usually designed in parallel with the system architecture; hence, allowing the system model to evolve based on the feedback gathered from the analysis in an iterative cycle. The combination of information collected from the utilisation of both inductive and deductive analyses and the solutions provided to mitigate issues of classical analysis techniques make this method a valuable tool. However, FPTN is not without its own faults. In particular, as development proceeds, the generated models (i.e. system architecture and the FPTN failure model) become more detailed and their detached nature makes it difficult to trace any changes between them, which could result in substantial differences in later stages.

### 2.4.3 Fault Propagation and Transformation Calculus

Fault Propagation and Transformation Calculus (FPTC) is a method that allows for a modular description of systems, both hardware and software components, and provides the tools necessary for compositional analysis (Wallace, 2005). Although FPTC shares similarities with FPTN, it attempted to resolve FPTN limitations by establishing a connection between the architectural design and the FPTC expressions. Specifically, FPTC establishes a structure of modules that correspond to the architecture and each of these modules captures the failure behaviour of its component counterpart. Note that modules are not strictly system components but anything that encapsulates data or control flow in any shape or form. Based on (Wallace, 2005), this promotes simplicity, since the analysis of components in isolation is substantially easier than the evaluation of the whole system. Once the modular representation is set and the correct FPTC expressions have been assigned by the analyst, the system-level failure behaviour can be analysed with the use of the semantics and the propagation algorithm provided by the method. The algorithm is inspired by optimisation techniques used in compiler development and has been designed to eliminate any cycles within the propagation model. FPTC syntax has been formalised and the method has been employed in various industrial case studies such as engine controllers (Ge et al., 2010). Despite its success, FPTC does not provide the tools for quantitative analysis and therefore the probability of failure behaviour, important for some domains, cannot be computed. Fig. 2.11 shows the basic symbols that designate the propagation logic in FPTC (each symbol has certain transformations during the propagation). The failure classes provided in FPTC are almost identical to the guide words found in Hazard and Operability (HAZOP) studies.




		
signal	channel	pool
early -> * omission -> late commission -> value	early -> * omission -> late commission -> late * -> late	early -> * omission -> stale value commission -> * late -> stale value

Figure 2.11: FPTC Symbols as summarised in (Wallace, 2005:5)

#### 2.4.4 Dependence Diagrams

The Dependence Diagrams (DDs) is another approach for representing diagrammatically the failure behaviour of a system architecture and performing probabilistic analysis. The role of this method in the overall safety assessment of a system is identical to that of FTA and these two could be used interchangeably (SAE, 1996:104). The key difference of DD appears to be in its structure since there is no use of intermediate entities for logic management; instead, it utilises the structure itself to simulate logic. Each DD symbolises a failure condition (top event) and a typical structure features a series of interconnected blocks, where each block represents a fault event of a system component and encapsulates its failure properties (e.g. failure rate). The failure propagation flows from left to right, meaning that the inputs are always on the left and the outcomes on the right of each block. Management of logic in DD is demonstrated with two basic configurations. The entities in series represent the OR situations, whereas parallel configurations stand for the AND logic. As a result, any redundant entities added in the architecture, as a failure mitigation measure, can be shown by aligning the redundant components in parallel to their respective counterparts. Fig. 2.12 provides an example with the DD structure.

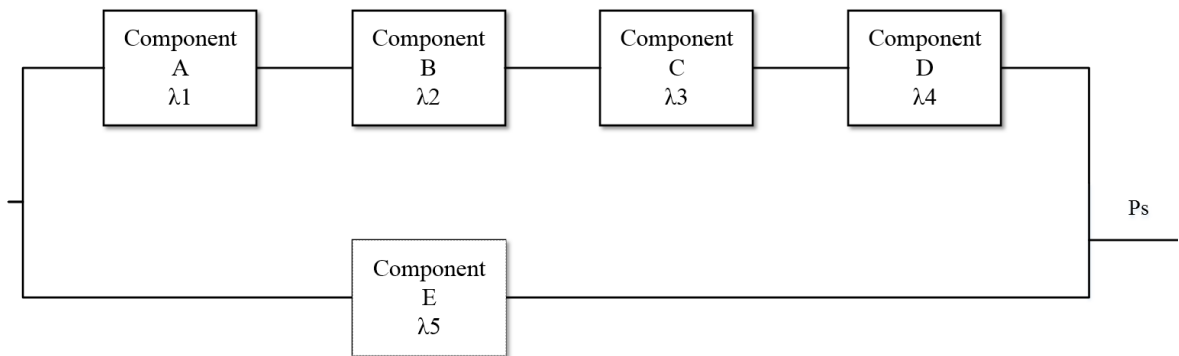


Figure 2.12: Example of a simple DD structure

The variables  $\lambda_1$  to  $\lambda_5$  represent the probability of failure for each component. The blocks that exist in the same row represent the elements that can affect the overall system only collectively (AND). On

the other hand, all parallel configurations are capable of causing a system failure individually. In this example, the probability of system failure,  $P_s$  can be calculated as:

$$P_s = (A\lambda_1 + B\lambda_2 + C\lambda_3 + D\lambda_4) * (E\lambda_5)$$

The structure with DD is seemingly simple; however, it can easily grow complex as certain failures are often used multiple times within the same structure (e.g. for redundancy purposes). In such case, Boolean algebra operations are required to compute the probability and generate minimal cut sets.

#### 2.4.5 Dynamic Fault Trees

Dynamic Fault Trees (DFTs) is a widely used methodology for quantitative analysis of dynamic systems. It was firstly introduced in (Dugan et al., 1992) as a gate-based approach, but DTFs as a concept have also been modified and proposed in (Cepin & Mavko, 2002) as an event-based approach. Due to the success of the version created by Dugan et al., this section focuses on that approach. The DFTs extended on the classical FTA with the inclusion of new gates and the integration of Markov models with fault trees. With this integration, DFTs enable the automatic construction of Markov models, which otherwise is a tedious process, from the tree structures while also unlocking the potential to analyse systems' sequence-dependent failure behaviour with fault trees. As a result, DFTs are capable of analysing systems with more complex behaviour that may depend on sequence events and standby mechanisms. The 'temporal' gates introduced in DFTs are summarised below:

1. **Functional Dependency Gate (FDEP)**, which incorporates the notion 'trigger event'. By doing so, DFT supports the modelling of components dependent on events. For instance, if one component fails and is connected with other components with FDEP gates, all the connected components will fail too.
2. **Spare Gate**, which allows for selective redundancy in a given architecture. Assuming 4 components are connected on a spare gate, the secondary component activates only when both the primary and all preceding secondary components fail in sequence.
3. **Priority-AND Gate**, a variation of AND-gate that allows the output to occur only when all the events happen in a specified order.

4. **SEQ Gates**, which explicitly state under which sequence the events will occur or not.

The graphical representations of these gates are shown in Fig. 2.13, reproduced from (Faulin et al., 2010).

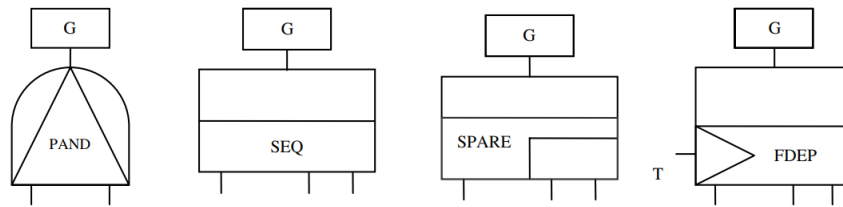


Figure 2.13: The graphical representation of DTF gates (Faulin et al., 2010:42)

## 2.5 Model-Based Safety Analysis

Model-Based Safety Analysis (MBSA) is an alternative safety analysis paradigm, firstly discussed under this term in (Joshi et al., 2006). The term has been used since to describe much of the work that centres safety analysis around models of a system. The motivation behind MBSA was that all the preceding techniques, such as the FTA and FTPN, were analysing and documenting system behaviour separately from the original architecture. Despite the frequency of the updates within the various informal models, generated during development, important information produced from the assessment processes were at risk of being disregarded and not reflected on the final system architecture. In addition, earlier approaches were typically performed manually and proved to be more inefficient as the size and integration of modern systems increased. Finally, pre-existing traditional techniques, such as the FTA, were highly subjective. In other words, the accuracy and correctness of said analyses were dependent on the experience of the analyst. Maintaining clear objectives, having a correct view about the system state and understanding the development phases were key aspects; thus, to ensure a good outcome, consensus among stakeholders was mandatory before approval of results. This could prove to be time-consuming and lead to an asynchronous relationship between safety assessment and system design. Any efforts to restore the balance between the two in later stages of development (e.g. manufacturing or production stages) could incur significant additional costs.

In response, MBSA inherits the merits from model-based development and centralises the development around a semi-formal model that links assessment properties with system properties and

promotes a more agile and iterative development. To achieve this, MBSA expanded the nominal model with failure information into an extended model, referred to as ‘fault model’. Further, with the inclusion of formal models, it encouraged developers to look towards the semi-automation of safety assessment procedures. That alone provides substantial benefits as it supports the mitigation of issues related to manual approaches.

Methods that employ the MBSA paradigm can be classified into two major categories, the Compositional Safety Analysis (CSA) and the Behavioural Safety Analysis (BSA) (Sharvia & Papadopoulos, 2011; Möhrle et al., 2016), discussed in the following sections.

## 2.6 Compositional Safety Analysis

CSA methods typically annotate system elements with failure logic and then derive, via composition, the system-level failure behaviour. Methods that employ this paradigm typically follow the ensuing steps to perform safety assessment activities. First, system functionality and the corresponding requirements and constraints are defined that support the construction of a preliminary architectural model. As development progresses, the requirements are further refined and decomposed into lower architectural elements (components) in an iterative manner. The analysts are able to annotate these components with the local failure behaviour and perform safety assessment techniques, such as FTA and FMEA. This supports identification of functional hazards and their probability of occurrence, as well as mitigation means that could lead to further requirements and potentially design modifications. Note that most of these methods can be deployed at any stage throughout development.

There is a plethora of methods and tools that use the CSA paradigm with the most notable being summarised in the upcoming sections.

### 2.6.1 Component Fault Trees

Component Fault Trees (CFTs) is proposed in (Kaiser, 2003) as an extension to the FTA by modifying the notion of components and introducing hierarchical decomposition. In traditional FTA, modularisation is only applicable on the tree structure, whereas CFTs support this idea with regards

to the system architecture. In the CFT method, components are referred to as ‘technical components’ and resemble actual system components represented with extended fault trees. Apart from the basic events and logical gates, the components in a CFT include a set of ‘ports’, namely input and output, that allow interconnectivity and collectively this network forms a higher-level system architecture. Each component’s tree is considered a function where the logic propagation to the output can be computed based on the logic received in the input port and the internal events. Components can still be evaluated separately and stored as component libraries for later use. The latter promotes concurrent development and safety assessment of system elements. Moreover, any qualitative or quantitative analysis techniques, typically used in FTA, are still applicable in CFTs. Despite the name of the method and its close relationship to the FTA, CFTs are often structured as directed acyclic graphs, also known as Cause Effects Graphs (CEGs). Following this logic, the analyst examines any repeated events and places them only once in the structure. Fig. 2.14 shows an example, adapted from (Kaiser, 2003:5), of how a system can be designed when using FTA and CFTs. On the left, the system is represented in a classical fault tree and on the right, it is depicted as a CEG diagram.

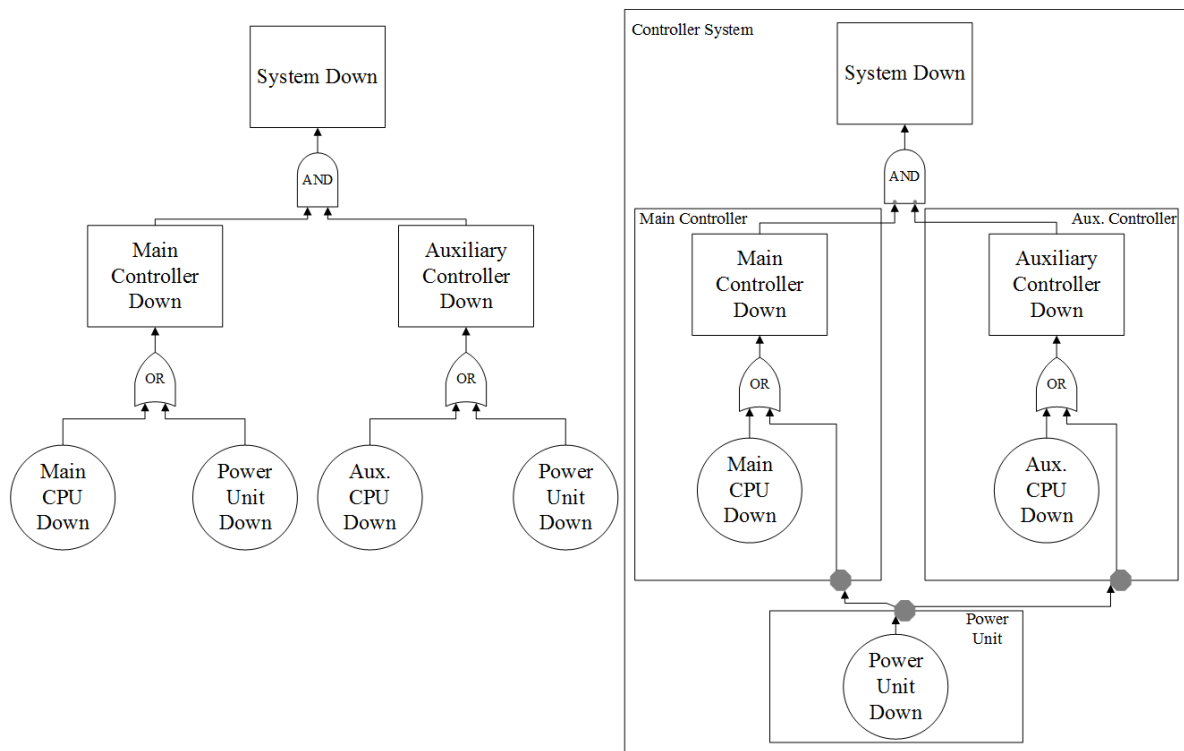


Figure 2.14: Differences in structure between FTA and CFT (Kaiser, 2003:5)

In this graph, the occurrence of the top event (System Down) depends on the simultaneous occurrence of the intermediate events (Main Controller Down and Auxiliary Controller Down). In response, these components will fail either if their basic events (i.e. CPU failure) or the power unit failure occur. Note that in the standard fault tree on the left the power unit appears twice, whereas after consideration it was concluded that this unit is shared among the controllers; therefore, it was separated as its own component and is represented only once in the CFT.

### 2.6.2 State Event Fault Trees

State Event Fault Trees (SEFTs) is another compositional analysis technique proposed in (Kaiser & Gramlich, 2005) and later in (Grunske et al., 2005). SEFTs were developed as an extension to the classical FTA and the primary objective was to take into consideration complex system (e.g. hardware or software) behaviour. Specifically, this technique employs the concept of ‘states’ and ‘events’ to encapsulate sequences of actions, utilises Markov models to evaluate failures probabilistically and also manages to maintain the simple visualisation of ‘cause propagation’ from fault trees. Under this method, states are defined as conditions of a component (or system) that last for a specified period of time, whereas events are abrupt occurrences that are responsible for declaring transitions between the different states. A component can only be in one state at a time for the specific structure. In principle, SEFTs are closer to ‘state machines’ rather than combinatorial models; however, the intent was to combine the two towards the emergence of better analysis performance for subsystems. Like other state machines, the states and events in SEFTs do not always symbolise failures. Further, SEFTs utilise various logic gates including, yet not restricted to, the AND, OR and NOT gates as well as variations such as the History-AND gate. The flow of information between components is achieved via the ports, which in addition to fault trees can also accept Markov models and other diagrams. Finally, SEFTs require the use of Petri nets for the analysis. Petri nets provide the means necessary for modelling systems with finite state spaces and concurrency features. Under the Petri net domain, the components (fault tree, Markov chains, etc.) are merged into one flat model and processed through



specific analysis tools. Fig. 2.15 is reproduction from (Kaizer & Gramlichm, 2004:9) that depicts the basic structural elements of SEFTs.

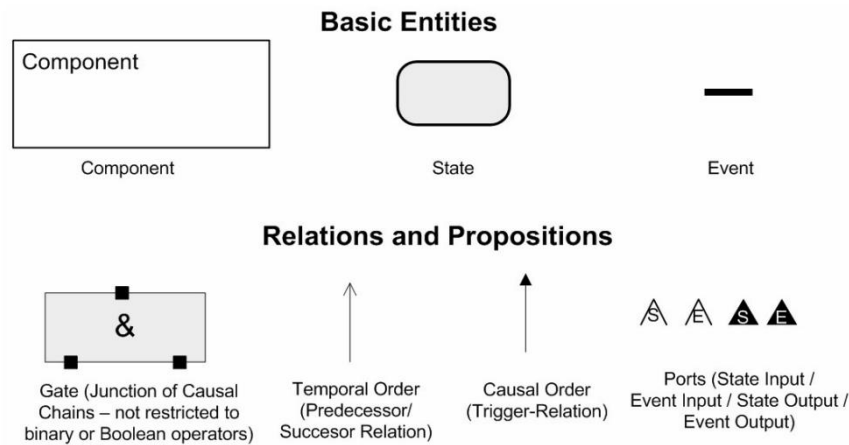


Figure 2.15: Basic elements of SEFTs (Kaiser & Gramlich, 2004:9)

### 2.6.3 Hierarchically Performed Hazard Origin and Propagation Studies

Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) is a model-based safety analysis method proposed in (Papadopoulos and McDermid, 1999). The primary feature of this method is the annotation of system elements with failure behaviour, via logical expressions similar to those used in FPTN, and the automatic synthesis of fault trees and FMEA tables that allow safety assessment of complex system architectures. The structure of HiP-HOPS comprises the model (system architecture), subsystems that act as groups of lower-end entities and components that often represent system functional elements. The connection between components is realised with the use of ports and lines. Ports act as the input and output interfaces of the component whereas lines carry the flow of data between each connection. The basic method for safety assessment consists of three major phases which are summarised below:

- **Modelling phase**, where the engineer builds the system model with the basic structural elements described in HiP-HOPS. This process requires a modelling tool compatible with HiP-HOPS. Once the model is created, the analyst annotates locally each component of the system with failure

behaviour information. More importantly, all the information is encapsulated within the actual system model; thus, any safety assessment activities are performed in parallel with the system development.

- **Synthesis phase**, where the HiP-HOPS engine examines the local failures of components and the provided architecture to establish how component-level failures propagate and how they cause system-level failures. With this information, the algorithm generates a fault tree for each system failure, and after applying various techniques to simplify and enhance (e.g. to avoid circular logic) the trees, it synthesizes the final fault tree for the whole system.
- **Analysis phase**, where the generated fault trees are analysed both qualitatively and quantitatively. The results from the analyses produce the minimal cut sets and the FMEA table. The minimal cut sets are first subjected to logical reduction techniques for simplification purposes and then undergo further qualitative analysis to obtain useful information for the system in question. Finally, the FMEA table and the list of minimal cut sets are provided in XML format and can be also viewed via a web browser.

Fig. 2.16 demonstrates the basic workflow in HiP-HOPS via a concise diagram.

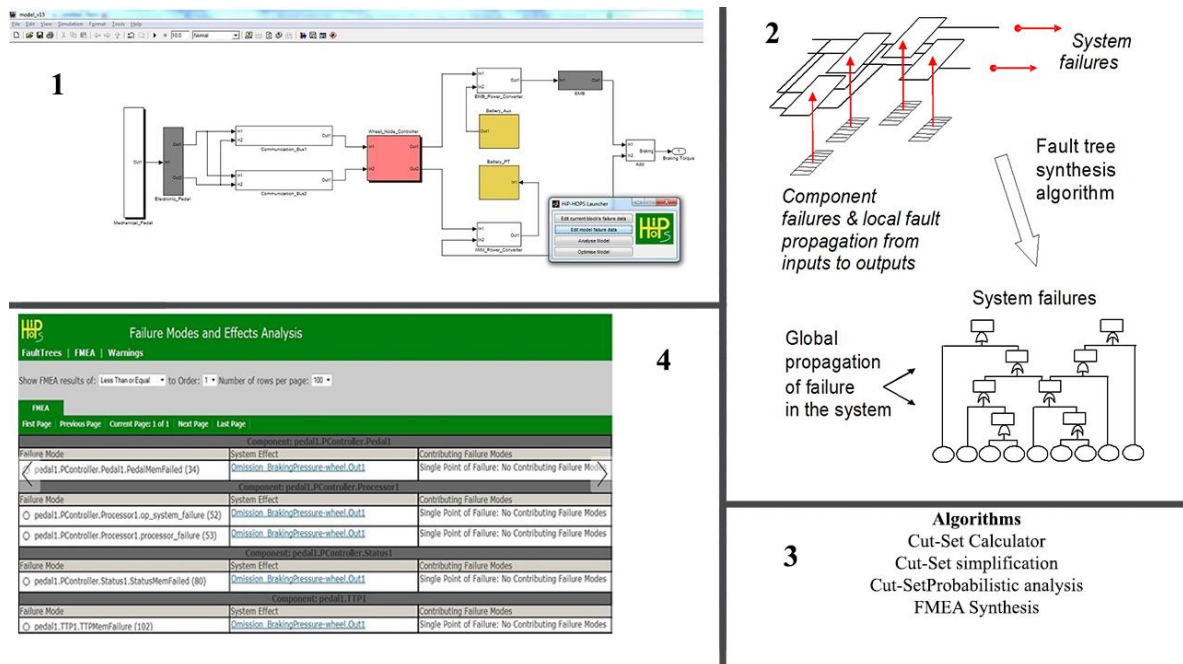


Figure 2.16: Basic workflow in HiP-HOPS

The top-left part of the figure (label 1) illustrates partly the modelling and annotation of the system architecture. The top-right section (label 2) depicts the synthesis of fault trees from the given architecture whereas the bottom-right section indicates the algorithms that take place during the analysis phase. Last, the bottom-left part provides the produced FMEA table as presented by the tool.

The initial framework of HiP-HOPS was implemented under a tool developed by the University of York known as the Safety Argument Manager (SAM). Since then, HiP-HOPS has established autonomous tool support and has been extended to support dynamic fault trees (Walker & Papadopoulos, 2006), multi-objective genetic algorithms for architectural optimisation (Parker, 2010), temporal FTA (Walker, 2009) and optimal allocation of safety requirements (Azevedo, 2015). The multi-objective optimisation allows the exploration of large design spaces and promotes design strategies that optimise the cost and reliability of a system. The capability of HiP-HOPS for automatic analysis of complex models, with multiple failure modes, enables the genetic algorithm to produce system architectures of non-trivial (e.g. parallel) configurations. The temporal FTA with the use of Priority-AND gates is capable of identifying whether or not a specific sequence of events causes the system failure. HiP-HOPS can be utilised either as a safety analysis engine with other modelling tools, such as MATLAB Simulink, or can be integrated with other assessment methods under different domains. HiP-HOPS, as a safety analysis method, is flexible and scalable and has been employed in various industrial and research projects over the years. The main limitation of HiP-HOPS is that the analysis focuses mainly on electromechanical systems and has no support for failures that occur from human activity (i.e. bad decisions by the operator).

#### 2.6.4 Failure Propagation and Transformation Analysis

The Failure Propagation and Transformation Analysis (FPTA) is a model-based compositional analysis technique proposed in (Ge et al., 2009) that attempts to address the issues found in the preceding FPTN and FPTC. As discussed earlier, the inability of FPTN to connect its proposed structure for failure analysis with the system architecture exposed the system development to inconsistencies. On the other hand, the lack of support from FPTC for quantitative analysis denied

the developers the ability to calculate the probability of specific failure behaviours which could lead to suboptimal designs. FPTA expands on the concepts of FPTC with the integration of probabilistic model-checking. Thus, on top of the benefits from its predecessors, FPTA enables system verification with formal methods as well as quantitative analysis (probabilistic).

FPTA achieves failure behaviour modelling via the utilisation of the notation provided by FPTN and provides a suitable framework for the application of model checking techniques using the PRISM model-checker. In FPTA, a system is defined as the top-level element and every other entity (i.e. module or subsystem) is referred to as component. The latter has a main body that represents the logical function as well as input and output ports. The components are linked with connectors and together compose the interaction model. One of the major features introduced in FPTA is the notion of ‘mode’, which provides an additional layer of detail for the components by describing the way they should be looked at. In fact, components in FPTA are expected and therefore designed to behave without fault; hence, their default mode is characterised as ‘normal’ and involves the non-failure model. However, one component is also designed for all its modes (normal and failure modes). Fig. 2.17, a reproduction from (Ge et al., 2010:4), demonstrates how a generic processor would be modelled in FPTA.

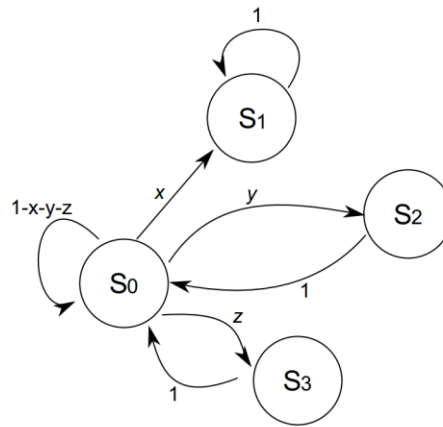


Figure 2.17: Example of a generic processor (Ge et al., 2010:4)

In this example, each entity S0 to S3 indicate a different state (mode) of the same processor. The S0 represents its normal mode whereas states S1, S2, S3 encapsulate the potential failure modes of crash

failure, timing failure and value failure, respectively. The general format for expressing the failure behaviour of the component is:

$$input\ state \rightarrow output\ state, probability$$

The annotation includes the corresponding ports, failures classes and probability rates. Fig. 2.18 provides the propagation of failures, as annotated in FPTA, from an example presented in (Ge et al., 2009).

<b>Input</b>	<b>Output</b>	<b>Probability</b>
Input.omission	output.omission,	0.0001
Input.value	output.omission,	0.0001
Input.normal	output.omission,	0.0001
Input.omission	output.omission,	0.9999
Input.normal	output.normal,	0.9999
Input.value	output.value,	0.9999

*Figure 2.18: An example of probabilistic data inclusion in FPTA*

## 2.7 Behavioural Safety Analysis

The Behavioural Safety Analysis (BSA) is a class of methods that follow the MBSA paradigm. In BSA, faults are inserted deliberately into the formal specification of the system and the analysts examine what are the effects on the system behaviour. BSA techniques employ formal models in conjunction with the model checking method in order to formally verify whether or not system properties (e.g. safety requirements) satisfy the system specification and also to enable evaluation of dynamic system behaviour. These formal models are precise mathematical representations of the system model and its specification. Most methods that follow this paradigm often provide their own language to express these formal models within their framework and are capable of solving the problem algorithmically. Apart from the benefits of BSA methods to automatically perform formal verification and evaluate dynamic behaviour, they provide the means to differentiate failures based on their duration (i.e. temporary or permanent) and also capture whether a time-gated sequence of failures is responsible for a certain event. Naturally, this paradigm has its own weaknesses. For instance, formal methods can be time-consuming when analysing systems (e.g. software) with a large

architecture and extensive specification space (input/output), even when automation is provided. Further, formal methods require mature designs and specifications that often are available only during later stages of development, which is not ideal for safety-critical systems. As a result, they are typically deployed after other safety assessment methods.

### 2.7.1 FSAP/NuSMV-SA and xSAP

FSAP/NuSMV-SA is a platform for safety analysis firstly introduced in (Bozzano & Villafiorita, 2003) and further evaluated via an industrial application in (Bozzano & Villafiorita, 2006) as part of the ESACS project. The main goal of the method was to enhance the development cycle of complex safety-critical systems. To do so, it incorporated many analysis techniques, including but not restricted to model checking, automatic synthesis and analysis of fault trees as well as other analysis artefacts, such as minimal cut sets and property verification results. The platform consists of two major elements; a) a graphical user interface (FSAP), which is designed to be user-friendly for operators with limited knowledge of formal methods and b) the engine (NuSMV-SA) that inherits many of the features provided in the NuSMV model checker. The method's basic phases are summarised below:

- **Model capturing**, where the formal system model is defined in the NuSMV input language (text-based). The model can be defined either from the view of the designer (nominal model) or the safety engineer (formal view of the system that focuses on the safety traits).
- **Failure Mode Capturing and Model Extension**, where the engineer provides the specification under which individual components may fail (i.e. failure modes). The failure modes can be stored in a data structure, referred to as 'Generic Library of Failure Modes'. At any point, it is possible to inject these failure modes into the model; thus, creating the extended system model (ESM).
- **Safety Requirements Capturing**, where both the designer and the engineer define the functional and safety requirements, respectively, under which the system is evaluated in later stages. The requirements can be predefined in earlier projects or written in a formal notation (e.g. temporal logic).

- **Model Analysis**, which incorporates formal verification tools (in this case method's model checker) to assess the system based on the previously defined requirements. This phase involves two verification tasks. In the first task, the tool checks the validity of system properties and generates a counterexample trace whenever a property is not verified. Consider an example where a system property, based on the specification, was supposed to not change its value, but has failed in doing so. In that instance, the tool would create an example (in plain text) that indicates exactly what caused the violation. During the second task, the tool focuses on the system safety requirements and produces all the minimal cut sets that are responsible for any violation of safety requirements.
- **Results Extraction and Analysis**, where the results from the analysis phase are extracted and processed to be presented in compatible forms such as plain text, XML or tabular structures. This allows the results to be used with other well-established tools in safety engineering.

One of the main limitations of this platform, argued in (Joshi et al., 2006), involves the generation of fault trees with a 'flat' structure. This reduces their value as it is more difficult to review the structure and assess the situation. Moreover, the method does not support probabilistic assessment. Finally, FSAP provides support only for a finite number of states and even in this case the options may still be numerous and pose performance issues.

Through collaboration between the FBK (Fondazione Bruno Kessler) group and the Boeing Company, a new platform for safety analysis, closely related to the FSAP, has been developed and presented in (Bittner et al., 2017). xSAP is developed by the FBK and while it retains all its predecessor's features, it also extends in many aspects. Firstly, xSAP supports both finite and infinite-state systems and provides additional libraries for the definition of fault modes. Further, it has expanded its capabilities towards the generation of counterexamples with the inclusion of new formal techniques such as the SAT-based bounded model checking (BMC). This platform also supports the probabilistic assessment of fault trees and common cause analysis, which are often required by safety standards for certification of systems. xSAP has been integrated as an engine in various industrial tools, such as the COMPASS (Bozzano et al., 2014) and AUTOGF (Alana et al., 2012). Finally, it

has also been deployed in many research projects, both in academic and industrial environments and was well-received in applications from the European Space Agency (ESA) and Boeing.

### 2.7.2 AltaRica

The AltaRica is a modelling language initially proposed in a PhD thesis (Point, 2000) and later was refined and developed by the collaboration between the LaBRI and ELF research groups (AltaRica Project, 2020). It is supported by two main software tools, the AltaRica Studio and the ARC, which provide the necessary framework for system modelling. Under this framework, models are specified as hierarchies of nodes and sub-nodes. Each node can capture either the nominal or failure behaviour of system components. They also encapsulate different states, transition events and interfaces (Shaojun & Xiaoxun, 2014). Once the network of nodes (model) is complete and the states and transitions are defined, it is possible to assess how the system reacts to the events and why its state is affected. The whole process for system analysis, using the AltaRica language, can be broken down into the four stages summarised below:

- **System Modelling:** During this phase, the extended system model is composed forming a hierarchy of nodes. Each node is annotated with states, events, transitions and flow variables using the formal syntax defined by the AltaRica language. Flow variables in this context are dependent on the states and act as component interfaces for data communication between other components.
- **Formal Safety Requirements:** This phase is where the safety requirements are formalized with the use of temporal logic operators. The requirements can potentially be stored in a library for future use.
- **Graphical Interactive Simulation:** With the use of an interactive graphical simulator, engineers can conveniently examine the system architecture for failure occurrences. For instance, the practitioner can choose an event and the simulator will compute the corresponding state.
- **Safety Assessment:** At this stage, it is possible to re-compile the system model into alternative forms. AltaRica supports a number of compilers able to produce fault trees for reliability analysis, stochastic Petri nets for performance analysis as well as automata, which are used in formal verification techniques (e.g. model checking).



Fig. 2.19 presents how a switch could be modelled in the AltaRica framework, adapted from (Point & Rauzy, 1999:4). On the left part of the image there are the state/flow/event annotations and on the right part the system model is depicted.

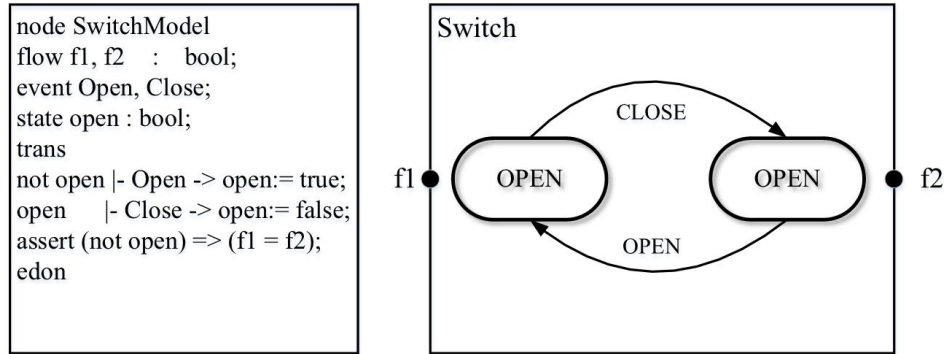


Figure 2.19 A switch representation in AltaRica (Point & Rauzy, 1999:4)

The ability to transform the results into lower-level (in terms of formality) structures, such as fault trees, renders AltaRica into a flexible tool that can be used as a support language in various safety assessment methods. Relatively recent research in (Shaojun & Xiaoxun, 2014) succeeded in establishing a technique that extracts failure logic information directly from AltaRica models and based on that synthesises fault trees. This research provides a foundation for a generation of automated analysis tools based on AltaRica. Useful as it may be, it does not provide enough ground for the allocation of abstract requirements, such as SILs. It was possible to integrate Altarica for the safety assessment part of the proposed method, but it would require significantly more work to develop the appropriate framework.

## 2.8 Other Notable Tools

This section provides a brief summary on other notable tools related to safety analysis that could have been used directly for the development of the supporting tool of the proposed method.

### 2.8.1 Galileo

Galileo (Sullivan et al., 1999) is a software tool for manipulation and analysis of dynamic fault trees. It implements the DIFTree (Dynamic Innovative Fault Tree) methodology which allows the analysis

of both traditional and temporal fault trees. The modelling of fault trees can be realised either via a text editor or via the MS Visio drawing tool, and Galileo provides a function to transform graphical representations into textual and vice versa. Galileo may not be a major tool in the safety-critical sector due to its lack of support and weaknesses (e.g. fault trees cannot be generated from system models), but as a research prototype, it has been utilised as the base for various research projects (Dugan et al., 2000).

### 2.8.2 DFTCalc

DFTCalc is a relatively recent software solution capable of modelling and analysis of dynamic fault trees. The tool allows the engineers to assign probability distributions to basic events and simulate how failure behaviour changes over time. The framework also incorporates stochastic model checking techniques to systematically explore the state space of stochastic systems. This allows the introduction of dynamic fault trees with basic events that are statistically dependent with each other; however, there is no support for any sort of repair mechanisms within basic events. On a positive note, the methods provided within the tool seem to address the issues occurred in time-dependent reliability analysis due to complexity of modern systems (e.g. large state spaces) (Arnold et al., 2013).

### 2.8.3 FaultTree+

FaultTree+ was firstly introduced as a tool for modelling and analysis of fault trees. Due to its efficient algorithms for minimal cut set generation from complex fault and event trees and its simplicity of use, it was popularised and commercialised. Isograph has integrated FaultTree+ with other software packages into a suite of tools for reliability, safety and availability problem solving (Isograph, 2017). This framework, currently referred to as ‘Reliability Workbench’, allows the analysts to model and analyse fault trees, Markov chains, event trees and reliability diagrams and is capable of producing FMEA tables and other evidence artefacts often required when developing systems based on the ISO26262 and MIL-STD-1629A safety standards. Fig. 2.20 is a reproduction of a fault tree structure created in this tool as found in (FTDSolutions, 2019).

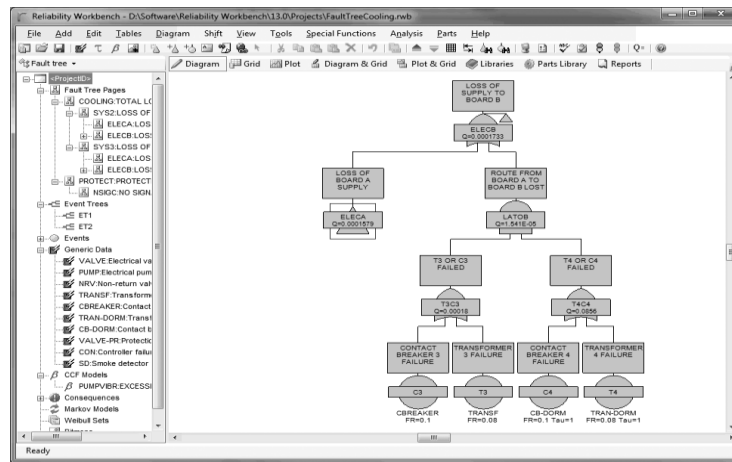


Figure 2.20: A fault within FaultTree+ (FTDSolutions, 2019)

Despite its capabilities, access to the source code of the tool, that would allow for a smoother integration, was limited compared to other more academic solutions.

## 2.9 Safety Cases

### 2.9.1 An Introduction to Safety Cases

Many disasters have occurred in the past decades as a result of industrial accidents. Every incident would sound the alarm bell for engineers to investigate the cause and assess the situation; hence, the development of safety analysis methodologies was emerged. Initially, major effort was placed in the collection of data from the accident, evaluation and formulation of a best practice ‘code’. Once the root cause was identified, a typical response was to reverse engineer the problem and define feasible solutions (Rushby, 2015). Following this regime, governments examined the most successful practices and formed a set of instructions which would be distributed as regulations for various industries to follow. At first, this strategy appeared to be effective and certainly was the first step in the right direction. However, a number of serious accidents, most notably the 1957 fire at Windscale nuclear plant, the 1974 Flixborough chemical plant explosion and the 1998 Piper Alpha oil platform explosion, alarmed the public about the effectiveness of the existing safety management practices (Maguire, 2006:18). Despite the fact that developers were following the safety code and implementing preventative measures, accidents still occurred; this led to the realisation that the

processes for safety were insufficient. In fact, the ‘prescriptive’ approach enforced a ‘check-list mentality’. Specifically, developers argued safety based solely on the requirements set by regulations; often omitting contextual information related to their system. Further, the incrementing complexity of systems was affecting the safety code, which in turn increased proportionally in size. In (Rushby, 2015:15), the author reviews industrial cases where the code could reach thousands of pages. Due to the size of the document and time constraints during development, it is plausible to assume that certain parts could have been omitted or misinterpreted by the operators.

These concerns and the aforementioned accidents forced regulatory authorities to re-evaluate earlier approaches and eventually adopt a goal-based strategy to address safety. The foundation of a goal-based assurance strategy was firstly introduced in a UK government report, known as the ‘Robens Report’ (Robens et al., 1972). This new approach requires proper justification of safety; as a result, the responsibility of assessment and assurance activities shifts back to the developers. The process, among other tasks, requires the careful documentation of all the actions taken towards safety and is referred to as **safety case**. The approach was soon adopted by various industries, including the nuclear, aerospace and automotive.

Initially, the notion of safety cases was unclear and the characteristics that define a ‘good’ safety case were difficult to determine. Since then, through substantial research and industry feedback, the actions towards the production of the safety case have become more methodical and the structure of the document has improved.

### 2.9.2 Definition of Safety Cases

Based on the literature, a safety case should build confidence about system safety. It is often described either as a set of documents or an argument of safety and its supporting material. This thesis adopts the definition issued in (Kelly, 1998:22). **“A safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context.”**

Following this definition, two keywords require further explanation:

1. **context** – There is no system that can be safe in any context. In this scope, any environmental conditions, human factor, system states or interactions with other systems are considered to be context. Therefore, developers should define the context under which their system is argued to be safe.
2. **acceptably** – To pursue a completely safe system is practically impossible. Thus, safety cases are meant to communicate that systems are adequately safe. Typically, there is a consensus between the governments, regulatory bodies and developers with regards to risk tolerance levels and what is considered acceptable. Then, it is up to the manufacturers to decide the trade-offs between costs and safety levels (assuming they prefer a safer product than what the regulations suggest).

There are two reasons why this thesis adopts the said definition. First, the proposed method supports the construction of safety arguments with cross-sector application and a domain-neutral definition, such as this, is fitting. Secondly, since the proposed method focuses on the synthesis of the argument (i.e. a basic feature of safety cases) and its supporting evidence, it is in line with the above definition.

Safety cases have been adopted by different safety standards, such as the UK Defence Software Systems DS 00-55 (MoD, 1996a) and the civil aircraft ARP4754-A (SAE, 2010). Further, safety cases are also required by regulations such as Railways (Safety Case) Regulations 2000 and Offshore Installations Regulations 1992 in the rail transport and petrochemical industries, respectively. The definition of safety cases differs slightly from standard to standard. For example, one definition taken from the UK Ministry of Defence Ship Safety Management Handbook JSP 430 (MoD, 1996b) is presented below.

“A safety case is a comprehensive and structured set of safety documentation which is aimed to ensure that the safety of a specific vessel or equipment can be demonstrated by reference to safety arrangements and organisation, safety analyses, compliance with the standards and best practice,

acceptance tests, audits, inspections, feedback and provision made for safe use including emergency arrangements”.

The most important aspect of this definition is the explicit reference to safety analyses and acceptance tests. This highlights the fact that any action or information needs to be justified and supported by evidence. It also indicates that a safety case is not just an argument of safety but encompasses a set of activities.

### 2.9.3 Structure of Safety Cases

The actual contents of a safety case might differ across industries. In (Kelly, 2004) the most common actions required for a safety case are explained and summarised below:

- **Scope**, where the system context, limitations and lifetime are defined.
- **System Description**, where a sufficient system description and its version are established. This is important for the familiarization of the user with the given system and its interfaces for communication with other systems (i.e. integrated systems).
- **System Hazards**, where the system-level hazards (for high-level functions) are identified and documented.
- **Safety Requirements**, which are usually set either by the customer, safety standards or defined as mitigation means after hazard analysis.
- **Risk Assessment**, which estimates and assesses the remaining risks after reduction measures.
- **Risk Reduction Measures & Safety Analysis**, where the risk reduction measures, applied for the associated hazards, and the supporting evidence, produced from safety analysis, are evaluated for sufficiency against the requirements.
- **Safety Management System**, during which the project’s safety plan is summarised. The safety plan includes any actions related, and not restricted, to the design process and identification of standards and requirements.

- **Development Process Justification**, during which justifications about safety processes, design methodologies or control procedures, employed at each level of development, are captured in safety-process arguments. These typically complement the product-based argument and together bolster confidence towards the system in question.
- **Conclusions**, where all the findings about system safety in a stated context are presented in a convincing way within the safety case report.

However, at its core a safety case consists of three fundamental elements: a) requirements, b) arguments and c) evidence. The requirements typically consist of claims or objectives related to the system in question. The evidence is the product of safety analyses and assessment activities and is intended as a means necessary to support the corresponding claims. Last but not least, the argument is the intermediate element that communicates the relationship between the requirements and the evidence. Of particular interest are the evidence and argument elements. One is never sufficient without the other. For example, a well-structured argument is not convincing without the appropriate evidence to back it up, whereas results from analyses can be hard to explain and rarely is their connection to the objectives intuitive enough.

#### 2.9.4 Benefits and Limitations

Despite the adoption of safety cases in various industries, there are regulatory bodies that do not necessitate a safety case report to approve system safety and authorise its operation. However, it can be beneficial for the developers to construct and maintain a safety case. First of all, keeping a documentation that explains the rationale and assumptions behind the decision-making with regards to system safety and development processes allows for clarity within the development teams and builds confidence to the stakeholders. Further, having a shared documentation that requires frequent updates acts as a reminder of the remaining tasks and highlights the issues that need to be addressed. Finally, the nature of safety cases to connect evidence, from safety assessment activities, with objectives and requirements in a clear and comprehensive way allows the extraction of high-level

findings to be communicated to the stakeholders regardless of their background in reliability engineering processes.

Nevertheless, safety cases do not come without faults. Many researchers have been questioning the goal-based approach for safety assurance and expressed their concerns on the matter. In (Maguire, 2006:39), the author emphasises that goal-based approaches require way better understanding and background knowledge of the processes involved and the industry best practices, compared to the prescriptive approach. Hence, proper application of a goal-based strategy demands not only the practitioner but the assessors (regulators) to be competent. In (Leveson, 2011b:3-4), safety assurance that is based on safety cases is criticised for often suffering from confirmation bias. Since safety cases aim to provide confidence about system safety and do not directly prove it, it is possible that developers might unintentionally structure claims or represent the evidence in a way that is biased to support their case. Additionally, it is argued that safety cases should be better referred to as ‘risk cases’ to highlight their role in development and avoid any misconceptions. In the article (Kelly, 2008), the author, despite being a great contributor to the goal-based approach, discusses how common mistakes and misinterpretations can undermine the value and degrade the purpose of safety cases. Most notably, the article explains that some safety standards erroneously define safety cases as sets of safety documentation; whereas in practice, “a safety case is an argument of safety and its supporting material that is presented and communicated through a structured set of safety documentation”. Following the line of thought in the article, the key principles to gain value from safety cases are: a) to thoroughly construct an argument that accurately depicts all appropriate activities regarding system development, b) to maintain the safety case throughout the system lifecycle and c) to evaluate the safety case with focus on clarity, realism and appropriateness of the supporting material of claims and not on the medium or notation related to the presentation.

## 2.10 Safety Argument Notations

Regulations and safety standards advocate that safety arguments should be presented in a clear and comprehensive way, but many of them do not require a specific medium for presentation or structure.



Initially, safety cases were introduced as a set of documents and the arguments were traditionally communicated in text format. The main issue with that approach was the large volume of information and the heavy use of cross-referencing that rendered safety arguments hard to follow. In addition, with plain text, structure and accuracy with regards to the description of the processes involved was subject to the author's writing ability. These challenges prompted concerns within the reliability community and many suggestions were provided to improve the presentation means of safety cases. The following list contains a summary of these suggestions:

- **Tabular structures**, where the related text of the safety case is organised in a set of core elements and placed in tables that are categorised into claims, arguments and evidence. However, this approach failed to appropriately illustrate the thought-process of assurance (relationships of higher-level claims with intermediate and lower-level claims/evidence). In addition, there were no reference sources regarding the contents of each element (Kelly, 1998:49-50).
- **Claim structures**, which encapsulate the safety cases within tree graphs. In this concept, the nodes represent claims which are interconnected with logical gates and form a hierarchical claims structure (MoD, 1996a). The problem with this approach was the lack of reasoning, evidence and other details.
- **Bayesian Belief Networks (BBNs)**, where a safety case is represented in the form of a directed graph. Elements related to safety cases constitute the nodes of the network. Each node represents a variable and encapsulates probabilistic data which can be used to quantify the probability of the parent-claims. However, not all variables are measurable in safety assurance (e.g. developer experience or guideline understanding). In addition, the lack of entities necessary to properly represent argument logic and assumptions make BBN not an ideal candidate for safety assurance. However, some research projects have used BBNs for safety assessment; for instance, for the analysis of safety measures to prevent time-dependent accidents in (Mancuso et. al., 2019), safety assessment of remotely piloted aircraft in (Washington et al., 2019) and risk evaluation purposes in (Marsh, 1999).

### 2.10.1 Goal Structuring Notation

The Goal Structuring Notation (GSN) is a graphical notation that provides the means necessary for the representation of safety arguments. GSN was developed at the University of York in the early 90's as part of the ASAM-II project and its continuous development is undertaken by the Assurance Case Working Group (ACWG, 2018). The work of Stephen Toulmin in (Toulmin et al., 1984; Toulmin, 2003) had a major impact in the conception and early development stages of GSN. Specifically, Toulmin set the fundamental entities of reasoning, including the claims/assertions as well as any considerations related to the appropriate supporting elements. GSN's aim was to bestow a systematic way for structuring and representing safety and assurance arguments.

GSN features a set of basic elements: a) **goals** that represent objectives, b) **strategies** that communicate the argument logic and c) **solutions** that constitute evidence fragments. Goals can be viewed as requirements, objectives or other properties the system is argued to fulfil. The solutions are the elements responsible for supporting goals (usually low-level goals) and typically consist of references to the results of safety assessment methods. The strategies describe the rationale that links goals with subgoals or solutions elements. These interconnected elements form a tree graph. This structure typically mimics the system's hierarchy, in the sense that it starts with a high-level goal (usually high-level system elements), which is then decomposed into multiple lower-level goals (down to component elements). In other words, the combination of subgoals and strategies act as inferential steps within the argument structure that describe the basis in evidence and reasoning that led to the satisfaction of the top-level goal. Fig. 2.21 shows all the basic argument entities provided in the GSN, adapted from (Wei et al., 2019:5). Any goal or strategy elements that are not completely developed can be assigned with a special symbol (i.e. a rhombus shape) underneath their corresponding node.

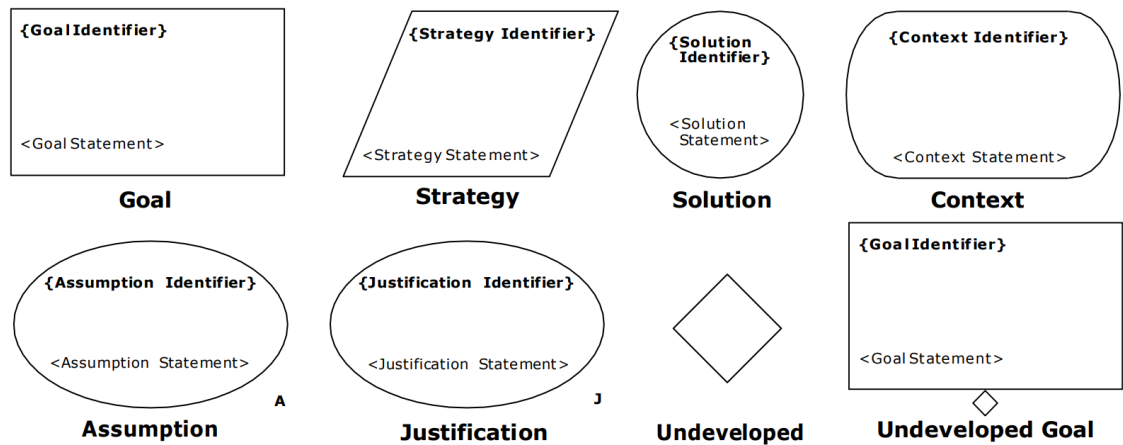


Figure 2.21: Basic GSN Elements (Wei et al., 2019:5)

Apart from the basic elements, GSN also provides a set of supplementary entities which allow the developers to communicate concerns, assumptions and contextual information in the argument. These are summarised below:

- **Justifications**, that explain the decision-making behind actions (e.g. why FTA instead of MA).
- **Assumptions**, which present any hypotheses or logic that support a claim or strategy (e.g. if a claim argues elimination of all hazards; an assumption could specify that all credible hazards have indeed been identified. The reader would immediately understand that the claim refers to that set of hazards).
- **Context**, which provides useful information related to the element it is linked to (e.g. a claim that argues ‘component is safe’ is not always valid, even when the evidence supports it, because it is important to take into account the circumstances under which the assessment evaluated that the system is safe (operational phase, etc.).

Fig. 2.22 shows how a simple example of an argument structure is presented in GSN. The argument structure presented below is a reproduction of the structure found in (Joba, 2015).

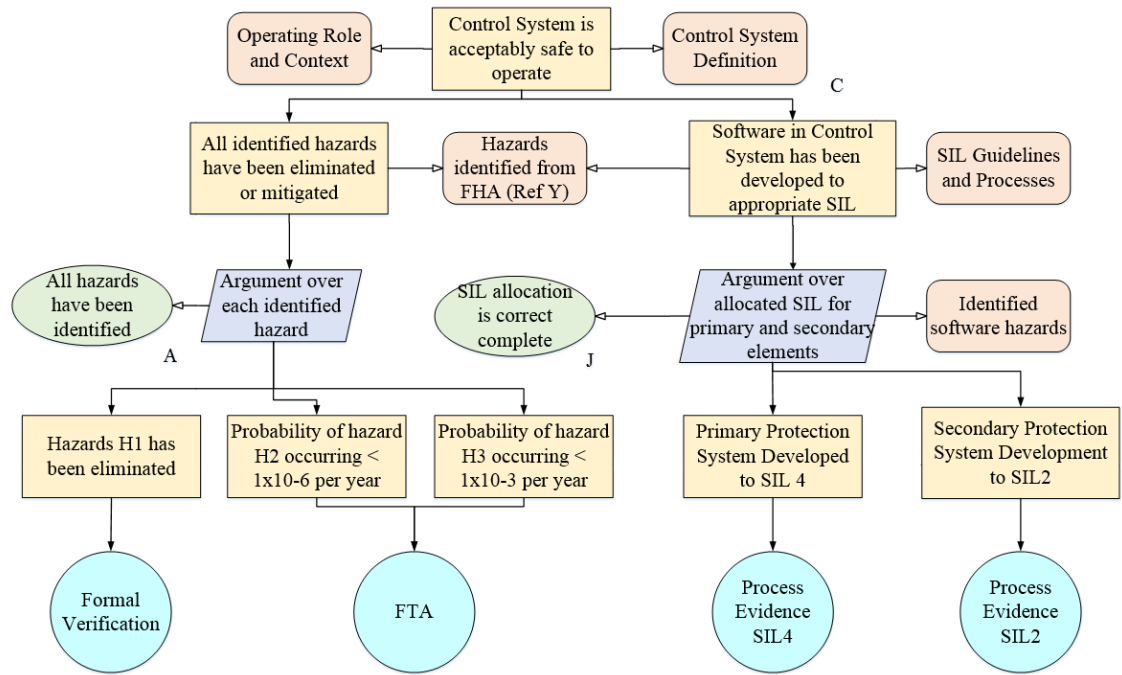


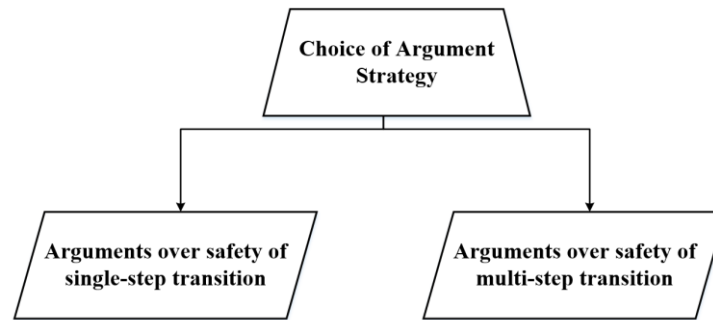
Figure 2.22: Example of an argument in GSN (Joba, 2015)

Even though there are no standard rules, the structure above uses some colours to help separate the different GSN elements and make them more clear for the reader. Goals are represented with a yellow colour whereas strategies use a blue colour and evidence artefacts use a cyan. Context and justification elements are highlighted with orange and green hues, respectively. In this example, the top-level goal argues that the ‘control system’ is acceptably safe to operate and the rest of the structure (subgoals, strategies, assumptions and context elements) communicates how this is achieved. Two subgoals directly support the top-level goal, one by claiming that all system hazards are identified and mitigated, whereas the other argues that software is developed and analysed appropriately based on the assigned safety requirements (SILs). The strategy elements communicate how these are further supported by lower-level subgoals. The low-level subgoals (on the left) claim that each of the identified hazards has been either eliminated or mitigated. Evidence elements provide the corresponding results from analyses (in this case FTA and formal methods). The goals on the right side, claim that each of the software elements have been developed based on the allocated SIL. Evidence elements provide results from appropriate activities that directly support these claims. Justification and assumption elements, although they are complementary in the structure, establish a

degree of confidence before a strategy further develops. For example, the justification in the middle of Fig. 2.22 strongly suggests that the SIL allocation process was performed correctly. Finally, contextual elements (orange-coloured shapes) represent useful information about the system and the SIL allocation process.

The GSN has many benefits for the presentation of safety arguments and has been widely used. First, the solution graphs provide an explicit representation of evidence elements (references to analyses results). Secondly, the justification and assumption elements increase the confidence on the safety argument by communicating the rationale behind certain tasks and decisions. Finally, the use of directed relationships makes the logical flow easy to follow compared to text-based approaches.

Initially, GSN was criticised in (Kelly, 1998:56-57) for drawbacks. The concerns were raised due to lack of proper documentation. In fact, there was no guidance to aid the engineers with the construction of arguments under the GSN. Furthermore, the semantics were not mature at the time and certain elements were unclear. These issues were resolved with the standardisation of GSN in 2011 and over the years, it has been extended with supplementary features that support both the management and maintenance of safety arguments. This has been achieved with the inclusion of modular extensions, structural and element abstractions, and the addition of new symbols. The latter are referred to as ‘non-modular extensions’ and expand the GSN’s ability to describe new concepts. For example, with the choice-of-strategy elements, developers can indicate that there might be multiple strategies available for the construction of the argument structure from that point on. Although it appears to fall under the category of ‘undeveloped’ (diamond-shaped symbol), it is more useful to signify an undeveloped strategy and simultaneously present the available choices (i.e. argument strategies). Naturally, this concept is usable only during the early versions of the argument; meaning that by the time the final argument is complete, all the choice-of-strategy elements and undesirable choices must be removed. Fig. 2.23 shows a simple example found in (ACWG, 2018:77).



*Figure 2.23: Strategy choice example*

In this example, the argument structure stops at the point where the strategy for the project transitioning from one operation to another has not been decided. The choice-of-strategy is represented with the trapezoid shape, which is connected with two strategy-elements that encapsulate the different strategies available at this time in development. Strategies can be added or removed based on the needs; however, once the decision is made, the rest of strategy-elements and the choice-of-strategy symbols are discarded.

### 2.10.2 Claims Arguments Evidence Notation

The Claims Arguments Evidence (CAE) is yet another notation for the structuring and presentation of safety arguments (CAE Framework, 2020). It was developed in the late 90's by Adelard, an independent specialist consultancy firm in the UK (Bishop & Bloomfield, 1998). CAE features a set of entities, comparable to those of GSN, that are responsible for representing basic argument elements:

- **Claims** that represent the objectives
- **Arguments** that communicate the rationale
- **Evidence**, which are references to supporting information

In terms of structuring, the CAE notation uses arrows of an opposite to the GSN direction. Further, it utilises colours to emphasize the different elements on the graph. Fig. 2.24 presents a simple structure in CAE.

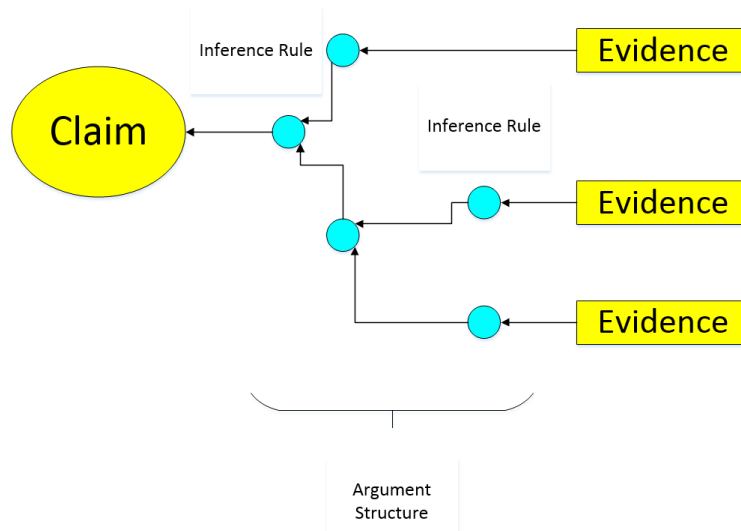


Figure 2.24: A simple structure in CAE notation

Similarly to GSN, CAE is heavily influenced by Toulmin’s work. In fact, it follows the same philosophy that reasoning is impossible to be communicated with just flow charts and logical networks. In CAE, ‘assumption’ and ‘context’ elements are kept as rich and verbose as possible.

CAE and GSN share common features and serve a similar purpose; thus, it is up to the developer’s preference and expertise to determine which notation should be used. There are many modelling tools that support either GSN or CAE and others that support both. For instance, the OMG standard has developed the Structured Assurance Case Metamodel (SACM) that supports both notations and transforms a model from GSN to CAE and vice versa. This is useful for transforming a safety argument and its entities between the two notations and use the most appropriate based on the needs. On that note, there are no real disadvantages as to which notation is used. The notations merely provide the means to represent arguments graphically, but it is up to the safety case engineer to identify how the argument should be structured, what the claims argue and how these are supported.

## 2.11 Methods for Managing Safety Cases

The method proposed in this thesis utilises a simplified version of GSN for the representation of arguments, mainly for demonstrating purposes (i.e. CAE could be an equal candidate). In addition, GSN has been extended with a set of management features, described in (ACWG, 2018), that facilitate

the argument construction and visualisation. The following sections discuss in more detail these concepts, previously reviewed in the introduction (Chapter 1), regarding the management of safety cases.

#### 2.11.1 Argument Patterns

The concept of software patterns has emerged to simplify the resolution of re-occurring problems by establishing a reusable strategy towards their solution. Patterns are frequently used in software design to abstract the details of a particular problem and define a ‘blueprint’ that can be shared and used under the appropriate circumstances. The successful application of patterns, especially after the publication of (Gamma et al., 1994), has inspired researchers to extend the concept and adapt it for the safety engineering domain. Specifically, in (Kelly, 1998) the notion of safety case patterns is introduced with the purpose to capture repeated structures of successful arguments and use them in safety cases. These patterns typically emerge and inherit traits from successful argument structures (i.e. correct, comprehensive and convincing) and are designed by experts in the industry of interest. The pattern is often accompanied by documentation that contains information regarding the intent of the pattern, advice on when to use and supplementary details. The latter usually include the behaviour of the generation process of the argument structure and potential alternatives to the chosen pattern. Generally, patterns that serve a similar purpose tend to be grouped into categories and often distributed within the domain as documentation. The first iteration of such document, referred to as the ‘catalogue’ for safety case patterns, was introduced in (Kelly, 1998:332) and was later extended in (Hawkins et al., 2011). Fig. 2.25 shows an example of a GSN pattern, adapted from (Hawkins & Kelly, 2013: 12).





changes the output based on a set of input variables. The engineer chooses the appropriate pattern and instantiates it either manually or, given the appropriate algorithms in place, automatically.

The GSN standard supports pattern specifications with two types of abstraction: a) structural and b) element abstractions. Structural abstraction is mainly achieved through the multiplicity and optionality extensions. The multiplicity allows the generalization of n-ary relationships between GSN elements whereas the optionality enables capturing alternate or optional relationships between elements (e.g. 1-of-n relationship). With the element abstraction, GSN allows one entity from an argument structure (instantiated) to be abstracted from any particular details. Fig. 2.26 shows one generic example, adapted from (Kelly, 1998:170; ACWG, 2018:26), for the optionality extension and one for the entity abstraction with the un-instantiated symbol ( $\Delta$ ).

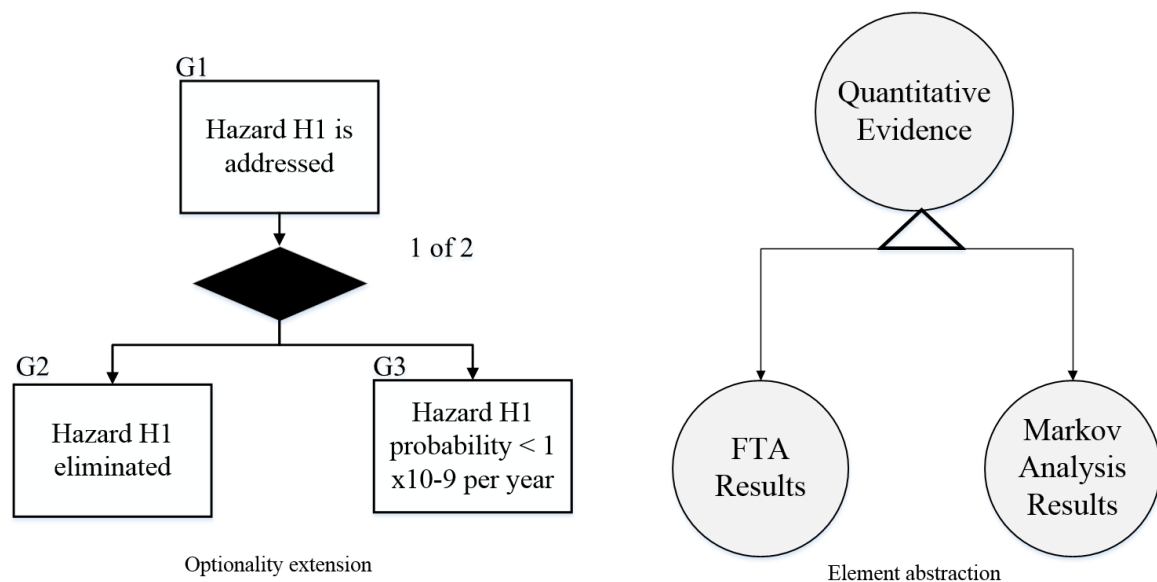


Figure 2.26: Example diagrams for Element abstraction and Optionality extensions (Kelly, 1998:170; ACWG, 2018:26)

On the left, the diagram indicates that the goal G1 can be supported by either subgoal G2 or G3. On the right, a solution element, that would potentially support a goal claiming a failure rate, can be generalised to represent evidence from quantitative analysis instead of stating the exact analysis method, such as FTA or MA.

### 2.11.2 Modularisation of Safety Cases

The GSN standard has also been extended with modularisation capabilities. This allows the practitioners to organise a large or complex safety case into individual, yet interrelated, argument modules. This can be useful for dividing the initial argument into multiple argument subgraphs that focus only on specific facets. For example, an argument claiming aircraft safety can be divided into multiple modules that claim safety for all the top-level aircraft functions. To achieve this, the GSN has been extended with new elements, such as the ‘module reference’ and the ‘away elements’. Fig. 2.27 depicts an away goal on the left and a module element on the right, reproduced from (ACWG, 2018:29).

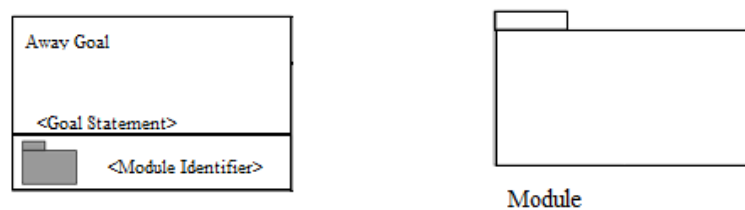
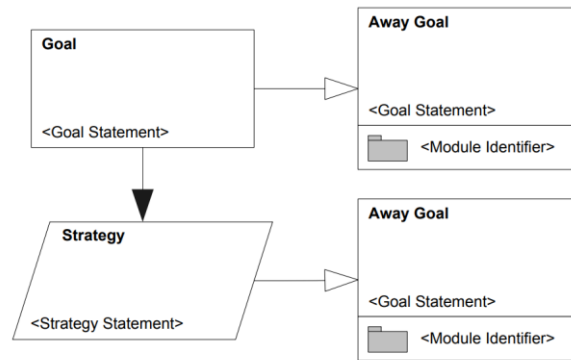


Figure 2.27: The Away Goal and the Module reference elements (ACWG, 2018:29)

A module reference element essentially contains a reference to an argument module which captures an argument subgraph. An example of an away element is the away goal, which can be a reference to a goal captured in another module. It basically mirrors the referenced goal and can be used as a subgoal that supports the argument within its local module. With this extension, a safety case can be organised into multiple modules, which allows a high-level view and makes the overall argument structure easier to comprehend. Additionally, the segmentation of the argument structure underpins the concurrent development of multiple sub-arguments that can be valuable when many analysts collaborate on the construction of a safety case. The GSN standard provides a few rules to make the use of these new elements clear and help maintain their intent. For instance, away elements exist only to repeat their base counterparts and therefore cannot be decomposed into lower-level elements. If development requires further support for any of the away elements, the referenced elements (to which the away element refers) should be decomposed and supported by core GSN elements instead

(ACWG, 2018:32). They can also be used to provide contextual details to goals, strategies or solutions from one module to another (i.e. remote modules - unconnected to the current overall argument). That is especially useful when a claim or a strategy requires more detailed justification than the one provided with the core justification elements. Fig. 2.28 illustrates how the latter is achieved.



*Figure 2.28 Replacement of justification by away goals*

## 2.12 Safety Integrity Levels

The International Electrotechnical Commission (IEC) set two task groups in 1985 to develop a generic standard for programmable electronic systems. One of the groups focused on a holistic systems-based approach, whereas the other one dealt mostly with safety-related software. Later, these two groups collaborated with the purpose to create an international standard, today known as the IEC 61508. This standard is about the “functional safety of electrical, electronic and programmable electronic safety-related systems” and has been used both as a standalone standard or as a basis for producing new standards for various industries (e.g. railway or nuclear industry) (IEC, 1998). One of the key features of the IEC 61508 is the utilization of **safety integrity levels (SILs)**. SILs are used for specification of the appropriate level of safety integrity for safety functions, that are to be implemented by electrical, electronic and programmable electronic safety-related systems. Such systems are considered capable of functionality that can be carried out with the required safety integrity. The latter measures the performance of a safety function, or more specifically the likelihood of a safety function being achieved when requested. A safety function is a function of a machine, whose failure could result in an immediate increase in risk (and potentially lead to an accident). The standard provides a technical

framework that offers a systematic way to achieve the required safety integrity, also known as ‘Overall Safety Lifecycle’. Fig. 2.29 is a reproduction from (Bell, 2014:10) that depicts this lifecycle and its relevant phases.

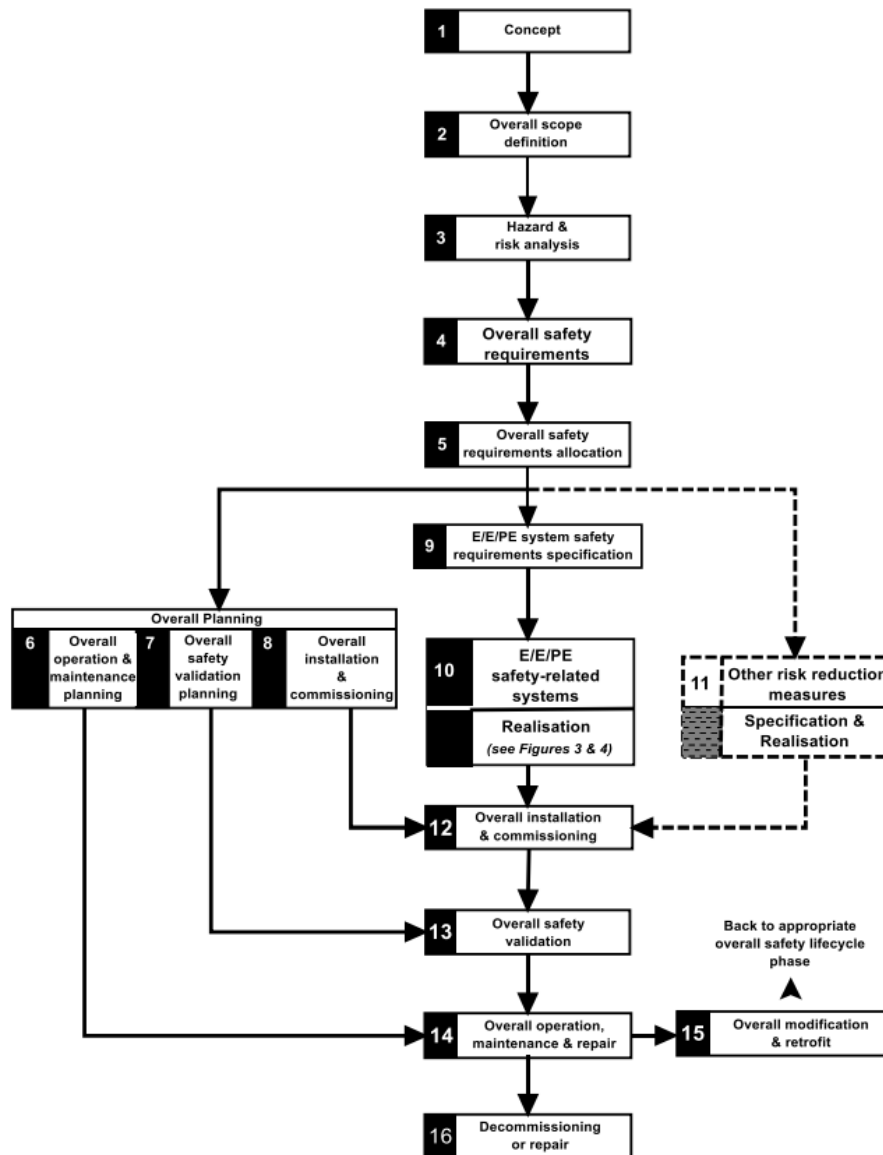


Figure 2.29: Safety lifecycle in IEC 61508 (Bell, 2014:10)

In general, SILs are used to describe the level of rigour required when performing the respective assessment activities on system entities (e.g. functions or subsystems). In IEC 61508, SILs are divided into four different levels, where SIL one is the lowest and SIL four is the highest level of safety

integrity. The standard advocates that these safety requirements should be decomposed and allocated from system level to subsystems, and finally down to items (with no further dependencies) for either software or hardware architectures (Azevedo et al., 2013).

Various standards have adopted the concept of SILs with minor changes. The Automotive Safety Integrity Levels (ASILs) in ISO 26262 and the Development Assurance Levels (DALs) in ARP4754-A are two known examples, which have an integral role in the development of systems compliant with the regulations in the automotive and aerospace industries, respectively. Further investigation in the next chapter presents more details on how these domains utilise the concept of SILs for safety assessment. This will allow to understand not only how the standards use SILs but also how the proposed method employs SILs to generate arguments of safety. Finally, through the broad use of SILs in the safety critical sector it becomes clearer why the method can be applicable in various domains.

## 2.13 Modelling in Safety-Critical Systems

This section briefly presents related work on modelling languages and metamodels related to systems engineering applications. This investigation showcases how earlier research approached this topic and helped in the development and gradual improvement of the proposed metamodel in terms of structuring and organising models (e.g. element relationships) and evidence information (e.g. analysis results produced X by operator Y at time Z during development). Any instance of the word model might refer to either a model or a metamodel, which in this context is considered as a set of concepts and the relationships between them, independently of the graphical or textual languages used to represent them.

### 2.13.1 SACM Metamodel

Structured Assurance Case Metamodel (SACM) is an Object Management Group (OMG) metamodel designed to represent structured arguments and more specifically assurance cases. In the specification (OMG, 2018), an assurance case is described as a set of claims, arguments and evidence developed

to support a high-level claim that a system or service satisfies a specific set of requirements. In its general form, an assurance case is a document that enables information sharing between system stakeholders, such as developers and regulators. The information communicated can be system knowledge, operational details and claims related to system safety or security, which must be presented in a clear and comprehensive way. There are no specific restrictions as to what information should or should not be included, but at the very least most assurance cases should contain the following:

- The scope of the system
- The operational context
- The particular claims for the system phase
- The safety and/or security arguments and their supporting evidence

Note that the backbone of any assurance case is the cogency of the argument, which helps build confidence that a system meets the appropriate requirements, and the appropriateness of the supporting evidence. The SACM was originally created in 2012 from the combination of two metamodels, the Structured Assurance Evidence Metamodel (SAEM) and the Argumentation Metamodel (ARM). Ever since, the SACM has received changes to further integrate its dual-metamodel characteristics, to support the GSN elements and extensions, to include and support patterns and templates, as well as to simplify its metamodel. The SACM consists of five components as shown in Fig. 2.30 adapted from (Wei et al., 2019:7).

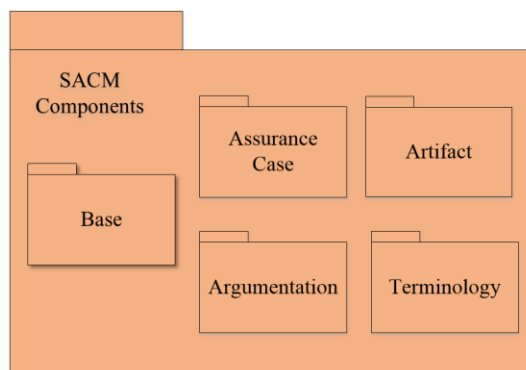


Figure 2.30: The SACM's basic components (Wei et al., 2019:7)

These components include technical descriptions for elements, such as specialised strings and expressions. Certain elements can be linked together to represent an assurance case. All components, except for the base, contain a specific package element, which serves as a high-level container for its component's elements. Package elements may individually contain different metamodel elements, other package elements or references to package elements. A brief description of the components found in SACM is given below:

- The **Base Component**, which encapsulates all the basic entities included in SACM, such as strings enhanced with multi-language capabilities. It also provides a description of the SACMElement, which contains all the basic features found in other SACM elements, such as the global identity (+gid) and the abstraction Boolean notifier (+isAbstract).
- The **AssuranceCase Component** includes elements related to an assurance case as a concept. The most notable element is the AssuranceCasePackage, which is the highest-level exchangeable element in the SACM, and it may contain argumentation, terminology or artifact package elements. Due to its recursive nature, it is capable of containing other lower-level AssuranceCasePackages and therefore can be viewed as an assurance case module (similar to the GSN modules, for creating subgraphs of assurance cases). Users are typically expected to use this container when exchanging content.
- The **Artifact Component**, which describes all the elements responsible for capturing evidence artefacts from internal or external models. The ArtifactPackage elements are composites that contain other sub-elements, such as resources, techniques and participants.
- The **Terminology Component** principally features the TerminologyPackage that encapsulates the elements necessary for creating user-controlled expressions, vocabularies and terms.
- The **Argumentation Component**, which provides all the descriptions and entities for representing structured arguments. The ArgumentationPackage acts as the container of these entities and can also facilitate the exchange of assurance arguments from external models.



Overall, in SACM an assurance case (represented with one or more AssuranceCasePackages) consists of arguments (contained in ArgumentPackages) that are supported by evidence (contained in ArtifactPackages), where all definitions or system expressions are contained in TerminologyPackages (OMG, 2019:29). Since the AssuranceCasePackage is the key component that showcases how an assurance case is organised under the SACM, Fig. 2.31 depicts a reproduction of its class diagram.

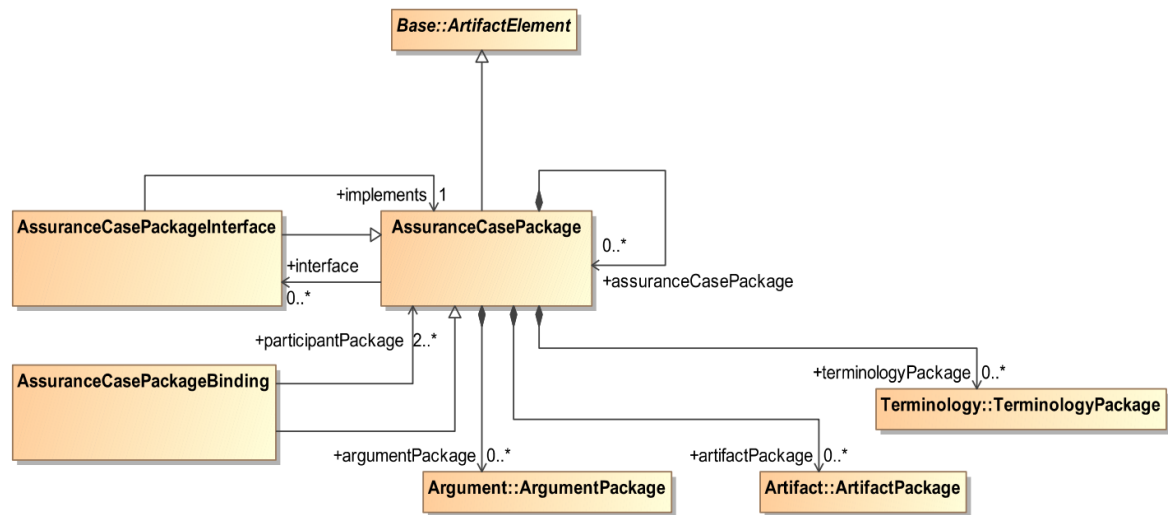


Figure 2.31: The SACM's AssuranceCasePackage Class Diagram (OMG, 2019:29)

The AssuranceCasePackages can be used recursively to build more complex structures (i.e. modularised assurance cases). In addition, when two systems are to be integrated and the developers needs to create an assurance case from the combined assurance cases of these two systems, they may use the AssuranceCasePackageBinding and AssuranceCasePackageInterface elements. As such, the SACM also allows one argument structure to be built from multiple argument structures provided from different vendors. The AssuranceCasePackageBinding elements link two or more AssuranceCasePackages via the use of other binding packages, such as the TerminologyPackageBindings and ArgumentPackageBindings. This means that systems integrate at the level of argumentation via the appropriate underlying logic contained within the ArgumentPackageBinding.

Generally, the SACM has several merits; it is standardised, relatively mature and offers the appropriate infrastructure and level of abstraction to support openness when referencing external information (other models and/or documents). Given an appropriate implementation, SACM is an excellent choice for open adaptive systems (e.g. Cyber Physical systems), which are open for dynamic integration and adaptiveness in real-time situations. Despite its relatively recent introduction, the SACM has been employed in various research projects, such as in (de la Vara et al., 2016) and others discussed in Chapter 4. However, the SACM is not necessarily suitable for every application; thus, further investigation to identify whether or not it is fit for the proposed method is presented in Section 4.6.1.

### 2.13.2 SysML

The Systems Modelling Language (SysML) is a specification language standardised by the OMG in 2005. The purpose of SysML is to be used by engineers as a common modelling language across a variety of domains. It uses a subset of UML 2.5 and further extends it to support the specification, analysis and design of systems in engineering applications. With its direct connection to the UML and the changes to the architecture over the years, SysML allows both systems, and software, engineers to collaborate more effectively on models of software-intensive systems. Given substantial customisation, SysML can be utilised in various industries, such as the aerospace or the automotive; however, it focuses more on project-specific and architectural aspects rather than the details of assessment and other processes described in safety standards. For instance, there is no infrastructure for modelling the requirements defined by safety standards nor the underlying actions that provide the supporting evidence.

### 2.13.3 AADL

Architecture Analysis & Design Language (AADL) is a metamodel developed and standardised by SAE that supports system architecture modelling (SAE, 2017). The AADL focuses on the aerospace industry and provides both a textual and a graphical notation to describe the architecture and any functional interfaces of either a hardware or a software system.

AADL follows a component-based paradigm and views systems as a set of components with interactions flows and further subcomponents. Different configurations and topologies are typically represented with different modes (operational states), flows across a sequence of subcomponents and properties. A summary of the basic AADL elements, as found in (Feiler et al., 2006:253), is shown in Fig. 2.32 below. A key feature of AADL is the concept of the Error Model Annex (AADL EA), which describes a state transition mechanism that represents the transition between normal and failed states. The properties defined in the error model may include fault propagation behaviour, fault assumptions or fault tolerance policies. The propagation is specified with rules set explicitly by the engineer.

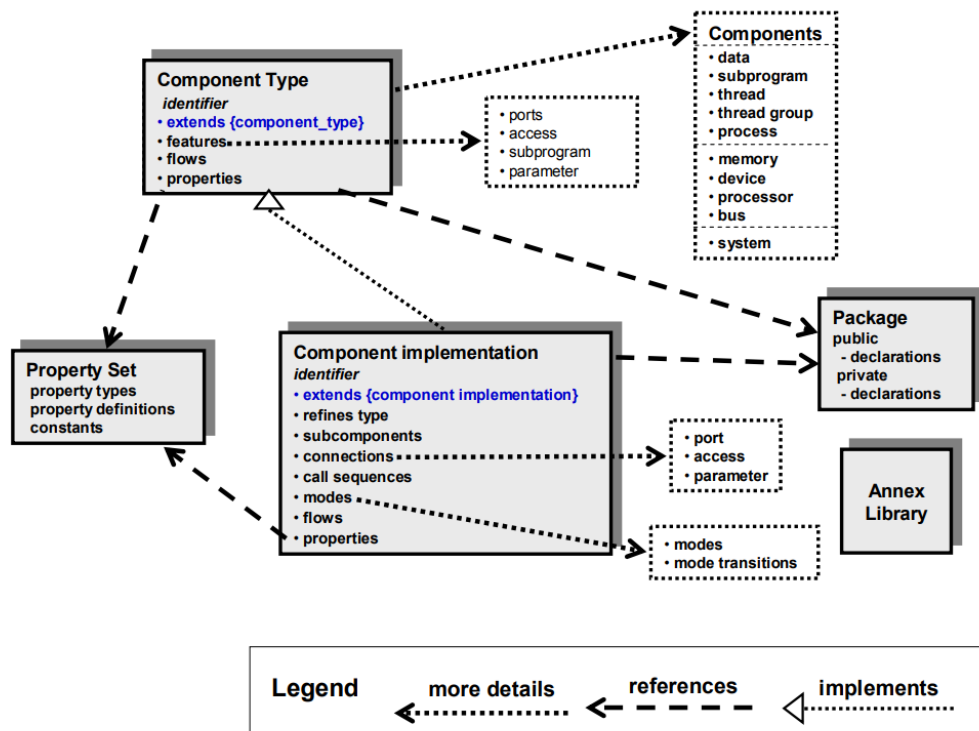


Figure 2.32: A Summary of the AADL Elements (Feiler et al., 2006:253)

AADL has been the base in architectural modelling for a number of projects such as in (Feiler & Rugina, 2007) and (Mian et al., 2012). Finally, its ability to support state machine definition is considered highly valuable because it allows the incorporation of temporal features into the design and analysis. State machines enable analysis of system failures that depend upon a particular sequence

of two or more failure events. Despite the discussed benefits, AADL's primary focus on the aerospace industry makes it less desirable for use in other domains; hence, it could be restrictive for the method of this thesis.

#### 2.13.4 EAST-ADL

EAST Architecture Description Language (EAST-ADL) was defined during the ITEA project in the early 2000s, and since then, it has been refined through several international funded projects. EAST-ADL and its most recent version EAST-ADL2 both provide an approach for describing electronic systems for the automotive industry. The system is described via an information model which consists of 4 abstraction layers, each one with a distinct role.

1. The **vehicle layer** that describes vehicle level functionality.
2. The **analysis layer** that describes any input, output or control elements.
3. The **design level**, which provides the functional architecture allocated to a hardware architecture.
4. The **implementation layer** that links any embedded systems with the corresponding AUTOSAR elements.

The AUTOSAR is a collection of automotive systems specification standards that focus on low-level architecture. The latter consists of different elements, such as the basic software, runtime environment and application layer. The communication between low-level elements is achieved via the use of the SOME/IP protocol, under which control units can share data and request operations independently through the vehicle's network. The dependability package featured in EAST-ADL is designed to support the development guidelines described in the ISO 26262 safety standard. EAST-ADL describes an error model that is used to capture potential errors and their propagation. Moreover, the utilisation of state machines, similarly with the AADL, provides the means for extensive behavioural analysis of a system. Fig. 2.33 features an overview of the EAST-ADL abstraction levels as found in (EAST-ADL, 2017).

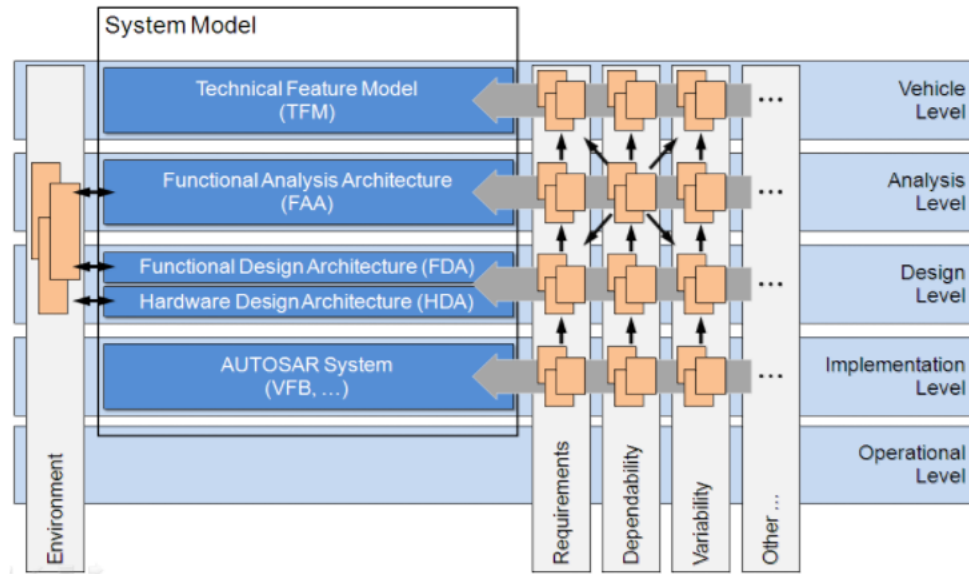


Figure 2.33: The basic EAST-ADL model (EAST-ADL,2017)

Despite its usefulness in the automotive sector, EAST-ADL metamodel is not the best candidate as a metamodel for the proposed method. It would require significant effort in order to be able to utilise the produced models and their failure information with any of the investigated safety analysis tools. Also, the proposed method has a wider scope; it includes industries of the safety critical sector that use the concept of SILs for the development and safety assurance. Not all industries suggest a level of complexity similar to the one found in the automotive. For example, next chapter showcases some of the differences, related to system modelling and development, between the aerospace and automotive industries.

## 2.14 Summary

This chapter provided the fundamental terminology in safety assurance, as well as background information on safety assessment activities, safety analysis methods and tools. This allowed for an investigation of suitable techniques and frameworks that could prove useful in the materialisation of the proposed method. Notably, HiP-HOPS was found to be a good candidate for hypothesis testing, as it automatically analyses system models and generates valuable information artefacts. The chapter also discussed the concept and history of safety cases and established current issues related to their

production and maintenance. Further, it presented the GSN and CAE graphical notations, which support the structure of modern safety arguments, along with their extensions. This helped in identifying the means for presenting arguments and related issues. Moreover, the exploration of argument patterns signified the potential for the automatic argument generation. The chapter concluded with an investigation on notable metamodels for assurance and safety cases with the purpose of understanding core engineering concepts on connecting regulatory and analysis artefacts with system models. Some of the key points of the chapter are:

- a) Safety cases are widely used for certifying safety-critical systems (e.g. aviation industry) and domain standards require them to be maintained throughout the system's lifecycle.
- b) Whether document-based or graph-based (via the argument notations) and despite the advancements and automation in safety analysis tools, they are still constructed manually.
- c) Manual approaches typically introduce errors and can be costly in time and effort especially in the evolutionary development of safety-critical systems. This drives the developers to omit proper maintenance of safety cases during, at least, the early stages of development.

The information presented in this chapter helped to understand not only the current state of the research area but also highlighted that automation, both for the assessment and assurance tasks, could be a major factor for mitigating the issues, related to the safety cases, that are found in the literature. Finally, the brief review on the safety-critical sector history and previous accidents signified the importance for introducing safety-related activities from the early development stages. This realisation clarifies the need for improvement in our practices and this research work provides a possible solution.

## Chapter 3 Safety Assessment in Automotive and Aviation

---

The previous chapter was centred mostly around the safety assessment, analysis techniques and tools as well as common practices in assurance. Instead, this chapter focuses more on the certification process mandated by regulatory bodies and the guidelines advocated by modern safety standards for the development and safety assessment of safety-critical systems. Specifically, it provides an overview of the safety lifecycle in the civil aircraft and automotive industries under the guidelines of ARP4754-A and ISO26262, respectively. This is important for the identification of common elements and potential differences between the standards that are important for implementing the method in a way that is applicable to various domains whilst keeping it in compliance with the guidelines. This chapter also presents the safety integrity requirements (i.e. DALs and ASILs) in more detail and explains what the decomposition rules in the investigated industries are. The chapter continues with a simple example on how this task is handled in the proposed method with one of the extensions of the HiP-HOPS analysis tool. Finally, the chapter concludes with a review on other prevalent optimisation techniques for the optimal allocation of SILs and a showcase of the algorithm underneath the HiP-HOPS's extension.

### 3.1 Functional Safety

Before looking into the details of safety assessment activities for the aforementioned industries, it would be helpful to understand how safety standards view safety. Arguably, safety-critical systems may lead to catastrophes for various reasons, such as component failures, environmental conditions and human illegal activities. However, the safety standards discussed in this section address mostly concerns related to **functional safety**, although there are differences between them on the matter. For example, ISO 26262 does not address hazards, such as fire or radiation, when not originating directly from component malfunctions or system interactivity (ISO, 2011:157). These fall under a different category and are often handled with risk management processes specific to the vehicle type. On the other hand, the ARP4754-A deals with hazards caused from external events (i.e. atmospheric or environmental), excluding attempts of sabotage, with Particular Risk Analysis (PRA) (SAE,

1996:27). Under these standards, functional safety is the part of safety that depends on a system and its elements (E/E/PS systems) operating correctly in regard to the corresponding inputs (IEC, 2020). Note that the framework for safety provided in certain standards (e.g. ISO 26262) is extended to support other types of components, such as hydraulic and mechanical. Most functional safety standards categorise failures, briefly discussed in Sections 1.5 and 2.2.6, into a) random, b) common cause and c) systematic. Of particular interest are the systematic failures and random failures, which increase as the systems become more complex and integrated, especially in mechatronic engineering. Random failures from material degradation can be proactively reduced with use of higher-quality components; however, systematic failures can be hard to eliminate when the causes are unidentifiable. Fortunately, the latter can be addressed primarily with inductive (e.g. FTA or Reliability diagrams) or deductive (FMEA or MA) analysis. Once the causes become known, elimination or reduction of systematic failures is possible either via design alterations or reuse of successful system designs and well-trusted components (ISO, 2011:12).

### 3.2 Safety Assessment in Modern Safety Standards

Following earlier discussion, it is established that to achieve acceptable safety, and specifically functional safety, it is important to alleviate any risks related to hazards caused by system malfunctions. Depending on the standard, the system malfunction may be referring to electronic, electrical and programmable electronics (e.g. for the ISO 26262) or to an aircraft and its embedded systems (for the ARP4754-A). The IEC 61508 was created with the intent to act as a generic safety standard with which industry experts would be able to create sector-specific standards and supporting documents responsible for guiding the safety assessment. For the most part, IEC 61508 was used directly in many domains; however, the need for specific guidance prompted its adaptation and led to the production of specialised standards (IEC, 1998:11; Baufreton et al., 2010:2). The following list includes a few notable industry-specialised standards (either standalone or adaptations from IEC 61508):



- **ED-79A & ARP4754-A:** for the civil aircraft industry, specifically for systems that implement aircraft-level functions (SAE, 2010).
- **ISO 26262:** for the automotive industry, specifically for safety-related systems that include E/E components and are installed in passenger cars (ISO, 2011).
- **EN50128 & EN50129:** for the railway industry, specifically for software and electronic systems for communication, processing and signalling (CENELEC, 2011).
- **IEC 61513:** for the nuclear domain, general guidance for installation and control systems in nuclear power plants (IEC, 2011).

All of the above standards provide a framework to the developers for the identification and mitigation of failure causes; thus, enabling risk-management. For random failures correlated with hardware components, due to physical degradation and wear out, the standards typically define safety requirements in the form of acceptable probability failure rates (maximum) and in some domains they also prescribe the acceptable number of individual faults that lead to such failures. On the other hand, when dealing with systematic failures introduced during the specification, design or implementation, none of these standards suggest a probabilistic assessment (Baufreton et al., 2010:6). For such failures, standards define a different type of requirement, known as SIL, and prescribe a process based on this concept. The process includes several assessment activities that need to be performed during the development and validation of the system. The role of SILs is to control the level of rigour appropriate for each of the assessment activities according to the criticality level of the target failure. Note that each domain has different criteria for assigning criticality levels and therefore many standards have customised the concept of SILs to suit the industry's needs. For instance, in the civil aircraft sector, requirements are known as DALs, whereas in the automotive industry they are referred to as ASILs. To further expand on the discussion in Section 1.6, the notion of SILs is important for managing systematic failures, due to their nature. Random failures can be studied, and to a degree quantitatively predicted, based on data from previous usage and knowledge in material science. However, systematic failures of software, and the logical elements of a hardware architecture, cannot

be determined with statistical analysis. There are methods to address systematic failures; however, rather than mandating specific techniques, standards provide generic requirements (i.e. SILs) to allow developers to indicate the level of rigour of activities used for specific parts of the architecture. As a result, the developers can choose and adjust the assessment methods based on the severity of the risk associated with the failures and provide an appropriate level of confidence.

The investigation of safety standards suggests that there are converging elements in the prescribed process across various industries for the assessment of safety-critical systems. The two key features among standards are summarised below:

- Safety assessment activities are centralised around the concept of Safety Integrity Levels (SILs). Under this notion, safety can be viewed as a controlled attribute that is affected by safety requirements. In brief, safety requirements are allocated to system functions based on a risk analysis of the defined hazard. Then, SILs are progressively decomposed and re-allocated to lower-level systems and components (hardware and software), responsible for implementing those functions.
- They propose a similar framework for the product development, known as the V-model (more details in Section 3.4), wherein requirements specification, design and implementation activities are achieved in a top-down approach, and integration, testing and verification follow a bottom-up approach.

As mentioned earlier, there are discrepancies among standards. For instance, the decomposition of SILs typically follows a set of rules, which vary across industrial domains. However, the process is practically identical, as re-allocation is proposed in a top-down manner that respects the system architectural dependencies. Another example is the criteria considered when assigning SILs. In civil aircraft, DAL allocation is mostly based on the severity of the identified hazard, whereas in the automotive sector, ASIL allocation is also affected by the exposure (i.e. frequency and duration of exposure of a system-user to a hazardous event) and the controllability (i.e. the ability of the system-user to gain control of the hazardous situation and avoid harm).

The overview of safety standards has highlighted the aforementioned commonalities. These elements have been one of the founding layers of the common framework for safety assessment, across industrial domains, integrated in the proposed method. Considering that certification of safety-critical systems is typically demonstrated by compliance with safety standards, it is plausible that the method described later in this thesis is compatible with certification approaches.

### 3.3 Regulations and Certification

Certification of safety-critical systems has one main objective; to verify and establish confidence to the public that a product or service is safe. In general, the process involves the system developer or service provider, the standard or regulation, and an independent authority. The regulations may be enforced on a national or even international level to cover interoperability affairs. Note that even though certification is provided from third-party bodies, it is the production team that ultimately holds responsibility for both the justification of system safety and accidents caused from system failures (during its lifecycle). The justification of safety encompasses the conduct of safety assessment activities, prescribed in safety standards, and an argument of acceptable safety along with its supporting evidence. The latter is typically captured in a set of documents, the safety case. As a result, safety cases are valuable for certification purposes and have been used in various domains with a few discrepancies in their structure or content. In practice, safety cases are produced progressively and maintained throughout the system's life span.

#### 3.3.1 Certification and Regulations for Civil Aviation

The certification process in the civil aviation industry is controlled mainly by the International Aviation Organization (ICAO) and the Federal Aviation Administration (FAA) or the European Aviation Safety Agency (EASA), for the US and the European regions, respectively. In this domain, certification revolves around the overall safety of the aircraft and its embedded systems. Even though authorities recognise software as a potential contributing factor towards system safety, they regard software to be a part of the system (i.e. a unity of hardware and software). As a result, despite the existence of a separate guideline (DO-178C), software-safety is not used explicitly; instead,

guidelines focus on the software-side of the system and address safety-related concerns mostly through processes for software correctness (i.e. failure avoidance by elimination of software errors and causes) (Blanquart et al., 2018:2). The FAA and EASA share common principles and the provided regulations among their corresponding regions have minor differences. The regulations from ICAO are region-neutral and therefore can be used as a baseline when there are conflicts between regional regulations; in such cases, developers either request advice from the ICAO or consult its guidelines. The base documents for certification of software and hardware systems in civil aviation are described by the Federal Aviation Regulations (FAR) (USA Government, 1964) and currently by the Code of Federal Regulations (14 CFR) as found in (FAA, 2018). In addition, the Advisory Circular document (AC 25.1309) was produced to provide support for showing compliance with the FAR for the design and manufacturing of software and hardware systems (FAA, 1988:2). The counterpart of FAR and 14 CFR for the European regions is the CS-25 (EASA, 2011).

In 1995, the Society of Automotive Engineers (SAE) published the Aerospace Recommended Practice (ARP4754) as a technical paper to further aid developers produce aircraft systems in compliance with the FAR regulations. Technically, the ARP guideline consists of two documents; however, due to its widespread acceptance it is commonly regarded as a standard. After its revision in 2010, the ARP4754-A focuses on the general framework for the development of aircraft functions and systems. The ARP4761, on the other hand, is the process handbook that demonstrates through examples how safety assessment processes, described in ARP4754-A, can be performed. The concept of SILs is fundamental in the ARP standard, yet they are referred to as Development Assurance Levels (DALs). Another set of supplementary documents has been published by the Radio Technical Commission for Aeronautics (RTCA) to provide additional guidance towards the development of specific components. This series is known as Document Object (DO) and the most notable documents are the following: a) DO-178C for software components, b) DO-254 for hardware components and c) DO-297 for integrated avionics. Fig. 3.1, adapted from (SAE, 2010:6), displays the

interrelationships of guidelines for system development, as well as hardware and software processes in the following diagram.

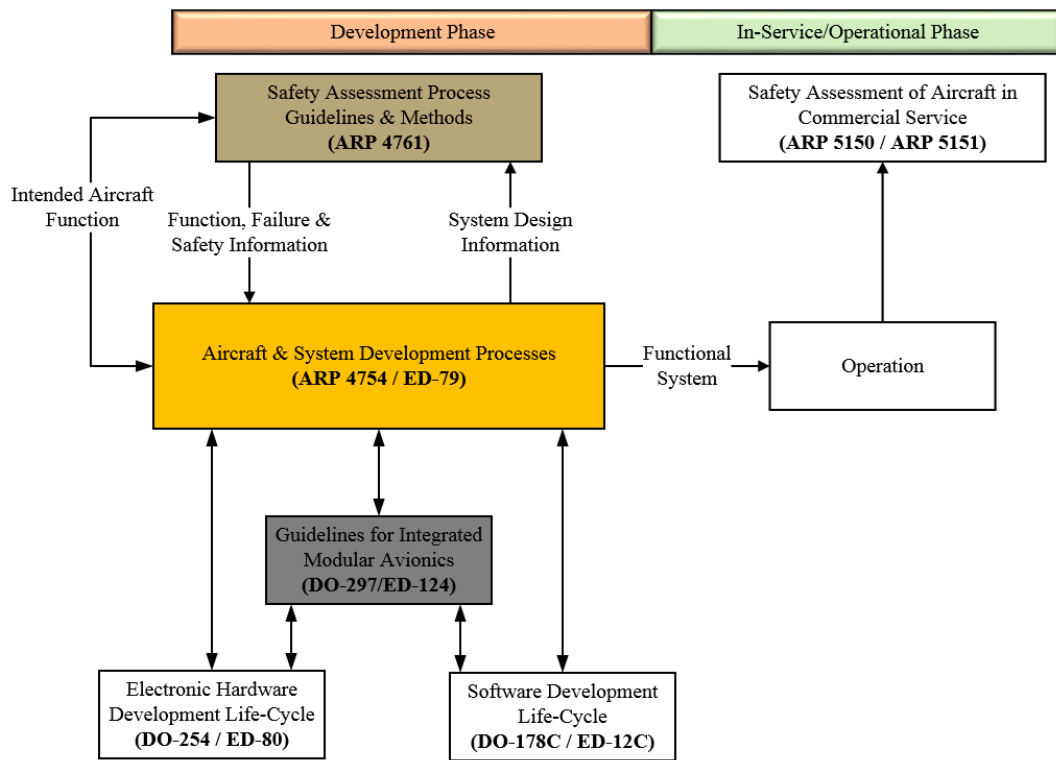


Figure 3.1: Aircraft system guidelines relationship diagram (SAE, 2010:6)

The European Organization for Civil Aviation Equipment (EUROCAE) has also developed a series of guidelines that is referred to as ‘EUROCAE Documents’ (EDs). For example, the European counterparts to the ARP-4754 and DO-254 are the ED-79 and ED-80, respectively. As mentioned earlier, there are no major differences between the two sets of standards. Even though the proposed method focuses on the SAE series, it can be used equivalently for the EUROCAE.

### 3.3.2 Certification and Regulations for Automotive

The Automotive industry follows a different certification regime. Although there is no distinct authoritative body, developers still have to comply with regulations on specific technical domains in order to attain permission in the corresponding market. For example, electric vehicle warning sounds follow the UN R138 regulation in the EU or the FMVSS in the US. After four years of development and refinement, the ISO/TC22/SC32 committee published the ISO 26262 in 2011 as a guideline for

functional safety in electrical and electronic systems installed to passenger vehicles up to 3500kg mass. As mentioned in Section 3.1, the framework also supports other technologies (e.g. hydraulic). In addition, the standard focuses mostly on hazards related to system behaviour rather than direct hazards (e.g. electric shocks). The ISO 26262 is meant to be integrated with the individual developer's framework and not overrule it. In terms of structure, it is segmented into 10 chapters whose content varies from technical-focused requirements to development process requirements. For example, chapters 3 and 4 describe the concept phase and product development at system level, respectively. During the former, the developers define the high-level functions and items and perform hazard analysis and assessment, whereas during the latter they specify the requirements related to the system during design, verification and testing. These stages of safety assessment are similarly described in the ARP series, for the civil aviation domain. As another example, ISO's chapter 6 focuses on the development and assessment of software components (ISO, 2011:245), which is comparable to the guidelines found in the DO-178C for civil aviation (RTCA, 2011). Once the safety activities and functional concept are complete, developers should provide a safety case that describes how all processes and work products (results from ISO recommended methods) justify their claims of system safety.

### 3.4 Development and Safety Lifecycles in Modern Safety Standards

Many safety standards adopt the 'V-model' system development lifecycle, as shown in Fig. 3.2. On the left, developers must define the system's general requirements, propose a design and iterate these steps until the architecture is complete. Then, through implementation, integration and testing, the final system is put into production and eventually operation.

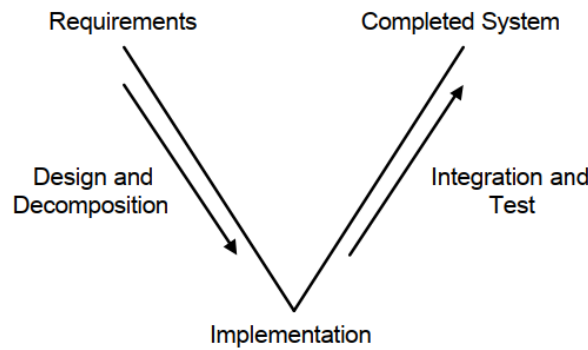


Figure 3.2 The V-model of system development (Kelly, 1998:67)

Authorities and developers have recognised that safety concerns should be addressed as soon as possible and therefore assessment and assurance should not be left for later stages. As a result, functional safety standards provide a framework where the safety lifecycle is merged with the product development efforts and acts in accordance with the V-model. The next sections examine how this framework is described in the automotive and aviation industries.

#### 3.4.1 The Safety Lifecycle in Civil Aviation

The development phases prescribed in ARP are depicted in a diagram form, adapted from (SAE, 2010:20), in Fig. 3.3.

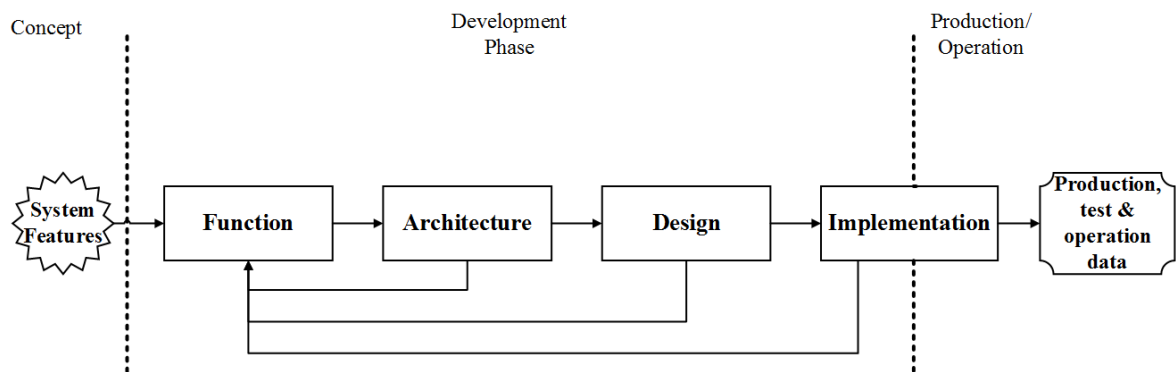


Figure 3.3: ARP Development lifecycle (SAE, 2010:20)

The concept phase involves the establishment of general features regarding the product and is considered to be a preliminary development phase. Ideas regarding the purpose, overall performance, application of contemporary technologies and other traits are discussed and once the stakeholders come to a consensus, the development phase commences. For the design and implementation of

architectural components, the standard utilises three basic types: a) **functions**, b) **systems** and c) **items**. Functions describe the nominal behaviour of airborne systems; when they refer to high-level functionality, such as ground deceleration or flight control, they are referred to as “aircraft functions”. Any low-level elements, either hardware or software, that cannot be segregated any further are indicated as items. Systems are mostly utilised as management elements; in particular, systems capture multiple components within the architecture and either individually, or in conjunction with other systems, support system or aircraft functions. The development phase resembles the waterfall model and is performed in an iterative manner. Initially, the basic functions are identified (i.e. Function), and engineers proceed with the design of the corresponding systems (i.e. Architecture). As this process progresses, the functions are further refined and decomposed into subfunctions and consequently more systems and subsystems are introduced in the model. The process is complete when the architecture satisfies all the requirements set during the concept phase and systems have been refined down to the lowest-level components. The final phases of design and implementation on the diagram represent the development of items based on the latest architecture.

Under the ARP guidelines, a set of safety assessment activities accompany the system development. The framework is adapted to the V-model and is segmented into two major process categories (SAE, 2010:13):

- **Validation** activities that enable developers to determine if all the identified requirements are correct and complete (i.e. Are the systems or functions appropriate?).
- **Verification** activities that allow the current system implementation to be tested and help confirm whether or not it meets the requirements (i.e. Has the system been built correctly?).

Fig. 3.4 shows the ARP-specific V-model adapted from (Kelly, 1998:67). The left part of the figure illustrates the development phases as described earlier for the core V-model, whereas the right part demonstrates how all the safety assessment activities are reflected on this scheme.



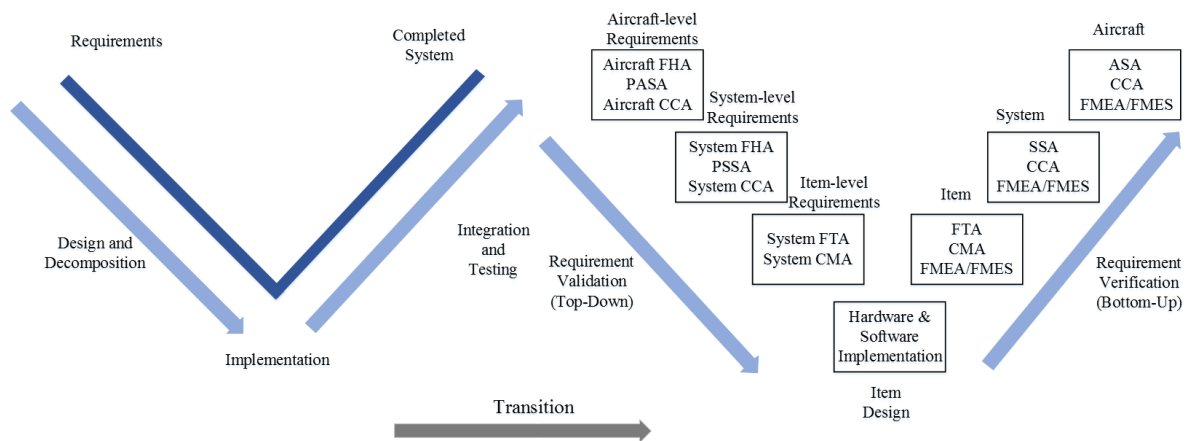


Figure 3.4: Development lifecycle under the V-model (Kelly, 1998:67)

Note that during validation and verification, the assessment activities follow a top-down and a bottom-up approach, respectively.

Now that the notion of functions, items and systems and the development lifecycle under the ARP4754-A are provided, it is possible to investigate the safety assessment activities for each phase. Once the general system parameters have been decided by the stakeholders, the development proceeds with the identification of high-level functions via the functional analysis discussed in Section 2.2.2. At this stage, safety engineers perform FHA to consider any associated hazards and determine the consequences and probability of occurrence. The standard refers to hazards as Failure Conditions (FCs) and suggests that development, apart from failures and errors, should also consider operational and environmental conditions. Based on each FC's effects (e.g. severity) on the aircraft (system as well as crew and passengers), the engineers assign the appropriate DALs and may also set additional safety objectives if stated in regulations. The functions are also allocated with a DAL identical to the FC; in case of multiple FCs, it receives the DAL from the FC that is considered the most severe (i.e. most stringent requirement). As the functional model becomes more detailed with new emerging subfunctions, the process above re-iterates so that requirements reflect on the current architecture.

To evaluate the system architectures that support the aircraft-level functions and their supporting systems, the standard prescribes a Preliminary Aircraft Safety Assessment (PASA) and a Preliminary System Safety Assessment (PSSA). These analyses help engineers understand how failures in the

target architecture could cause the identified hazards and define additional requirements. Further, during PSSA, DALs (previously assigned only on lower-level functions) are allocated to subsystems that directly support those functions or their systems. If any functional hazard depends on the failure of multiple systems, then these systems can be assigned lower DALs, based on a set of rules established within the standard.

At this point, a Common Cause Analysis (CCA) is conducted on the high-level architecture with the aim to examine the correctness of the assumptions (in earlier analyses). The CCA essentially investigates whether or not the functional requirements have been distributed correctly across systems. If any of the assumptions were wrong and led to improper requirements, then the allocation process has to be repeated. Once this cycle of analyses is complete for the aircraft functions, the process is repeated for subsystems and until systems are only supported by items. The result of this effort is a preliminary system architecture, a set of hazards along with associated risk, and a set of requirements (DALs) that is allocated throughout the elements, from functions to systems and items. Before the implementation begins, the requirements must be examined for validity with a variety of methods (the standard suggests but does not enforce any particular method) discussed in Sections 2.3 & 2.4. Other validation methods include: a) traceability, where the developers follow how lower-level requirements are linked to the higher-level requirements and document the rationale, b) simulation and item testing and c) engineering reviews, where the analysts examine the produced model, based on their experience.

Once the architecture is determined to be stable and validated, development proceeds with the implementation of components (Joshi et al., 2006). The items implemented enter the verification stage. Verification methods involve either testing or formal methods, which were summarised in earlier chapters. This process neatly follows a bottom-up approach; that is, when verification for items is complete, it extends to systems via system safety assessment (SSA) and finally functions via aircraft system assessment (ASA). Note that during these stages, developers may also employ the FMEA and

FTA techniques, or similar, to confirm the correctness of requirements. When verification for all aircraft functions is complete, development proceeds to the production stage (i.e. manufacturing).

### 3.4.2 The Safety Lifecycle in Automotive

The framework in ISO 26262 is separated into the concept, development and production phases. Fig. 3.5 depicts a simplified version of the ISO 26262 lifecycle adapted from (ISO, 2011:42).

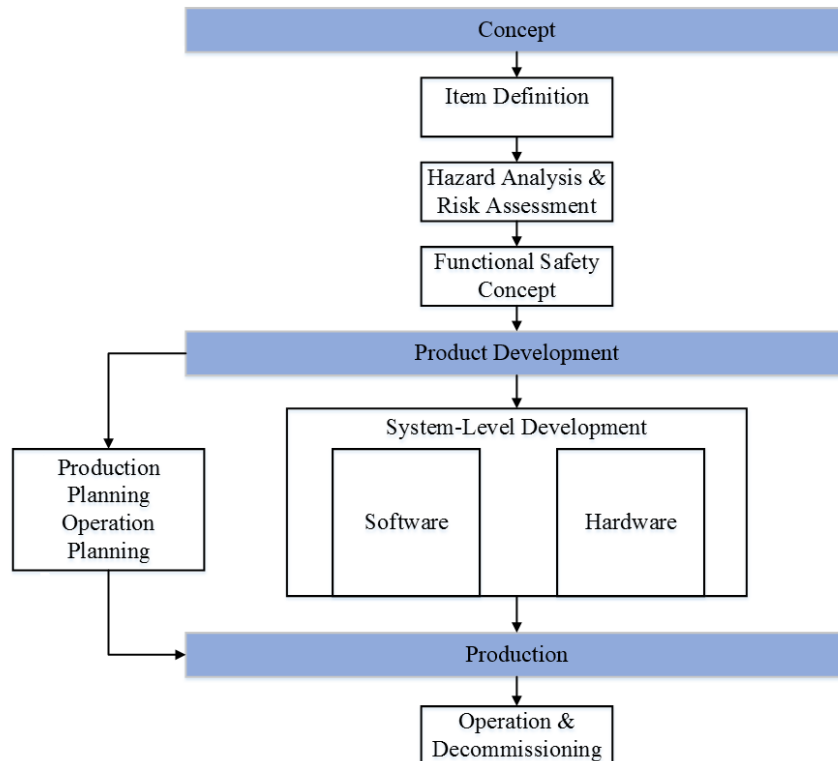


Figure 3.5: Simplified Version of ISO 26262 lifecycle (ISO, 2011:42)

During the first phase, engineers identify a set of items and their functionality, potential hazards and safety goals. Items can be either a system or a set of systems that substantiate a high-level vehicle function. Safety goals are the high-level safety requirements. Once the hazards are established and their risk is estimated, they are assigned requirements in the form of ASILs. Similarly to ARP, ISO 26262 allows the developers to utilise various safety analysis techniques, such as HAZOP, FTA and FMEA. This allows the investigation of functional redundancies as the architecture is refined, and the allocation of ASILs to lower-level elements accordingly. The outcome of this phase is the functional safety concept, which is the system architecture with specified safety requirements designed in a way

that achieves the safety goals (ISO, 2011:77). Moreover, other information, such as item interactions, operational constraints, assumptions with regards to the item behaviour and legal requirements, are also documented. The next phase focuses on the design, safety analysis (FTA, HAZOP and FMEA) and implementation of architectural elements (software or hardware). Validation and verification activities, similarly to the aviation industry, follow the V-model (ISO, 2011:110). Fig. 3.6, adapted from (ISO, 2011:107), shows how the ISO has adjusted and integrated the V-model.

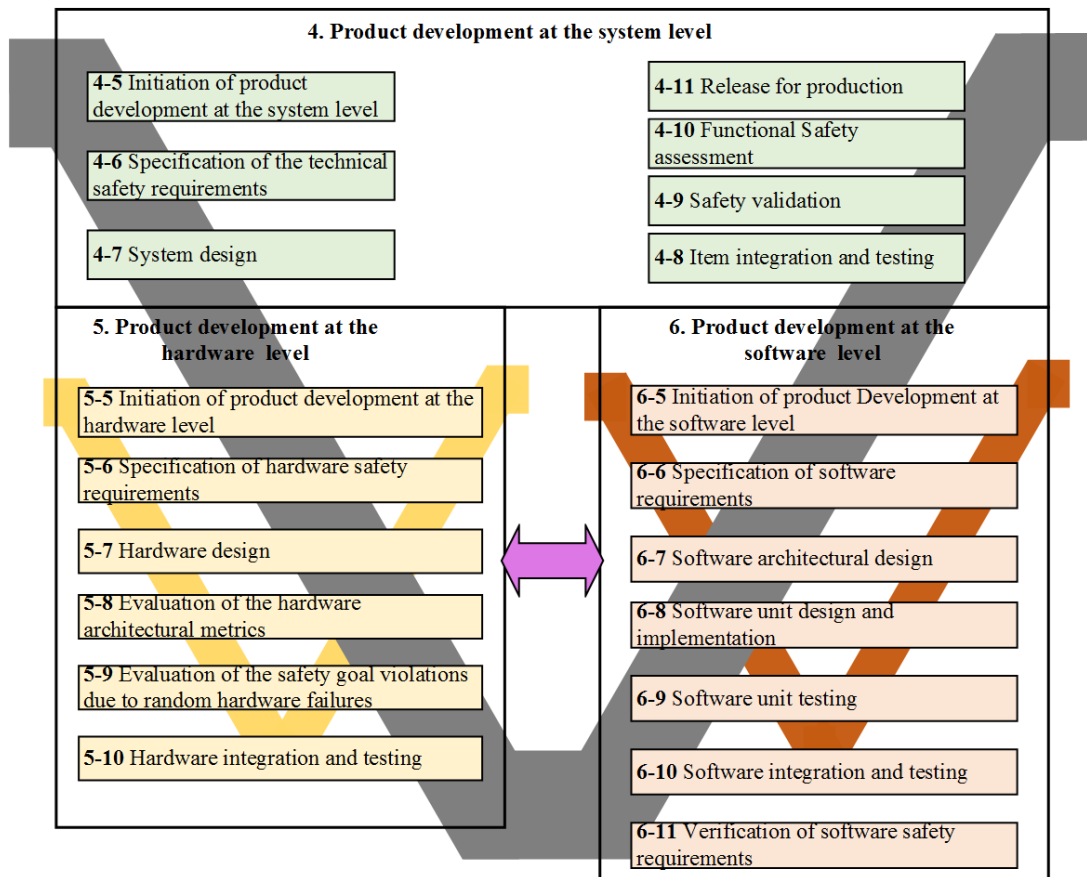


Figure 3.6: The V-model adaptation in ISO 26262 (ISO,2011:107)

Once the final model is complete, meaning that safety goals are satisfied and all requirements verified, the system may proceed to the production phase. The main objective is to maintain functional safety during the production process, whether it is during the installation of E/E parts or system manufacturing processes. Compliance for items with safety-related unique attributes is established during development and is transferable to the production phase. These item characteristics vary from specific manufacturing idiosyncrasies (i.e. temperature tolerance) and if production does not meet

these specific requirements, it could discredit the functional failure supported during development (ISO, 2011:295). An example, adapted from (ISO, 2011:113), of how an integrated system (with subsystems) is developed under the ISO 26262 is demonstrated in Fig. 3.7 below.

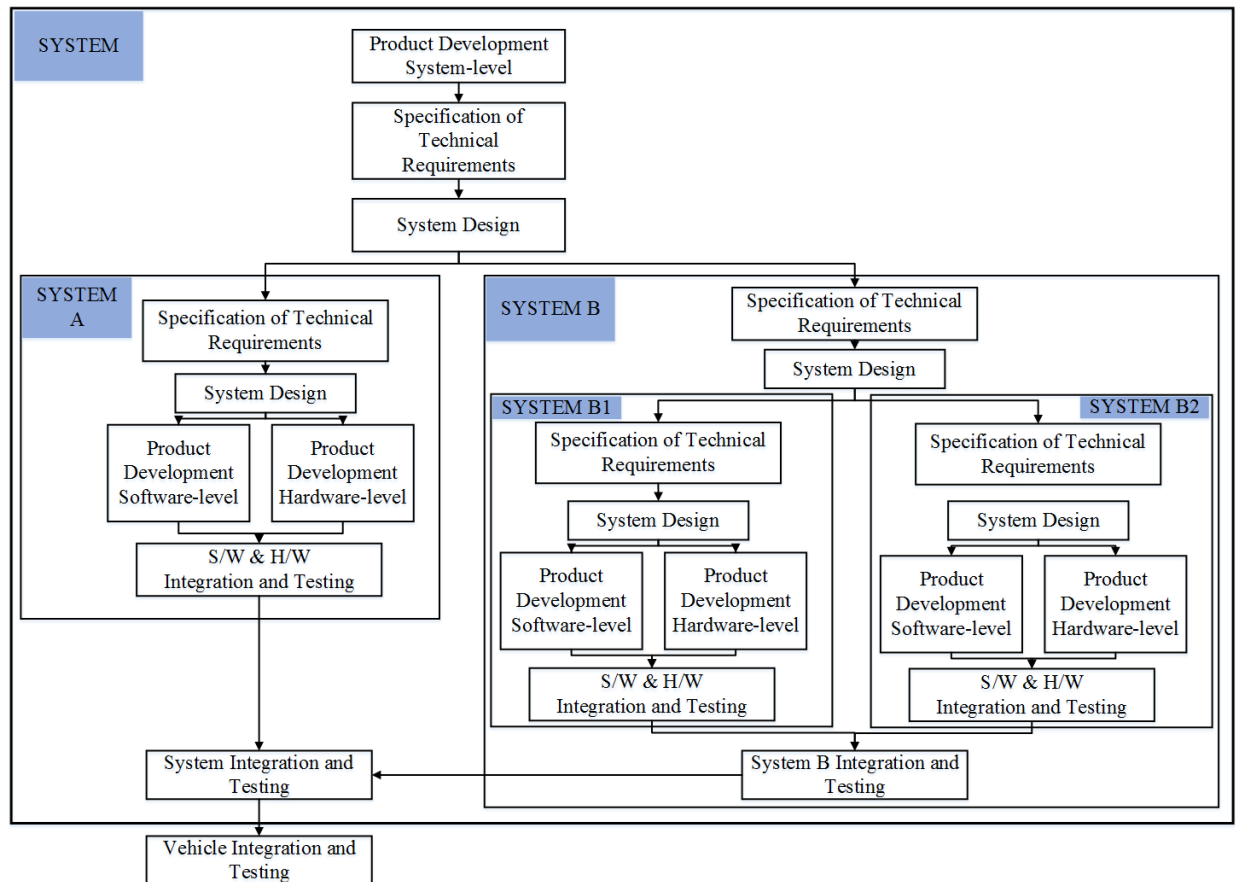


Figure 3.7: Diagram of System Development (ISO, 2011:113)

The diagram above showcases a general system that is to be installed in a vehicle. The system is composed of two subsystems, A and B. Subsystem B comprises two additional subsystems, B1 and B2. For each of the system and subsystems, the developers specify the requirements, produce a preliminary model, and proceed with analysis and development of their software and hardware elements. Note that for the latter, ISO 26262 recommends specific methods to provide substantial evidence based on the allocated requirement. The appropriate methods per item are portrayed in a tabular format for each ASIL, and the standard provides a symbol to indicate: a) high recommendation, b) recommendation or c) optional use. For example, Table 3.1 below is a reproduction of table 5 originally found in (ISO, 2011: 127).

Table 3.1 Example of methods for correct implementation of hardware-software level (ISO, 2011: 127)

Methods	ASIL			
	A	B	C	D
Requirements-based test	++	++	++	++
Fault injection test	+	++	++	++
Back-to-back test	+	+	++	++
* ++ indicates high recommendation, + indicates recommendation, O indicates optional				

The purpose is to lay out a set of test methods suitable for demonstrating the correct implementation of technical requirements during software-hardware integration. Validation and verification activities are performed for each subsystem individually, and once the subsystems are integrated, for the system as a whole. Finally, the main system is then ready to be integrated with the rest of the vehicle's systems.

### 3.5 Safety Cases in Automotive and Civil Aviation

Past approaches relegated safety case construction to the later stages of development. The main objective was to create a report, with clear arguments, able to assure authorities and regulatory bodies that the system is safe. This approach was found to be problematic. In cases where the authorities were not convinced, the system may have had major parts redesigned and re-implemented, which could lead to months of additional development. Further, with improper documentation of the processes involved, the rationale was hard to track and therefore the decision-making was inadequately argued. As a result, modern safety standards that incorporate the concept of safety cases in the safety lifecycle, such as the ISO 26262 and ARP4754-A, suggest that the production of the safety argument should follow the V-model. For instance, the ISO 26262 suggests that 'the safety

case should **progressively compile the work products that are generated during the safety lifecycle**’ (ISO, 2011:51). As another example, the DS 00-56 standard requires that the synthesis of the safety case ‘should be initiated at the earliest possible stage’ (MoD, 1996a:10) and JSP 430 suggests that it should be updated at each major milestone (MoD, 1996b). To conform to a gradual development of safety cases, maintenance and submission of the safety argument should follow major phases in the lifecycle. DS 00-55 proposed the creation of at least three versions (MoD, 1997:E1), summarised below:

- **Preliminary Safety Case** that could be constructed immediately after the system definition and requirements identification.
- **Interim Safety Case** that could be produced after the initial system design and validation activities are completed.
- **Operational Safety Case** that could be established just before the production phase, when there is substantial evidence that all requirements are satisfied.

Following such an approach, there are benefits regardless of the number of safety case versions. Safety assurance is incorporated in the development lifecycle, which allows for better evaluation of the current system state in terms of safety. Any assumptions, processes or activities utilised by the developers can be assessed and changed when necessary. Fig. 3.8 indicates how the different versions of the safety case, as suggested by the DS 00-56, could be mapped on the ‘V-model’.

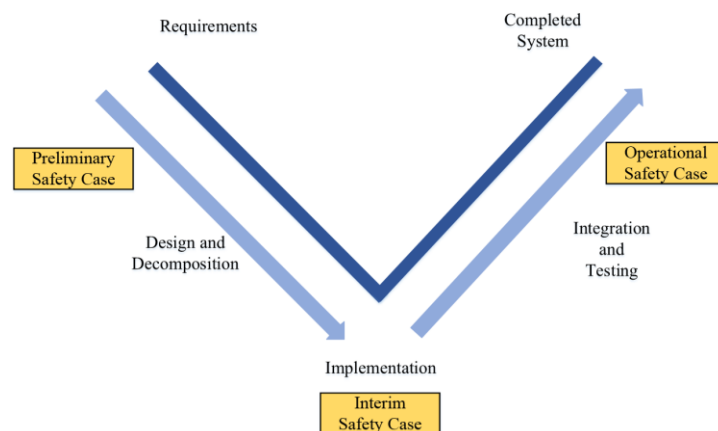


Figure 3.8: Versions of the Safety Case mapped on 'V-model'

The ARP4754-A does not provide any information related to safety case production phases and information is limited to the documentation of the results from assessment processes. However, by adopting the number of versions shown in Fig. 3.8, it is possible to define suitable milestones. The construction could start once the PSSA was completed, then the safety case could be updated just before the implementation stage, when the results from the validation of requirements are satisfying and finally proceed to the third version after the functional SSA (i.e. verification of aircraft functions).

In the automotive industry, even though a safety case is required to show compliance with the ISO 26262, the standard does not explicitly guide the developers through the construction of a safety argument nor how to evaluate it. Since there are no constraints with regards to the time of the construction and due to the similarities of the lifecycle between functional standards, it is plausible to create and maintain the safety argument after each major development stage; therefore, construction could start once the concept phase is completed and maintained throughout the product development and production phases.

As for the structure of the argument, none of the standards provide a definitive form or contents. However, the ISO 26262 suggests that: 1) 'a safety case should argue that all safety requirements for an item are complete and satisfied by evidence compiled from the work products of safety activities during development' and 2) 'a safety case could even extend to cover safety beyond the scope of ISO 26262' (ISO, 2011:20). The latter signifies that a complete safety case communicates system safety beyond the E/E/PS systems covered by the ISO. Thus, considering the ISO's description and other implicit descriptions from safety standards, we are led to the definition of key elements in safety argument synthesis:

- Hazard identification and risk analysis artefacts from the conceptual architecture
- Mitigation means in the form of requirements and safety goals
- Evidence that requirements are satisfied throughout the lifecycle stages
- Validation and verification results that solidify the chosen requirements and establish that the implemented architecture is correct against those requirements



Utilisation of the above elements could result in an argument of safety that demonstrates functional safety or freedom from unreasonable risk. Specifically, the argument would communicate how all the hazards identified have been prevented or mitigated to an acceptable level through satisfaction of requirements or other measures. These elements are adequate for building arguments that justify system safety against the system properties and requirements (product-based arguments). However, a safety case should also communicate that the processes involved in the development are standard-compliant too. Further, process-based arguments in the automotive industry may also take into consideration additional elements responsible for the generation of the work products, such as correctness of supporting tools and personnel's competency (Birch et al., 2013).

### 3.6 Safety Integrity Levels in Aviation and Automotive

As discussed earlier, both the ARP4754-A and ISO 26262 safety standards have adopted the concept of SILs and incorporated it in the development lifecycle. Despite the standards sharing a similar basis as to what SILs represent, they have distinct differences in how SILs are allocated for each hazardous event. Further, the decomposition rules to lower-level components also vary from one standard to another. The following sections examine how SILs have been adapted in the two industries and also demonstrate, via a simple example, how SIL decomposition is performed.

#### 3.6.1 SILs in the Civil Aviation

##### *3.6.1.1 DALs Classifications*

Development Assurance Levels (DALs) is the adaptation of SILs within the ARP standard. The first allocation of DALs can start once the hazard analysis for aircraft functions is completed. The standard defines five DALs: E-A from least to most stringent. For each identified hazard, a DAL is appointed based on the potential effects. Table 3.2, adapted from (RTCA, 2011:13), shows the appropriate DAL per severity classification and the effects description.

Table 3.2: DAL per severity classification (RTCA, 2011:13)

DAL	A	B	C	D	E
Severity of Hazard	Catastrophic	Hazardous	Major	Minor	No Safety Effect
Description of Effects	<ul style="list-style-type: none"> <li>Loss of airplane</li> <li>High-mortality accident</li> </ul>	<ul style="list-style-type: none"> <li>Serious injuries and/or low-mortality accident</li> <li>Seriously restricted functionality and/or safety margins</li> </ul>	<ul style="list-style-type: none"> <li>Major physical discomfort for both crew and passengers</li> <li>Significantly restricted functionality and/or safety margins</li> </ul>	<ul style="list-style-type: none"> <li>Minor physical discomfort for both crew and passengers</li> <li>Slightly restricted functionality and/or safety margins</li> </ul>	No effects

Based on this table, a hazard that could be the cause of multiple fatalities or loss of the airplane is characterised as catastrophic and allocated with DAL A. Another attribute that affects the DAL classification, and typically identified during the FHA, is the **probability of occurrence**. This acts as a quantitative condition that complements the DAL requirement. It is measured as occurrences per flight hours and is captured within the corresponding component or system (EASA, 2011:8). For example, the hazard characterised as ‘major’ may not have more than  $1E-5$  and lower than  $1E-7$  probability per flight hour. These classifications and their connection to probabilistic data are based on knowledge or prior experience (SAE, 1996: 181).

### 3.6.1.2 DALs Allocation and Decomposition Rules

The allocation of DALs begins just after the functional FHA, when all the failure conditions (FCs) for the top-level aircraft functions have been identified. For each of the FCs, the aircraft function is assigned a DAL based on the highest severity classification of that FC. Once the preliminary

architecture (subfunctions and systems) is defined, developers may proceed and allocate DALs to the rest of the architecture in a top-down manner. To do so, ARP4754-A provides the Functional Failure Sets (FFSs), a concept identical to the minimal cut sets. System safety assessment techniques, such as FTA or Markov Analysis, are conducted to identify all the FFSs, and their members, capable of causing the target (top-level) FC. Each FFS contains the minimum number of members which in combination can lead to top-level FC. If the FFS contains only one member, then that element receives the same DAL as the FC. In cases where the FFS has two or more members, then the elements are assigned DALs based on the following options (SAE, 2010:44).

- One of the members of the FFS that contribute to the top-level FC is assigned the same DAL with the parent system, whereas the rest of the members are assigned a DAL equal or up to two levels lower than the top system.
- Two of the members of the FFS are assigned a DAL one level lower than the parent system, whereas the remaining members are assigned a DAL equal or up to two levels lower than the top system.

Note that both options are valid, and the decision is up to the engineers and implementation needs.

Even though only two options are available, the overall number of alternatives is subject to combinatorial explosion, as FFS members increase in number across all FCs. Moreover, the allocation of DALs is an iterative process since new functions or systems might have an impact on system safety. Therefore, a manual approach may result in significant time consumption. To alleviate these issues, recent research focused on automating the DAL allocation process (Bieber et al., 2011; Azevedo et al., 2013; Sorokos et al., 2015).

### 3.6.2 SILs in Automotive

#### 3.6.2.1 ASILs Classifications

SILs in the ISO 26262 are known as Automotive Safety Integrity Levels (ASILs). The standard uses a risk-based scheme to assign ASILs to hazardous events. As mentioned in earlier sections, the allocation of requirements begins during the concept phase. Once developers define all the hazards

for each of the vehicle's operational situations, a process of risk assessment allows them to determine the severity, exposure and controllability per each hazard (see Section 3.2). ISO 26262 provides a set of tables to guide the parameters classification based on specific criteria. Tables 3.3, 3.4 and 3.5 shown below, present the severity, exposure and controllability classes respectively, as adapted from (ISO, 2011).

*Table 3.3 Classes of Severity (ISO, 2011:83)*

Class	S0	S1	S2	S3
<b>Description</b>	No injuries	Moderate Injuries	Severe Injuries (survival probable)	Life-Threatening injuries (survival uncertain), fatal injuries

*Table 3.4 Classes of probability of exposure based on operational situation (ISO, 2011: 83)*

Class	E0	E1	E2	E3	E4
<b>Description</b>	Incredible	Very low probability	Low probability	Medium Probability	High Probability

*Table 3.5: Classes of controllability (ISO, 2011:84)*

Class	C0	C1	C2	C3
<b>Description</b>	Controllable	Simply Controllable	Normally Controllable	Difficult to control, Uncontrollable

Based on the above parameters, the hazardous event is classified and assigned an ASIL. There are four categories of ASILs that range from A to D, where D is the most stringent and A the least stringent. Events that do not pose any safety concerns are annotated with QM (i.e. Quality Management handles development for these requirements). Table 3.6, adapted from (ISO, 2011:84), reveals how the combination of the parameters formulate the appropriate ASIL.

Table 3.6: ASIL determination table (ISO, 2011:84)

Severity Class	Exposure Prob.	Controllability Class		
	Class	C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

### 3.6.2.2 ASILs Allocation and Decomposition Rules

Following the process of ASIL allocation to hazardous events, a safety goal (SG) must also be determined for each of these events. The SGs represent higher-level safety requirements for items and ultimately lead to the functional requirements necessary to prevent or mitigate unreasonable risk. Note that SGs may be achieved with transition to safe states, such as stationary vehicle or switch-off; therefore, every possible state transition should be specified and later be reflected on the functional safety concept (ISO, 2011:86). Since SGs are decided based on the hazardous event, it is only natural to assign them with the same ASIL. From that point on, the standard follows a top-down approach for allocating safety requirements. In essence, the ASILs of safety goals propagate throughout system (or item) development and are refined into lower-level requirements, which in turn are later assigned to the components of the system. First, SGs help derive the functional safety requirements (FSRs), which describe a safety behaviour or measures and are compliant with the SGs. Then, from the FSRs, a set of technical safety requirements (TSRs) are specified, which describe how FSRs are

implemented on the system-level. Finally, from TSRs, software and hardware requirements are formulated and assigned to the corresponding elements. In cases where the architecture establishes independent elements (i.e. elements whose failure is probabilistically independent and do not violate SGs), requirements can be implemented redundantly by these elements and the standard permits the allocation of lower ASILs. The decomposition arrangements are diagrammatically shown in Fig. 3.9 as adapted from (ISO, 2011:378).

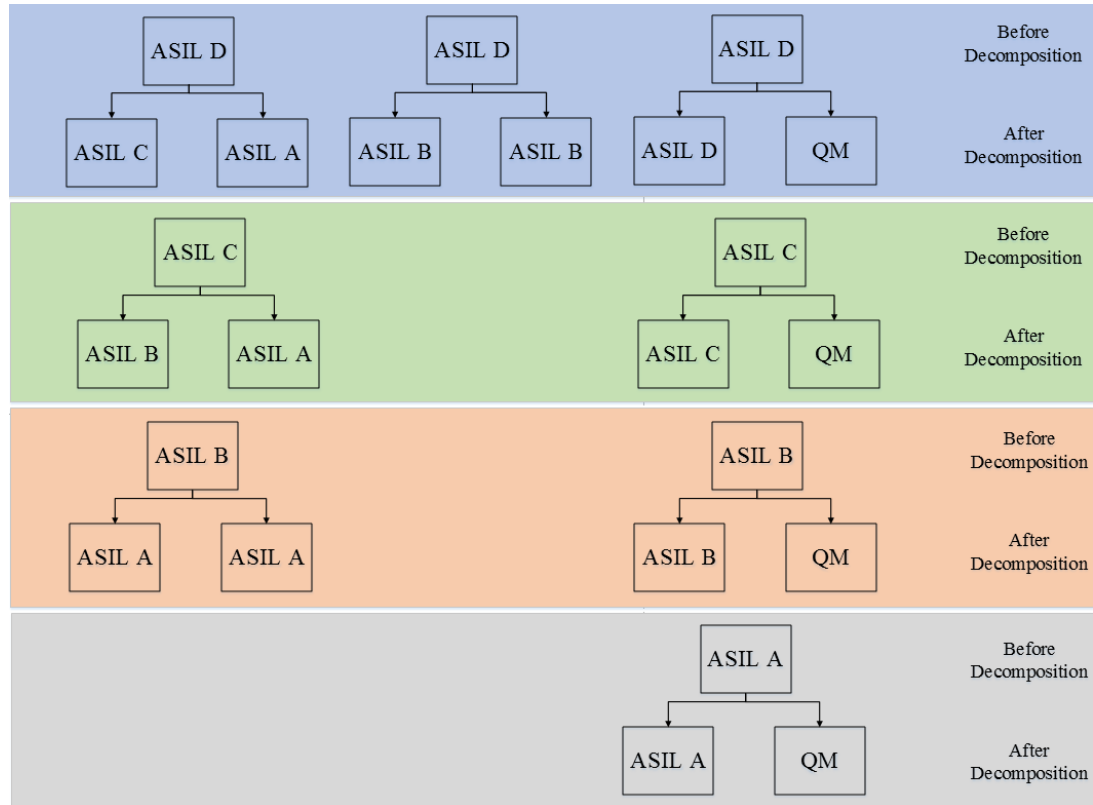


Figure 3.9: ASIL decomposition diagram (ISO, 2011:378)

The decomposition can be valuable during development as it allows developers to preserve safety by satisfying the high-level requirements of safety goals without enforcing them to develop all elements with highly demanding processes.

### 3.6.3 A Decomposition Example with DALs

SILs dictate the stringency of safety assessment activities during system development; highly demanding processes are often linked to incremental costs. As a result, different SIL configurations for a given architecture may lead to differences in development costs. This section demonstrates how

DAL decomposition is performed on an abstract system model. This example, also shown in (Retouniotis et al., 2017), aims to exhibit how the rules, prescribed by the standard, are used in practice and what are the effects on development costs for different DAL configurations. Fig. 3.10 depicts the system architecture.

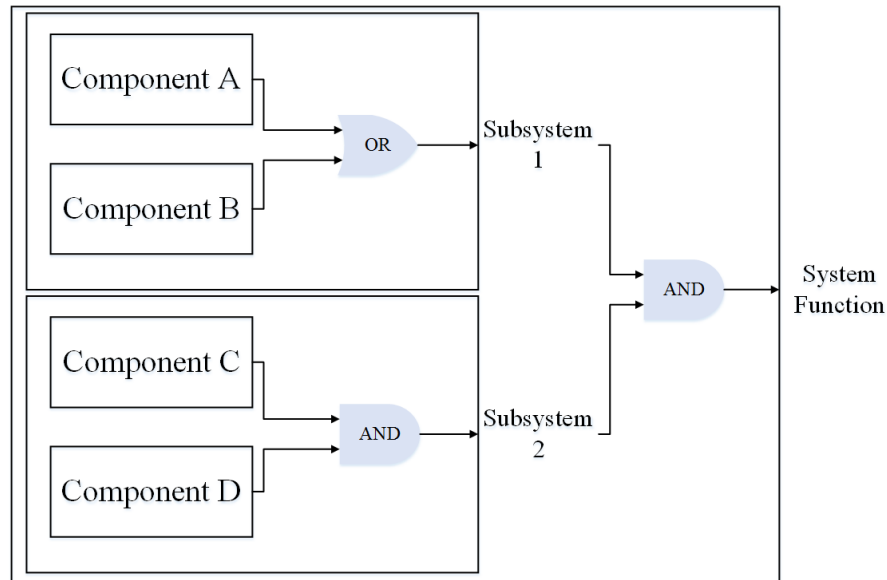


Figure 3.10: Abstract system model

The system has a single output and comprises two subsystems, which in turn include two components each: A and B, C and D. The logical gates symbolise the failure propagation between components and subsystems that consequently lead to the functional failure of the main system. Note that subsystems 1 and 2 do not impose any internal failures (i.e. basic events) that contribute to the system failure. Additionally, it is established that all the elements are shown to be appropriately independent.

After system analysis, it is concluded that the only two FFSs are:

- **FFS 1**, with member components A, C and D
- **FFS 2**, with member components B, C and D

Typically, the number of available options increases based on the population of FFS members. In this scenario, there are only nine options from the combination of possibilities across the two FFSs. Assuming the system function was initially assigned DAL A (from PSSA), application of the ARP4754-A rules results in the following decomposition options, shown in Table 3.7 below.

Table 3.7: DAL Decomposition Options for Components

Element Name	Component A	Component B	Component C	Component D
<b>Option 1</b>	A	A	C	E
<b>Option 2</b>	A	A	E	C
<b>Option 3</b>	A	A	D	D
<b>Option 4</b>	C	C	A	C
<b>Option 5</b>	C	C	C	A
<b>Option 6</b>	C	C	B	B
<b>Option 7</b>	B	B	B	D
<b>Option 8</b>	B	B	D	B
<b>Option 9</b>	B	B	C	C

All of the above options are valid; however, given the development costs for each item per DAL, engineers can estimate the most cost-effective option. Table 3.8 below shows indicative costs for illustrative purposes. In practice, the cost of development between systems with the same DAL is not necessarily equal.

Table 3.8: Indicative cost catalogue for DALs

DAL	A	B	C	D	E
<b>Cost</b>	100	80	40	20	0

With simple addition, the cost of each configuration is determined. In this example, many options have identical outcome. Specifically, options 1, 2, 3, 6 and 9 have a cost of 240, options 4 and 5 have a cost of 220, and finally, options 7 and 8 add up to 260. Options 4 and 5 seem to be the most cost-effective. The simplicity of this example translates into a rather trivial solution. If the system had more components and slightly more complex design or even a realistic cost catalogue for each item per DAL, then the solution would not be so apparent. The more complex a system is, the more extensive the design space of available options becomes. This renders the exhaustive search for optimal solutions into an intractable problem that manual approaches fail to solve adequately.



### 3.6.4 SIL Allocation as an Optimization Problem

Optimisation can be conceived as the process of exploration with the aim to find the best available solution to a specific problem. In its simplest form, an optimisation problem is the pursuit of maximizing or minimizing the output of a function, given a finite number of inputs. More complicated cases introduce sets of constraints on the function. The allocation of SILs can be viewed as a constrained optimisation problem, in which:

- the **objective** is the minimisation of the development costs
- the **constraints** are the decomposition rules defined by the standard
- the **decision variables** are the SIL values of items/functions in the architecture

The next section begins with a brief overview in optimisation methods and then investigates the general categories of optimisation techniques capable of addressing the cost-effectiveness problem related to the allocation of SILs.

## 3.7 Optimisation Methods

In the field of optimisation there are two categories of methods for finding optimal solutions: a) exact methods and b) approximate methods. The complexity of a problem and certain requirements from the practitioner often determine which of the methods is suitable. Exact methods explore exhaustively the solution space and guarantee optimality, whereas approximate methods often result in high-quality results within reasonable timeframe. Most of the real-life optimisation problems in sciences, engineering and business typically involve time constraints and high complexity problems; hence, exact methods are not always desirable. The primary strategy for solving such problems relies on the employment of approximate methods. The latter can be further categorised into heuristic algorithms and approximation algorithms. Heuristic algorithms are further classified into specific-heuristics and metaheuristics. Both specific-heuristics and approximation algorithms are typically problem dependent; hence, both require careful construction to solve a specific problem at a time. Further, approximation algorithms have the benefit of producing solutions with provable guarantees and are

excellent for understanding the root difficulties of a problem, which later allows to design better strategies for heuristics. On the other hand, metaheuristics are associated with good quality solutions in acceptable time and are suitable for a wide array of applications. Fig. 3.11 delineates the basic categories and subcategories related to optimisation methods, adapted from (Talbi, 2009:18).

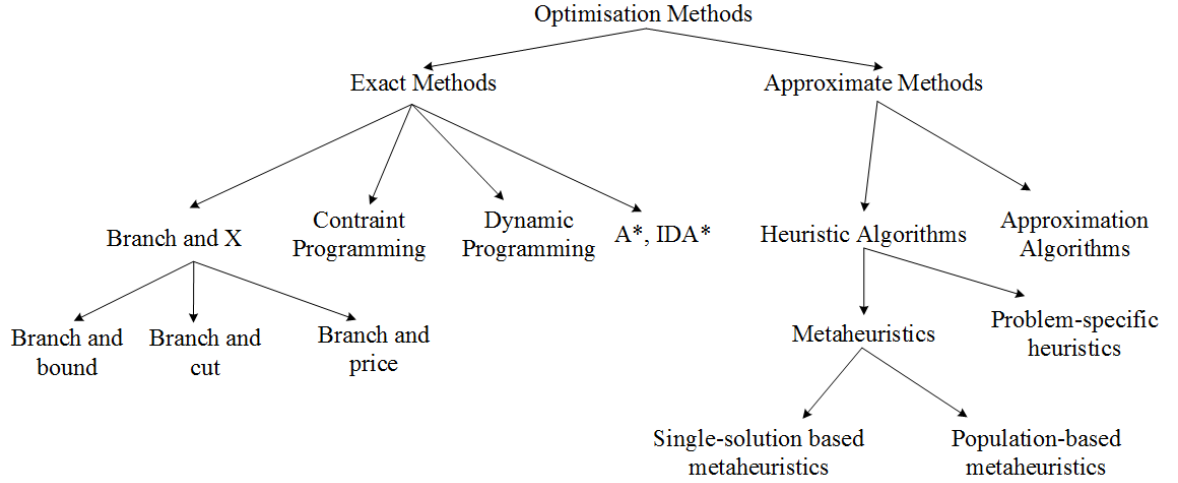


Figure 3.11: Optimisation methods (Talbi, 2009:18)

Optimisation problems are very common nowadays and the more complex a problem is, the higher the demand for optimisation methods becomes. Safety-critical systems are no exception; as explained earlier, research on optimisation methods attempts to solve problems related to SILs allocation and system design (optimality in redundant configurations or alternative components). Parts of the method, proposed in this thesis, utilise optimisation techniques so that the automatically generated argument structures are not only standard-compliant, but also desirable for practical use (i.e. reasonable development costs or optimal architectural redundancy). Thus, the next few sections review the families of optimisation methods used for tackling such a problem but focus more on the technique utilised within the proposed method through an extension of HiP-HOPS.

### 3.7.1 Metaheuristics

Metaheuristics is a family of optimisation algorithms that solve relatively hard problems by efficiently exploring the solution search space. In the past 30 years, scientists have carefully tailored a series of metaheuristics algorithms to solve problems in different fields. For instance, in computer science

metaheuristics, and specifically Evolutionary and genetic algorithms, are employed for problems in artificial intelligence or data mining in bioinformatics (Bäck et al., 1996). There are two major processes when building a metaheuristic search algorithm, the diversification and the intensification of the search space. The former focuses on searching unexplored regions to ensure that the overall space is evenly explored, whereas the latter investigates further already explored space regions, where ‘good’ solutions have been previously found. There are many more classification criteria for the various techniques proposed by researchers. There are algorithms inspired by natural processes, such as evolutionary algorithms or others derived from the behaviour of swarm-based animal communities, such as the ant colony optimisation. Another classification criterion of metaheuristics comes from their deterministic or stochastic nature. Deterministic algorithms (e.g. Tabu search) always lead towards the same final solution given an identical initial solution (i.e. certificate to the answer). On the other hand, stochastic metaheuristics (e.g. evolutionary algorithms) deploy random rules during the search process and often produce distinctive final solutions even when the initial solution given as input is the same.

The key attribute that made metaheuristic techniques successful and popularised them in the scientific community is their exceptional ability of dealing with large-scale problems within short periods of time. Nevertheless, this type of optimisation does not always lead to optimal solutions across the entirety of the solution space and based on the requirements or constraints of the application, it may not be appropriate. Further, there are other factors the practitioner needs to consider before opting for metaheuristics, over exact methods, when tackling a particular optimisation problem. First, it is important to know the size of the input, as even in NP-hard cases the problem might be solvable using an exact approach if the number of instances is small (e.g. 60-client vehicle routing). The second parameter is the structure of the instance, which might be more suitable for an exact method. Finally, any sort of time constraints requires careful consideration, as search time between optimisation methods varies greatly. In other words, best practice recommends to first examine the nature of the problem and its complexity, then consider any constraints and finally choose the suitable method.

Thus, for a given problem, an efficient and exact algorithm with no constraints may be the best choice compared to heuristic approaches. However, for the purposes of SIL decomposition, the HiP-HOPS analysis tool, through one of its extensions, utilises Tabu Search to tackle the problem of optimal SILs allocation. This is a subcategory of metaheuristics and since the supporting tool that implements the proposed method uses that extension to decompose SILs, the next section focuses only on Tabu Search despite the fact that other optimisation techniques could be good candidates too.

#### *3.7.1.1 Tabu Search*

The Tabu search falls under the category of metaheuristic methods and was firstly introduced in (Glover, 1984). It utilises the concept of local search during which a local solution is picked and then its neighbours are examined with the aim to find an improved version. However, Tabu search differentiates from traditional local search algorithms by allowing the selection of non-improving candidate solutions. In addition, the method maintains a data structure (list) of recently visited solutions which the algorithm avoids temporarily and therefore are considered a ‘taboo’. At each iteration, the algorithm looks through all the solutions of the neighbourhood, that are not Tabu, and selects the best one even if it is inferior to the current selected solution. This mechanism allows the algorithm to escape local optima and the Tabu list prevents any immediate reversals to previous solutions. The memory structure and the storing process might be demanding in both space and time but ultimately prevents the method from repeating cycles. Tabu search has been successfully applied to many optimisation problems, although it requires careful design around the problem it aims to solve. The HiP-HOPS tool has been extended to solve the optimal allocation of SILs, for the aviation and automotive domains, with Tabu search. Since the proposed method utilises these extensions, the last section of this chapter discusses some of the details.

#### *3.7.2 Exact Methods*

Exact methods are a class of algorithms that carry out exhaustive search throughout a solution space. The problem is often divided into smaller parts and exact algorithms are applied on the segmented space. The most notable subcategories are those of dynamic and constrained programming. Dynamic

programming focuses on a strategy that recursively divides the problem until it is simple enough to solve within a reasonable timeframe (Bellman, 1957). Constraint programming models the problem as a set of finite-value variables and constraints. The constraints are mathematical expressions that link the variables together (Brown and Miguel, 2006). Another subcategory of exact methods, the Linear Programming (LP), could be an excellent candidate for the optimal allocation of SILs. LP algorithms will always find the optimal solution, assuming one exists, but the search of space is typically exhaustive and might not terminate in a justifiable amount of time. Thus, it may not be as desirable to use, compared to heuristic approaches, when assessing large systems in an evolutionary development cycle.

### 3.8 Automatic Allocation of DALs: An Extension to HiP-HOPS

Earlier sections highlighted the contribution of safety integrity requirements in the justification of safety in argument structures and their impact on development cost. Standards provide guidance for SILs allocation; however, another significant factor when deciding the appropriate SIL configuration involves implementation costs of the constituent components. The HiP-HOPS tool employs a Tabu search algorithm to cost-optimally allocate ASILs and DALs to lower-level elements in a given system architecture. This section briefly describes the core elements and algorithm of the extension for DAL allocation, described in (Sorokos et al., 2015, 2016). The counterpart for ASILs, introduced in (Azevedo et al., 2013), follows a similar approach and therefore is not described here; despite its utilisation within the method for assessing systems in the automotive industry.

The allocation of integrity requirements in lower levels of the architecture is influenced by the contribution of sub-elements to a system failure. The standards do not mandate a particular assessment method; however, they suggest various methods that are suitable (e.g. FTA or Markov Analysis). HiP-HOPS already incorporates suitable safety analysis methods that can be applied iteratively on a system model. One of the products from these analyses are the minimal cut sets, or functional failure sets (FFS) as referenced in ARP4754-A, which are exploited for the allocation process. In civil aviation, aircraft hazards exhibit high severity, and the standard rules allow only a

DAL reduction of down to two levels. This enables an intermediate reduction step, when certain conditions are met, which disqualifies inefficient options and effectively decreases the size of the search space; hence, facilitating the optimisation method applied later. The **first condition** involves a non-linear increasing development cost for an architectural element as the DAL of its functional failures increases (i.e. becomes more stringent). The **second condition** is met when a set of system FFSs of size  $n$ , once arranged in a descending order, demonstrate the following pattern:

- There is always one common functional failure found in both  $FFS_i$  and  $FFS_{i+1}$  for all  $i$  until its value reaches the size of  $n$  ( $i$  represents the index of the list of FFSs).
- $|FFS_{i+1}| - |FFS_{i+2}| \leq 1$  and  $|FFS_{i+2}| - |FFS_{i+1}| \leq 1$ , which means that the difference in the number of members between sequential FFSs does not exceed the maximum of one.

The **third and final condition** is met when there is one FFS in the ordered list that satisfies the second condition and has only one FF member (Sorokos et al., 2016).

The implementation of Tabu search, within HiP-HOPS tool, for allocation of requirements follows the basic scheme found in the literature. Candidate solutions represent the various allocations of DALs over all the functional failures. The best candidate is considered the allocation whose total cost is the lowest. Initially the algorithm creates a candidate solution at random, which in the next iteration is used to produce the next generation of candidates. At each cycle, the generated solutions are compared and the best one is selected, which in this context is the candidate with the lowest DAL cost. At this stage, the algorithm scans the short-term memory (i.e. Tabu Tenure) for any occurrences of the selected solution. If the solution is found in the Tenure, it gets discarded and the next suitable candidate (i.e. second best and so on) becomes selected. The process carries on until the selected candidate cannot be found in the Tenure. There is also an ‘aspiration criterion’ that allows a candidate to be selected even when it exists in the Tenure; only if its cost is the lowest compared to other solutions in current memory. Naturally, during the generation stage, the produced candidates must not violate the DAL decomposition rules. This ensures that any allocations assigned in the reduction-

step cannot receive a lower DAL value than the assigned one, but only a higher value. The core algorithm presented below was originally introduced in (Sorokos et al., 2015).

1. Generation of random allocation
2. Set the generated allocation as ‘current choice’ and add it to the memory (Tenure)
3. Repeat until iteration count or time limit reached
  - i. Produce alternative allocations randomly from the ‘current choice’
  - ii. Re-order the generated allocations, based on DAL cost, in an ascending order
  - iii. Pick the allocation with the lowest cost as the ‘next choice’
  - iv. Repeat until a ‘next choice’ is selected or until there are no other alternative allocations to examine
    - a. If the next choice is not present in the memory, select it as the ‘next choice’
    - b. If it exists in the memory structure, but fulfils the aspiration criterion, select it as the ‘next choice’
    - c. Else, examine the next produced choice
4. If the generated allocations are neither Tabu nor aspiring, set the one with the lowest cost as the next choice
5. The next choice becomes the current selection
6. Add the current selection to the Tenure
7. Re-order the allocations in the Tenure, based on DAL cost, in an ascending order

The different options per FFS are stored in different ‘allocation packs’ within the code. During the random generation in the first step, a random option is selected from each of the allocation packs and then it is combined with any allocations occurred from the reduction-step. At the identification stage for Tabu candidates, the algorithm searches the tenure only for identical allocations. Finally, the second step is a separate process during which partial allocations are added in two separate lists, based on whether they are optional or non-optional. The reason behind the two lists is to find all the partial allocations included in the current allocation due to options taken from allocation packs. There are

two separate loop statements which handle the respective lists. In both loops, a random allocation pack, that affects the current partial allocation, is selected and the algorithm picks one of its partial allocations at random. If the selected partial allocation for the non-optional list assigns a higher DAL than the current one, it is added to a ‘resulting list’. For the optional list, if the selected partial allocation somehow alters the current allocation, the algorithm uses it to generate a new allocation, which is later added to the ‘resulting list’. The latter represents the alternative allocations in the above algorithm.

As explained earlier, the Tabu search does not guarantee optimal solutions. The use of the Tenure (memory structure) might be helpful in avoiding recently visited solutions; however, it sometimes keeps revolving around the same area in the solution space. This results in solutions that might not be global optima. To mitigate this effect, the extension of HiP-HOPS has a user-defined parameter in place that prevents the algorithm from selecting the same candidate repeatedly. The algorithm stores the exact composition of the current candidate. Once the same candidate has been selected consecutively more times than the value of parameter, the algorithm is forced to restart (i.e. step 1). This time, during the generation of the random candidate, the constituent elements must be different than the ones in the stored composition. In the worst case, this method has no effect in the whole process; however, in the best case, the search method shifts to a different region in the solution space with potentially better candidate solutions.

### 3.9 Summary

This chapter outlined the safety lifecycle as advocated by modern safety standards, where commonalities and differences were highlighted. It then explained the core elements of the certification process and investigated available guidelines (i.e. functional standards and regulations) related to compliance in the aviation and automotive industries. This research pointed at the commonalities in the development and safety assessment processes under these safety standards, such as the ISO 26262. In particular:



- a) SILs as abstract requirements are categorised and allocated differently in each domain, however they are used to represent the stringency of development and assessment activities and therefore are employed in safety arguments independently of the system element type (e.g. function or component).
- b) Due to recent accidents, standards suggest that a system should be analysed and argued for safety during its development and even post-production. This means that the safety case should be maintained regularly and follow the iterative nature of development. The use of the ‘V-model’ was found to be common among the standards investigated.

The above provide a way, through the SILs, to indicate system elements that need further attention in terms of safety (e.g. elements with higher SILs are more likely to introduce errors and incur a hazard) and they also describe a regime for assessing a system and allocating these requirements properly. In a way, the common elements found in the standards provide an implicit strategy of evaluating and arguing safety. Thus, the initial hypothesis that from these common elements a pattern of safety argumentation can be elicited, is proven. The chapter also presented the SILs decomposition as an optimisation problem and reviewed some of the available optimisation methods for tackling it. Finally, it demonstrated the main algorithm used by the proposed method through the HiP-HOPS tool.

## Chapter 4 Connecting Safety Cases to System Models

---

Chapter 4 presents the concept of model-connected safety cases; a method for the semi-automatic construction and maintenance of product-based safety argument structures. In general, the focus of this chapter is on the method outline, the description of its individual activities and the methodological contribution. First, before delving into the specifics of the method, some of the issues with current practices in safety case production and maintenance are revisited. This helps clarify the intention of the proposed method and emphasise the contribution of this thesis. Further, the chapter continues with an overview of the method and inspects the different phases in detail. Then, with the help of a simple example, the reader is guided through the proposed method and its core features. Finally, a review of earlier work on the automatic synthesis of safety arguments is provided and potential relevance with the method presented in this thesis is discussed. Note that any technical details and the metamodel, along with its constituent elements, that underpin the operationalisation of the approach and its supporting software tool are provided in Chapter 5.

### 4.1 Complications in Maintaining Safety Cases

In Section 1.5, major challenges associated with the construction and maintenance of safety cases were presented. Generally, one of the issues of document-based safety cases is their size and complexity. As system architectures grow large and more complex, the volume of the safety case documentation becomes greater and the argument can be hard to follow and understand. Further, the volume is not always a sign of effectiveness; instead, a clear, concise and well-structured argument can be easier to navigate and more sufficient in communicating system safety. To this end, graphical notations have drastically improved the structure and clarity of traditional safety cases. With the use of notations, the arguments are represented diagrammatically, which leads to an intuitive navigation through the argument structure and facilitates the tracing of reasoning. The latter is helpful both for the safety case developers and the regulatory bodies for improving and evaluating safety claims, respectively. Another issue with text-based safety cases is readability, which is diminished by inadequate organisation and frequent use of cross-referencing. Note that this affects mainly the

presentation of evidence artefacts and their connection with the argument. Graphical notations make the connection of claims with the supporting evidence more clear and further support reasoning with the use of the assumption, context and justification elements throughout the argument. Moreover, the utilisation of tool support can further reinforce readability by rejecting any cluttering information from the core argument structure and by providing it with the use of hyperlinks instead. This allows developers to redirect the reviewer to the original evidence artefacts or to a detailed document that explains thoroughly any assumptions or justifications made during development.

Despite the progress made from the introduction of graphical notations, the synthesis of safety arguments remains, for the most part, a manual process and safety cases are not connected to the system model which they represent. As a result, the safety engineer is required to gather information about the system scope, the architectural design and its failure behaviour before they initiate the production of the safety case. Then, after performing a series of activities (compliant with the corresponding safety standard), recording any assumptions and contextual information, and establishing a suitable argument strategy, the construction of the safety case begins. All these actions require significant time and effort, which grows proportionally with system complexity. Moreover, earlier chapters discuss that modern standards advocate that a safety case should be maintained as a ‘living documentation’ throughout a system’s life span. In other words, a system should have a safety case that reflects its current state (with regards to safety) during the stages of development, manufacturing and operation. Changes to system architecture are not always easy to evaluate and therefore in most cases the engineer needs to repeat all of the above-described tasks before updating the safety case. Considering the iterative design approaches and agile development for software components, it is logical to assume that the system is subjected to multiple changes over time. Based on this knowledge, production and maintenance of safety cases become cumbersome, due to the need for reassessing system safety, as well as validating arguments. Note that even when the system architecture is mature design-wise, assumptions and decisions in earlier stages of development might change later, during testing or even in the operation stage, based on feedback. This could potentially

lead to new safety goals and consequently new requirements, which the system needs to satisfy; thus, resulting in a safety case update.

The aforementioned challenges are arguably more prevalent with text-based safety cases. For example, continuous changes to a large portion of a text document of considerable size can be impractical in modern development. Unfortunately, even with the concise form of arguments that utilise graphical notations, some of the challenges persist due to the nature of manual approaches in the construction and maintenance of argument structures. The repetitive actions of gathering information, safety assessment activities and other necessary tasks may eventually slow down the maintenance of safety cases and lead to an asynchronous feedback towards the system development. The latter is likely to mislead engineers and introduce errors. On one hand, the issue could partially be addressed by employing more staff to perform the necessary tasks. This would lessen the workload on an individual level, but it would necessitate better management to ensure that all teams are informed about the current system state and come to a consensus on major decisions. Further, the efficiency is not guaranteed due to the increased economic costs. Sooner or later, the amount of resources that manual approaches require may potentially discourage the maintenance of safety cases. On the other hand, the application of iterative procedures suggests automation as a potential solution. Earlier research on automated system analysis methods and tool support also indicate that it is a viable solution. Through automation and tool support, the workload on individuals can be lessened, errors introduced by the human factor can be reduced and any safety associated tasks can be managed more efficiently by reducing reliance on document exchange.

## 4.2 From Safety Standards to a Pattern for Safety Assurance

Earlier, in the introduction chapter, it was stated that there are common elements in the guidelines found in modern safety standards. Chapter 3 investigated functional standards for safety-critical systems and reviewed in detail the safety lifecycle prescribed in the aerospace and automotive industries. Based on the findings, it was concluded that there is solid ground for developing a common

pattern for safety assurance. Fig. 4.1 below depicts the transition from standards to the method and tool implementation.

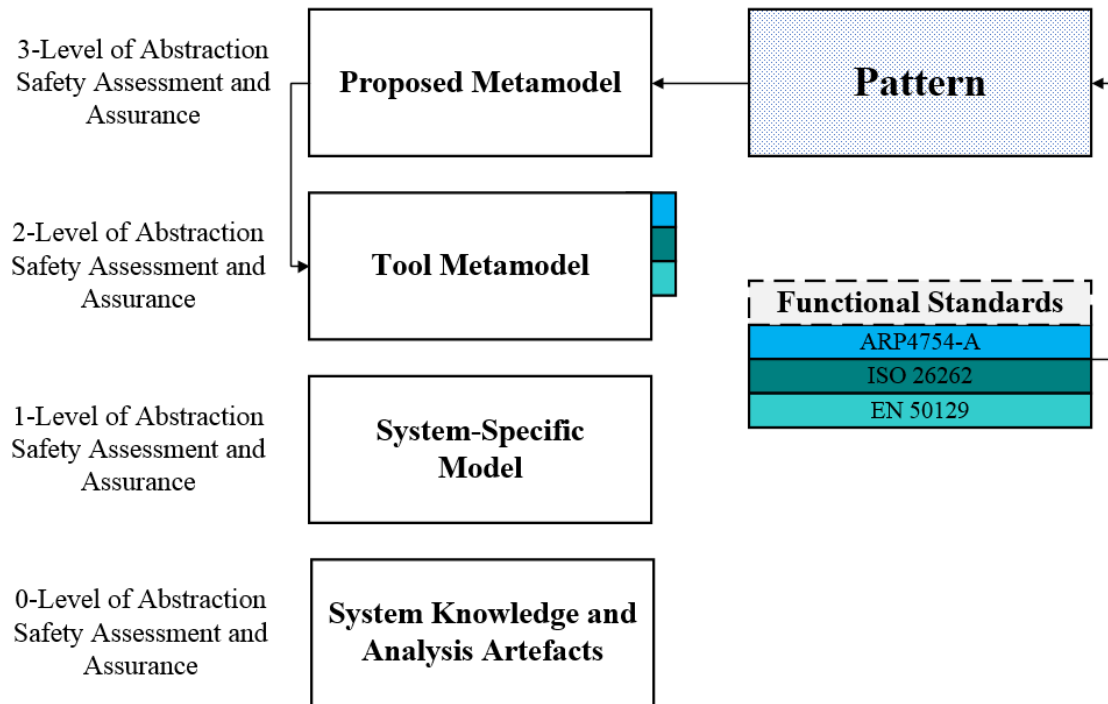


Figure 4.1: Transition from standards to the pattern and metamodels

Starting from the guidelines, the details of processes and analyses advocated were subtracted and a pattern was formed. The latter was operationalised through a metamodel, which was designed to describe the interconnections between safety assessment activities (found in all standards mentioned previously), the model-based paradigm, an argument structure notation (e.g. GSN) and argument patterns. To enable tool support, the metamodel is then further refined into the tool metamodel (lower level of abstraction) and implemented as a software framework. Apart from code-specific details, the tool metamodel can accommodate standard-specific operations and processes (colour-indicated in Fig. 4.1). For example, the tool changes the rules for decomposition of requirements based on the system in question. This way the tool maintains the method's approach for reasoning whilst remaining compatible with standards for certification purposes. The tool metamodel effectively describes the

system model and how information is captured. Having this knowledge, the user can create a model for a project-specific system, which contains contextual information, evidence artefacts and metadata. The latter is displayed as the ‘System Knowledge and Analysis Artefacts’ in Fig. 4.1. As an example, it may represent raw data from the analyses or stakeholder planning/requirements. These can be incorporated within the system model and automatically presented in the final argument or added manually by the user. Each of the elements from the ‘proposed metamodel’ to the ‘system knowledge and analysis artefacts’ are annotated with a level value, which symbolises the level of abstraction; starting from the proposed metamodel as the highest level (3) to the lowest level (0). The pattern is described in the next section, whereas technical details on the metamodel and the software framework are discussed in Chapter 5.

### 4.3 The Pattern and Automation Infrastructure

Chapter 3 presented the safety lifecycle for systems development under the guidelines of ARP4754-A and ISO 26262 and how safety cases can adapt in order to be maintained throughout the system’s life span. Safety is viewed as a system property that is controlled from the early stages of development. During functional design, a process of hazard analysis is recommended to establish safety requirements by evaluating system behaviour in different phases and conditions (e.g. environmental). Once a preliminary system architecture is designed, reliability engineers can determine the appropriate integrity requirements, which fulfil the higher-level safety requirements, and assign them to lower-level system elements. The allocation should respect architectural dependencies and failure propagation. This process is iterative, meaning that as the system architecture is refined and new subsystems and components are introduced, the analysis and allocation steps are performed again. Finally, the system is argued to be adequately safe because all hazards identified in the systematic risk analysis are shown to be addressed through meeting the appropriately allocated subsystem and component safety integrity requirements (SILs). Consequently, the final structure of a safety case, under the proposed method, follows this logical scheme. This common pattern is illustrated in Fig. 4.2 below.

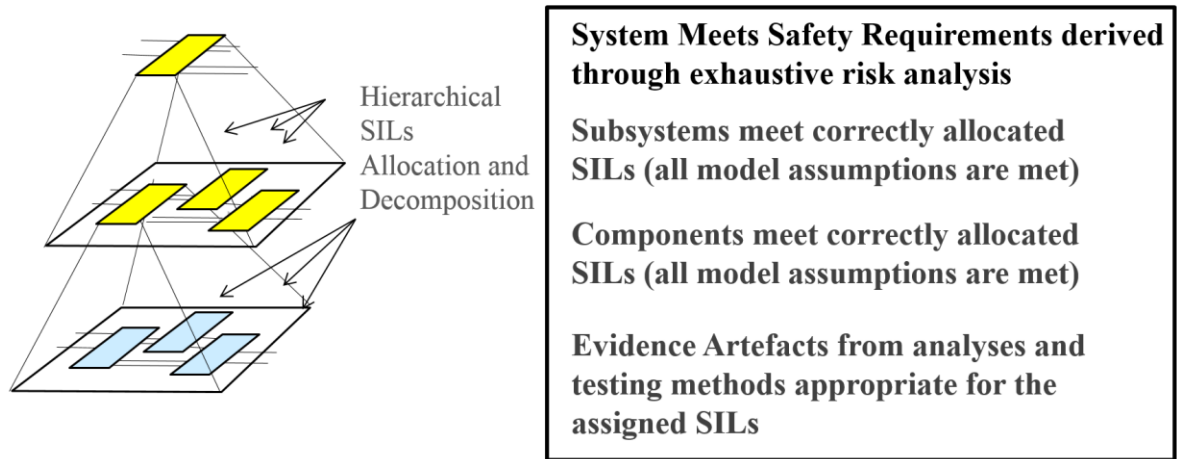


Figure 4.2: Representation of the pattern

Now that the conceptual basis has been formed, it is essential to review another important part of our method, the automation. Earlier, automation was proposed as a suitable candidate solution for challenges in regard to safety case construction and maintenance. To achieve this, it is important that the framework for assessment and assurance requires as little input from the practitioner as possible. Despite the method following a model-based approach, which certainly sets a good foundation towards automation, the means necessary to further aid this objective must be examined.

First, to support a safety argument, it is necessary to provide the appropriate evidence, which usually derives from safety analyses. For example, FTA can assist in the validation and verification of requirements. Chapter 2 reviewed various tools that have achieved automated safety analysis; thus, by exploiting such tools the automation capabilities of our method could be expanded. Following the guidelines and based on the pattern described earlier, to allocate SILs across the system architecture, the user has to follow domain-specific rules. In (Sorokos et al., 2015) and (Azevedo et al., 2014), it was shown that DALs and ASILs can be allocated automatically and near-optimally with regards to development cost. Hence, another significant part of safety assessment can be automated. Finally, another important aspect of the safety case is the synthesis of a clear and convincing safety argument. Towards this end, research has focused on argument patterns in an attempt to promote best practice and help engineers construct arguments. This is achieved by disassociating the details of an argument from the context of a particular system. We have already reviewed that the GSN has extended

structural abstraction capabilities that effectively support the concept of patterns. Despite their popularity, current practice still involves the manual instantiation of argument patterns by the developers. By implementing the appropriate data structures in the metamodel to capture the relationships between the system model, evidence artefacts and argument patterns, it is feasible to develop an algorithm to automate the instantiation process. Given sufficient information about the target system, engineers will only have to create the argument pattern manually since the algorithm will automatically instantiate the pattern based on the system model information and produce the argument structure. Apart from further minimising user input, the automatic instantiation can be helpful in reducing human errors.

#### 4.4 Method Overview

This section presents an overview of the method, shown in Fig. 4.3. The typical process flow is indicated with arrows. Steps S1 to S10 are performed in a sequential manner since they require some form of input from its predecessor; only exceptions are steps S4 and S8 which can be executed concurrently. Note that actions in square rectangles are manual steps (mainly modelling and information annotation), whereas rounded rectangles indicate automated procedures.



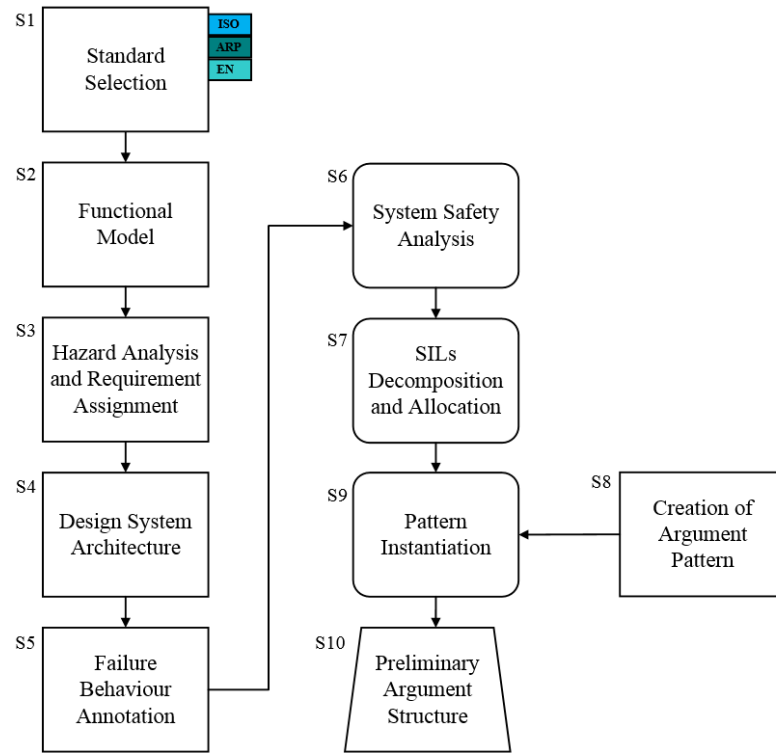


Figure 4.3: Method Overview

At the beginning, developers select the appropriate standard for the system in question, which initialises and prepares the tool for standard-specific details. Then, the process requires the users to identify system functions and design a functional model in the modelling editor of the tool. After that, they can perform hazard analysis and risk assessment activities and annotate the function elements with the hazards and corresponding variables via a set of customised dialogues. For instance, with the ISO 26262 version of the tool metamodel, the user can add the controllability parameter as a means to justify the decision-making with regards to the requirements allocated for each function. Afterwards, practitioners may proceed with the development of the system architecture, capable of supporting the pre-defined high-level functions, and the annotation of subsystems and components locally with failure logic (i.e. failure behaviour information). The model is then processed by the HiP-HOPS engine, which performs FTA and automatically produces the minimal cut sets. Given this information and the appropriate input (cost coefficient for each DAL), the tool utilises the specialised HiP-HOPS extensions (either for DALs or ASILs) to cost-optimally allocate the requirements across the architecture. With the information about the system and its context, the user defines a suitable

safety argument pattern. Once complete, the instantiation algorithm processes the pattern and architecture as well as other information (e.g. individual evidence artefacts or comments) from the data structures and generates the safety argument.

## 4.5 Method Demonstration

This section demonstrates the method with the use of an abstract example that follows the details provided by the ARP4754-A standard. The example features a simple braking system.

### 4.5.1 Functional Design

Firstly, it is important to determine the core functionality of the system and its purpose. The process typically involves the engineers designing aircraft functions, which are later decomposed into multiple subfunctions. For the civil aircraft, examples of high-level functions can be the ‘Navigation’ or ‘Braking’. For instance, ‘Navigation’ can be segmented into a set of subfunctions, such as the ‘Automatic Direction Finder’ (ADF), ‘Inertial Navigation’ or ‘Radar Navigation’. Specifically, for civil aviation, engineers also need to identify the different phases of aircraft operation and categorise the functions based on the corresponding phase. Examples of such phases are the ‘Taxiing’, ‘Take Off’, ‘Flight’ or ‘Landing’. Note that this stage is significant, as the ARP safety standard perceives the high-level functions to be at the very top of the hierarchy with regards to the system architecture. During the functional design stage, generic requirements, with respect to either nominal or failure behaviour, as well as operational constraints are also established for the already defined functions. For example, the weight of the aircraft could potentially impact the requirements for deceleration performance of the braking system.

Although software tools that support graphical modelling exist, it was decided to create a custom-made modelling editor and integrate it within the tool. Using our editor, the practitioner is able not only to create the functional and system architecture models, but also to annotate the standard-specific information as well as other contextual data (usable later for the argument structure). Further, our implementation stores data in way that is directly compatible with the HiP-HOPS metamodel and its

input constraints, which allows us to avoid unnecessary model transformations. Additionally, the editor provides a mechanism that is responsible for model consistency checking during the system modelling step (e.g. high-level functions cannot be connected directly with low-level components). Once the user has adequately created a functional model, they can choose to load this structure on the main data structure, the ‘MCSU’ (Model Connecting Storing Unit). The latter enables model tethering, which is useful for the automatic pattern instantiation. Each function element creates a new ‘MCSUElement’ (i.e. core storage entity), which takes the type ‘Function’ and captures all the associated data. More details on the MCSU are provided in Chapter 5.

For the purpose of the example, let us assume a generic function that represents the ‘Braking’ function of an aircraft.

- Function Name: Braking
- Function Description: High-Level Function
- Phases: Take-off (for directional control), Landing (deceleration)

#### 4.5.2 Hazard Analysis and Requirement Allocation

This step involves the annotation of the determined functions with information produced by the FHA as seen in Section 2.2.3. Note that this process is typically conducted manually. However, there can be cases where the examined system model and context can be reused from an existing system. Therefore, saving these in separate files and reloading when needed could be a useful feature in the software tool. In any case, the information produced by performing FHA is annotated in the model. The ARP4761 documentation provides a general guide of what information should be registered from the FHA within the data structures; however, the means for the identification of these features is up to the developer’s preference. For our example, FHA information can be shown in Table 4.1 below.

Table 4.1 Hazard Analysis Information

<b>FHA Name</b>	Function FHA	<b>Aircraft Phase</b>	Landing
<b>Function Name</b>	Braking	<b>Hazard Classification</b>	Catastrophic
<b>Hazard Name</b>	H1	<b>DAL</b>	A
<b>Hazard Description</b>	Loss of braking	<b>System Requirements</b>	Likelihood of occurrence 10E-9 per hours of operation
<b>Hazard Effect</b>	Potential crash or off-course landing	<b>Verification Method</b>	Quantitative and Qualitative analysis

To simplify the example as much as possible, the braking function is restricted only to one hazard (H1) and one phase (i.e. landing). The consequences could still be catastrophic if the aircraft fails to decelerate as expected; therefore, the function was assigned DAL A. In practice, the initial DAL could be lower due to other mitigation measures in the overall aircraft design; nevertheless, the requirement selected bears no significance for the purpose of this example. As for generic system requirements, we have arbitrarily provided an occurrence probability, which means that on average the failure is not expected to occur more than once per 1 billion hours of operation. Additionally, the hazard is easily understood and therefore no supporting material was necessary. Verification methods, although not specific here, should include results from both quantitative and qualitative analyses due to the hazard classification, as prescribed by the standard in (SAE, 1996:23).

#### 4.5.3 Designing the System Architecture

Once the FHA is completed, the engineers typically proceed with the establishment of a preliminary system architecture that supports the top-level functions. Fig. 4.4 depicts the control flow of a simple brake system architecture. In Appendix I, it is also shown through an image how this architecture is modelled in the developed tool of the method.

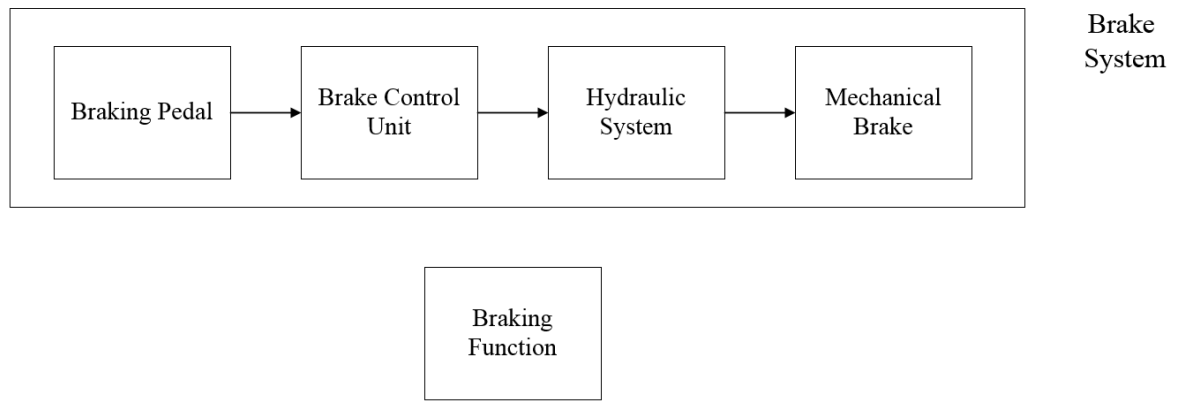
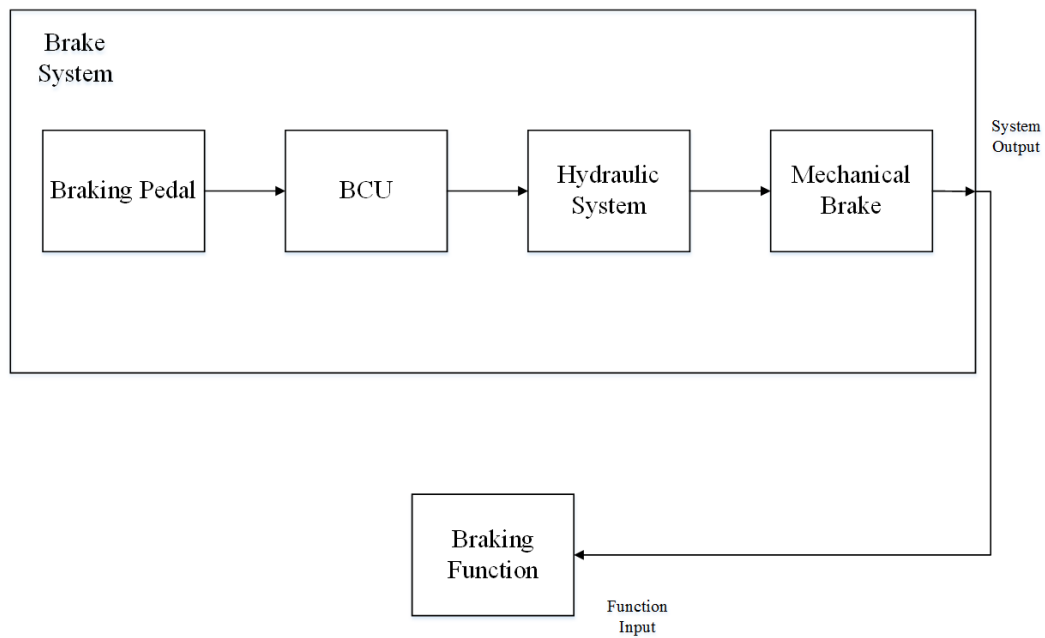


Figure 4.4: The control flow of the example system

The system presented here corresponds to an abstract model for a disc brake. The aircraft size, mass and average landing speed provided in the specification can heavily influence the design and complexity of the disc brake system. In this example, a single disc brake, mostly suitable for small and lightweight aircraft, is demonstrated. A system for a disc brake typically consists of several mechanical parts such as the brake disc, the brake calliper and the brake pad, a hydraulic system and for the purposes of this example a Brake Control Unit (BCU). Once the braking pedal is pushed down, the BCU interprets the signal and propagates a command to the hydraulic system. The command is normally interpreted by the actuator and this drives the regulation of hydraulic fluid released. Based on the hydraulic pressure created, the brake calliper pushes with proportionate force the brake pad on the brake disc. The continuous rubbing between the pad and the disc generates friction, which converts the kinetic energy of the wheel into heat (Aeronautics Guide, 2020).

#### 4.5.4 Failure Behaviour Annotation

The next phase is the Failure Annotation, where the user proceeds with the annotation of local failure behaviour on the elements that constitute the system. Fig. 4.5 shows the failure propagation of the brake system.



*Figure 4.5: Failure Propagation of brake system*

Initially we examine the failure modes for the target system. The hazard (H1) identified for the braking function is caused by an omission of output from our brake system. To proceed and further identify how the function can fail, we effectively have to identify what are the combinations of system elements that lead to a system omission of output. Due to the simplicity of the example and based on the description of the component interaction, it was simple to find the root causes. In fact, by just viewing Fig. 4.5, it is visible that failures propagate through the system as a chain (sequentially). Each of the components fails either from a failure of its predecessor or from an internal failure. Specifically, the Braking function fails, when the Brake System has an omission of output (so, the function suffers from O.Input). The latter is effectively the output of the Mechanical Brake component, which may fail either by an internal failure or when there is no pressure from the hydraulic component. The Hydraulic System can fail to operate either due to some sort of physical deficiency or due to an omission of input from the BCU. Clearly, the same happens for the BCU, which may omit to send an output command if any internal failure occurs or when there is an omission of input from the brake pedals. Brake pedals can fail to provide output only from internal failure. In practice, there might be other reasons, such as the operator failing to give the command. This, as well as other implications

are not covered directly by our method even though they could be added as supporting evidence in a safety argument; hence, these have been deliberately discarded to preserve simplicity in the example.

#### 4.5.5 System Safety Analysis

At this stage, given the failure behaviour and system architecture, it is possible to perform safety analysis to identify the minimal combination of component failures that may cause the hazard. The failure logic, annotated as logical expressions, is illustrated as a fault tree for simplicity in Fig. 4.6.

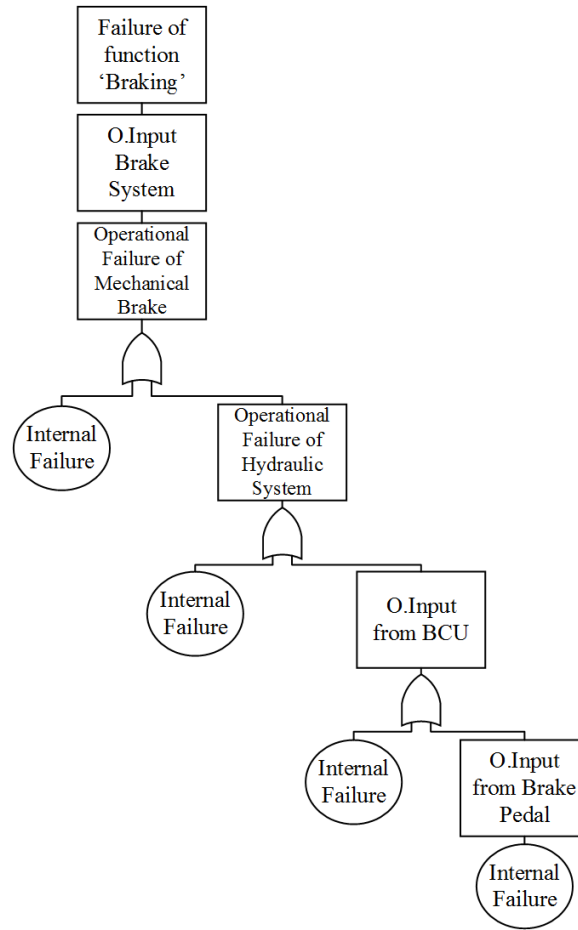
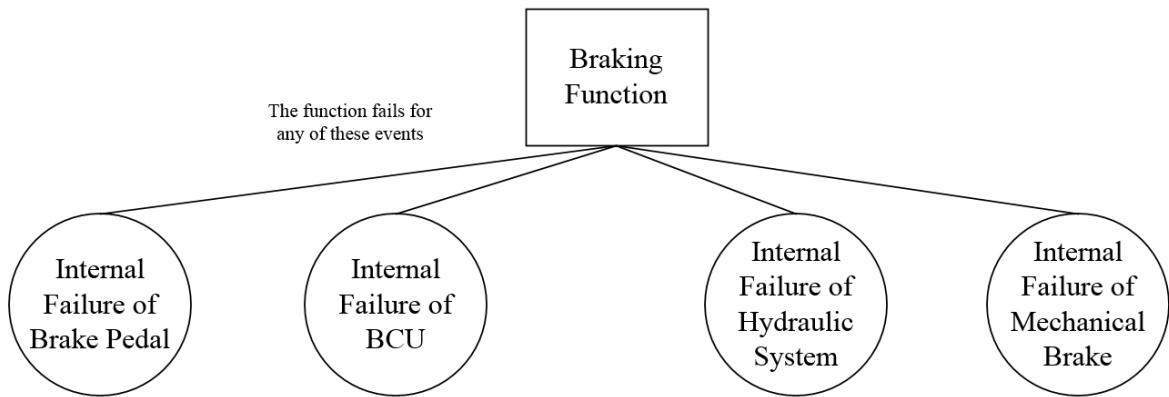


Figure 4.6: Failure behaviour of the target system via a fault tree structure

To do so, the tool utilises the HiP-HOPS engine for automating the analysis. HiP-HOPS creates fault trees for each component in the architecture and then combines all the individual fault trees (synthesis stage) into the overall system fault tree, with the hazard as the top node. Additionally, the HiP-HOPS engine also produces the corresponding minimal cut sets. As discussed earlier in Section 2.3.1, MCSs

are valuable for identifying which low-level failures are necessary and ample to cause system-level or function failures. In our case, the MCSs are depicted in Fig. 4.7, as a diagram.



*Figure 4.7: Minimal cut sets produced for the example*

As shown in the figure above, the Braking function fails if any of the basic events of the corresponding system elements is triggered (i.e. internal failures).

#### 4.5.6 SILs Decomposition and Allocation

At this phase, the method proceeds with the automatic allocation of DALs. Since the top-level function has already been assigned DAL A from the FHA, the algorithm, discussed in Section 3.8, optimally allocates DALs throughout the architecture. From the information extracted from the MCSs, it is concluded that every element in the model might cause the system-level failure and consequently trigger the function failure. As a result, the example satisfies the third condition of the reduction-step algorithm; therefore, all of the components are assigned DAL A without any further consideration. Due to the simplicity of the example, the DAL-cost registration step, useful for determining a cost-optimal allocation, was skipped intentionally.

#### 4.5.7 Creation of Argument Pattern

As mentioned earlier, the argument structure is driven by a user-defined pattern. The practitioner can create an argument pattern either via the graphical user interface or via the embedded text editor, in an XML format. The tool also allows to import previously made patterns in XML format. Fig. 4.8 presents the pattern created (with the GUI) for this example. The pattern elements outside the dotted



rectangle represent elements that are shown only once in the final argument, whereas elements within the rectangle are shown repeatedly, as the pattern iterates over the given function's ('x' in the figure) members, their subsequent members and so on. Each system element that has no further constituent subsystems (effectively being a component) provides evidence for the satisfaction of requirements only for its own. For subsystems with members, the instantiation algorithm traverses over their members and ultimately the process is performed for all the members in the architecture.

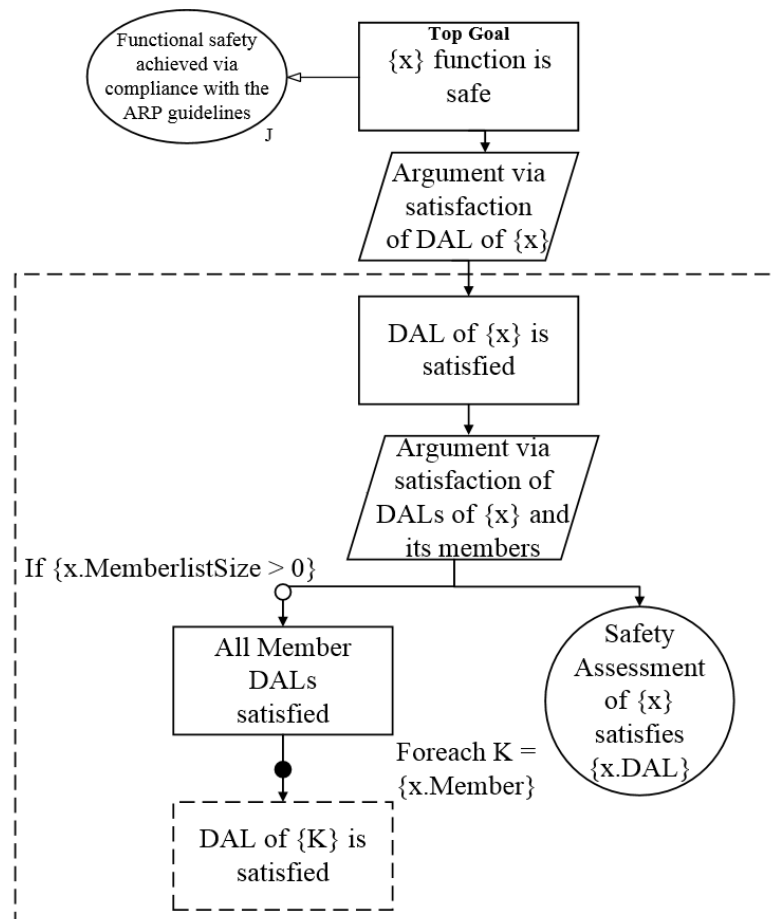


Figure 4.8: Pattern for the example

#### 4.5.8 Argument Structure Generation

Once the pattern is complete, the user activates the instantiation algorithm, which in turn looks within the pattern for instantiable system elements (i.e. functions, systems and subsystems) and replaces them with concrete argument elements. The process terminates when either: a) there are no further abstract elements or expressions to be replaced, or b) when the elements left are low-level items (that

cannot be further refined). At this point, the argument structure is generated and can be viewed, and potentially altered manually, in the argument editor of the software tool.

The size of the generated argument is comparable with that of the pattern due to the simplistic nature of this example. Further, the resulting structure is not complicated; hence, for these reasons the argument could easily be constructed manually. However, the pattern shines when the system architecture grows larger and more complex because the argument structure in that case would grow significantly.

By following the method presented in this thesis, the time required for the construction of an argument (assuming system modelling and assessment have been completed) is limited to the time needed for the pattern synthesis. Additionally, as demonstrated in the case study (Chapter 6), architectural changes are reflected in the argument simply by repeating the annotation process since the rest of the procedures are automated. However, any changes in the argumentation strategy would typically require the user to adjust the existing pattern where appropriate. The argument structure produced for the example examined in this section is shown in Fig. 4.9. There is also an screenshot of this argument structure as it was generated by the tool in Appendix I.

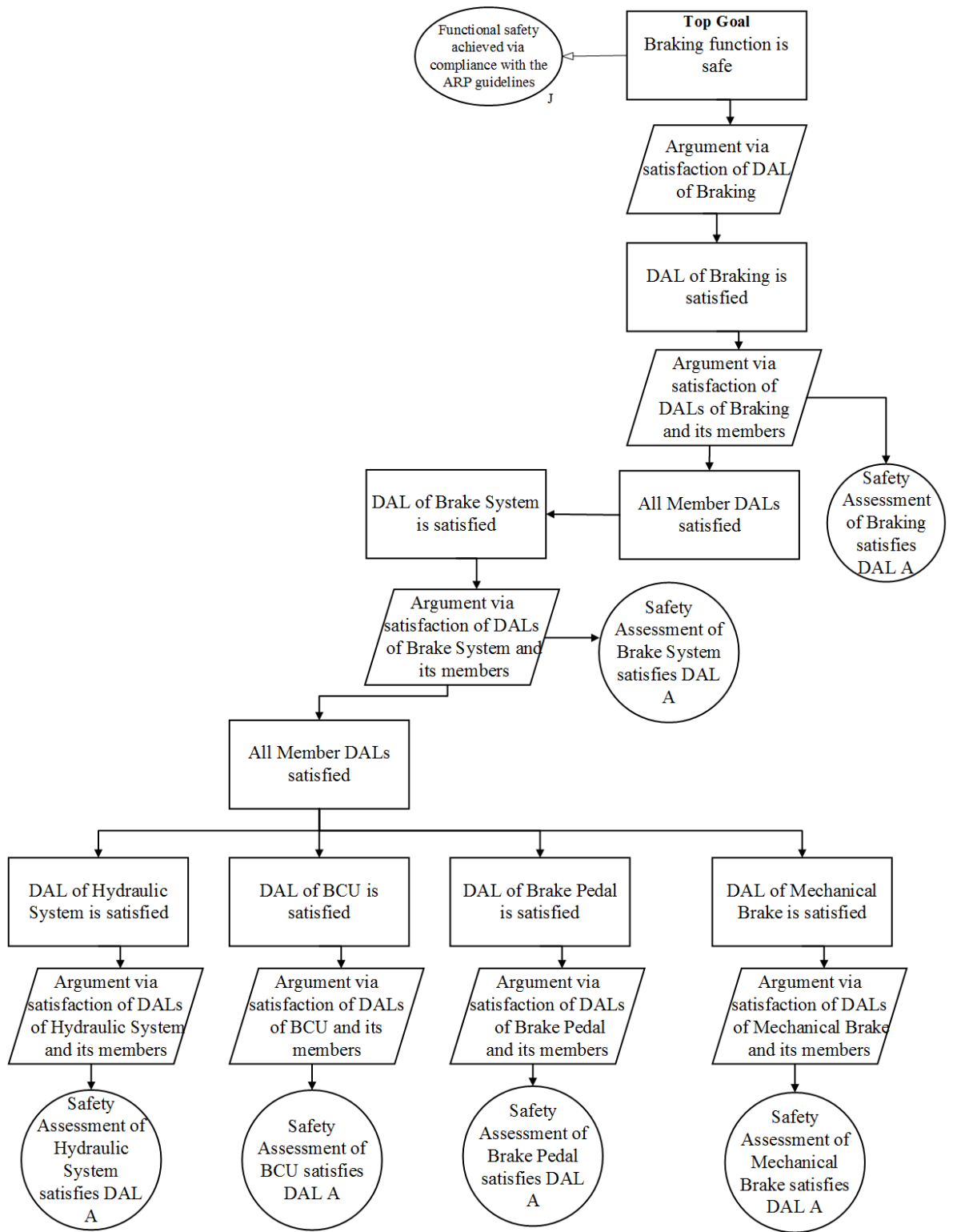


Figure 4.9: Produced argument from the abstract example

## 4.6 Comparison with Relevant Work

This section presents earlier work that focuses on the construction of model-based assurance and safety cases. The following subsections briefly describe the metamodels, methods or tools available to the best of my knowledge and discuss major differences to the method presented in this thesis.

### 4.6.1 SACM

Already introduced in Section 2.13.1, Structured Assurance Case Metamodel (SACM) is a high-level metamodel that can be used to construct structured arguments for model-based system assurance, as well as security and safety cases for various domains. Further, the SACM, with its dedicated elements for assurance case management, promotes modularity. This is important for splitting an assurance case diagram into smaller, more manageable, diagrams based on the developers' needs (e.g. security-related arguments can be separated from the rest of the assurance case). Other significant features revolve around the expressiveness of SACM, such as the controlled vocabulary (through the Terminology component) and the multi-language capabilities (through the Base classes). The former is useful for model-based assurance arguments since it allows the users to refer to various system elements via custom-made keywords. The multi-language feature is helpful for writing arguments in different natural languages, as well as in formal languages; thus, creating some foundation for automated reasoning. For these as well as other features, the SACM has become a powerful metamodel for argumentation.

Naturally, these capabilities are the product of continuous review and improvement over the last few years. Initially, the SACM suffered from integration issues between its constituent adopted metamodels (i.e. SAEM and ARM) and domain-specific content (e.g. prescribed descriptions of evidence artefacts were discarded and replaced with a standardised exchange model format), as well as inconsistencies between various element classes. Most of these issues were resolved in the specification of SACM version 2 (OMG, 2018). Unfortunately, even in that specification there was no provision of a graphical notation for SACM elements, which made the modelling process difficult and prone to errors. Despite the GSN's popularity for system assurance, the specification of SACM

was lacking a concrete mapping between GSN and SACM elements. Hence, whether using legacy or GSN-based argument structures, there was no support for translating them into the SACM model. Finally, there was also no detailed guidance on the intended usage of the SACM and considering the high complexity of the metamodel, there was no great incentive to adopt it. However, a great explanation of the SACM and a process for GSN to SACM model transformation were presented in (Wei et al., 2019). The latter enables a structured argument to be constructed with the well-known GSN elements and then translate it into an SACM model. There has also been progress in (OMG, 2019:49) with regards to the graphical notation for argument and evidence elements. The transformations and the concrete graphical notation for SACM have been implemented within the Assurance Case Modelling Environment (ACME). This tool was developed within the Eclipse Modelling Framework (EMF) by exploiting its infrastructure for graphical editors (Eclipse, 2020) and the incorporation of the GSN, CAE and SACM metamodels. The main objective of this tool is to enable the practitioners to transition from GSN (and CAE) to SACM models, and ultimately to provide the foundation for an integrated environment under the SACM (Wei et al., 2019:38).

Despite its valuable capabilities, SACM still poses some limitations for effective use within the proposed method in this thesis. First, by the time most of the issues, mentioned above, regarding the SACM were resolved, the metamodel for the materialisation of model-connected safety cases was already designed and partially implemented. Further, even with the SACM version 2.1, there are no references of tool support for embedding system modelling capabilities within the SACM. On that note, official contributors to the SACM specification have collaborated with academics and companies around Europe and together have initiated the DEIS project (DEIS, 2020). The main scope of that project was to tackle the challenges in dependability assurance of Cyber-Physical-Systems (CPS) and succeeded with the proposal and development of an innovative concept, the Digital Dependability Identities (DDIs). The SACM was used as the backbone for the metamodel implemented for assuring dependability of CPS (Wei et al., 2019). During this project various development engineering methods and tools have been presented and basically resolved the SACM's

aforementioned weakness of lacking tool support. When the project started, however, this thesis was already under development and shifting to another metamodel would be impractical.

In SACM, argument patterns can be represented with the use of `+isAbstract` feature, which makes certain elements act as templates (e.g. claims or `ArgumentReasoning`), and the `AssurancePackage` that contains the pattern. However, even though the metamodel is capable of being instantiated from abstract patterns and system information, the appropriate algorithms and tool support are missing at the time of writing this thesis. Arguably, the SACM element `ImplementationConstraint` can be used to specify instantiation rules for patterns and there is ongoing research on the matter in (Wei et al., 2019:30); nevertheless, a model management engine and further tool support is still required to automate the instantiation process. Although this was also true for the proposed metamodel, it was more practical, due to the thesis's time constraints, to develop the instantiation infrastructure for a simpler metamodel that is still in line with the research objectives rather than attempting this for the higher-complexity SACM.

Finally, the SACM is a domain-neutral metamodel and even though it supports evidence artefacts to be represented with the `ArtifactPackage`, it does not have any mechanisms for performing system analysis techniques, such as FTA. As a result, the practitioner needs to implement a framework or employ other third-party tools for system analysis. The results should then be translated from the third-party tool's format to the SACM metamodel. Instead, the metamodel created for this thesis is supported directly by the HiP-HOPS engine, and therefore analysis techniques did not require implementation. All that being said, the value of SACM should not be understated as it is a sophisticated metamodel for structured arguments that receives continuous development. Therefore, it is an excellent metamodel either as standalone or as an exemplar for engineers to build and improve their own metamodels for model-based assurance.

#### 4.6.2 D-Case Editor

The D-Case editor is an open-source assurance case editor developed as part of the Dependable Embedded Operating System (DEOS) project. There has also been a web-based version, the D-Case

Communicator (Matsuno, 2017), which allows multiple users to concurrently develop GSN-based assurance structures. The intent behind the D-Case editor was to create a tool that facilitates the development of dependability cases with the use of reusable patterns (Matsuno et al., 2010). It is implemented using the Eclipse graphical framework (GMF) for modelling and the GSN elements for representing structured arguments. The tool allows experts to create, store and reuse dependability case patterns via a built-in library. Since dependability criteria (e.g. SILs for safety-critical systems) and attributes (i.e. combination of safety, security, integrity, reliability, availability and maintainability) change based on the domain, the library was initially designed to include patterns for some general attributes and patterns for the project's specific cases. Nevertheless, the structure of the library is extendable and can be customised based on the domain's attributes and criteria. The D-Case editor has been enhanced with various features and utility mechanisms. For instance, with an advanced proof assistant, the tool is able to perform consistency checking on a graph structure. This prevents the user from creating cyclical arguments, as well as connecting nodes unintentionally (i.e. evidence directly below a strategy). Other notable functions are: a) the graphical comparison between two different assurance cases and b) the support of goal nodes from external sources with the use of URLs. Finally, the author in (Matsuno, 2014) has presented a framework that formalises GSN and its extensions as an assurance case language, whilst providing an instantiation algorithm for GSN patterns. These patterns are represented using the GSN context elements, since it is possible to have multiple per argument module, and parameterised expressions. The instantiation algorithm goes through all GSN elements, and when it identifies a connected pattern element, it proceeds with the instantiation based on the expressions and parameters. The pattern is created manually; the user selects an appropriate pattern (basic argument and pattern elements) from the library and then sets the correct values for all the parameters (e.g. names and loop indexes). From this point on, the instantiation algorithm synthesises the argument based on the user-defined pattern.

Arguably, D-Case editor shares some common features with the supporting tool developed for the proposed method; however, it has a different purpose and implementation. D-Case editor enables the

creation of argument structures both manually and semi-automatically, but it does not connect the pattern directly with the system architecture and the corresponding information. Instead, the user has to create the pattern with both the target argument structure and the system architecture in mind. Further, D-Case editor and the assurance language provided are domain-neutral, so they are not directly related with safety cases. Hence, there are no capabilities for safety analysis, suitable for supporting argument structures for safety-critical systems, nor other activities required by modern safety standards (e.g. FHA). Finally, even though it provides a systematic way for assurance case production, the user is still required to manually update the pattern elements when system changes happen, which leaves the user with the responsibility to perform that synchronization.

#### 4.6.3 ACedit

ACedit is an open-source tool that supports the graphical construction of assurance cases in both the GSN and the OMG's, older metamodel, ARM. It was developed at the University of York and implemented under the Eclipse framework as a set of plug-ins. The tool features a graphical user interface (GUI) with two different editors, for GSN and ARM metamodels respectively, whilst offering model management functionality through the Epsilon toolset (Despotou et al., 2011). Design-wise, the ACedit is implemented in a number of layers: a) the bottom layer, which consists of the basic framework (i.e. core Eclipse) and b) the second layer that entails all of the Eclipse plug-ins responsible for the modelling, the graphical user interface and metamodel specification (i.e. EMF, GEF and Epsilon). There is also a third layer that consists of plug-ins responsible for the editor functionality, which is based on the corresponding metamodel information (i.e. GSN and ARM), as well as the model management functions. The latter enables one of the key functions of ACedit, which is the ability to transform an argument structure from one metamodel to the other (i.e. ARM to GSN and GSN to ARM). ACedit also provides model validation for evaluating user-defined models via a set of GSN constraints. Overall, it is a solid framework for the construction of assurance or safety case argument structures and its capabilities far exceed those found in other simple graphical editors (e.g. MS Visio). Unfortunately, ACedit focuses solely on manual construction of arguments, and it



seems that there is no further support to create system architectures and link them to structured arguments. Finally, as opposed to the method presented in this thesis, there is no development for generating evidence artefacts; neither is it explicitly specified whether interoperability with other tools (for safety analysis) is possible.

#### 4.6.4 ACME

The Assurance Case Modelling Environment (ACME) is a software tool that allows the creation of assurance cases by utilising the SACM metamodel for the modularisation, evidence and terminology features and the GSN metamodel for the argument representation and reasoning (Wei et al., 2019:40). ACME is implemented under the Eclipse's Graphical Modelling Framework (GMF) and features different views to provide the ability to create elements from either GSN or SACM. For example, the assurance case view allows the user to create GSN modules and contracts, as well as elements from the SACM's AssuranceCase component (e.g. ArtifactPackages, TerminologyPackages and constituent elements). The module view allows the user to develop arguments with standard GSN elements. Each of the goals developed can also be an away goal and refer to an assurance case structure from a different module within the assurance case view. The key characteristic of ACME is that it allows the argumentation to be modelled with the use of GSN elements, with which the practitioners are likely to be familiarised with, whilst providing the advanced features of the SACM (e.g. controlled vocabulary). On top of that, it further provides a transformation mechanism from GSN to SACM. Due to the versatility of the Epsilon platform (Epsilon, 2020), ACME can be extended to support even customised GSN models and enable their conversion to the SACM. Despite ACME's solid foundation for assurance cases and its capabilities for model transformation, it is still in a prototypical stage and does not directly support system analysis techniques or automated instantiation of argument patterns (i.e. assurance templates).

#### 4.6.5 Weaving Model

The weaving model was firstly introduced in (Hawkins et al., 2015) as a model-based approach for assurance case development. The method proposed in that publication manages to link structured

arguments (presented in GSN) with architectural and process system models. The approach describes a model, referred to as the ‘weaving model’, whose primary goal is to establish interconnections between different models and store them with appropriate semantics. The software that supports the method was developed within the Eclipse framework and utilised the Epsilon toolset.

Model weaving is not a new concept as there are other methods and tools, such as (del Fabro et al., 2005), that utilise it. Specifically, in (Hawkins et al., 2015), the weaving model is used to encapsulate the dependencies between the GSN patterns and individual reference information metamodels. Note that model weaving can be performed manually by the user (via element linking) or automatically via a model transformation functionality. In brief, the method involves the following phases:

1. The user creates the appropriate information models within the tool (e.g. system architecture).
2. The user then builds a suitable GSN pattern by utilising the GSN metamodel and its extensions (i.e. structural abstractions and parameterised variables). Elements that can be instantiated are referred to as ‘roles’; they are abstract entities that can be later replaced with one or more concrete elements (based on system information) by the instantiation algorithm. The variables (in curly brackets) are typically replaced with the information from the respective system element.
3. The user establishes the dependencies between the various models; captured within the weaving model.
4. The user initiates the instantiation program, which:
  - a. Looks for instantiable elements within the GSN pattern.
  - b. Identifies all the information required from the information models and requests that from the weaving model.
  - c. Receives the data and creates the final instantiated argument structure with all the appropriate information.
5. The program renders the structure on screen.

Model-based arguments are useful for development since they allow automation and validation. However, the arguments are as good as the information and reasoning provided by the practitioner. The main differences between the approach in (Hawkins et al., 2015) and the method proposed in this thesis are:

- The reasoning behind the claims, which in our method is established around the requirements and the safety lifecycle prescribed by functional standards (e.g. ARP4754-A and ISO 26262).
- The ability to semi-automatically generate the essential evidence artefacts from the system architecture, responsible for supporting the arguments.

#### 4.6.6 Reusable Safety Case Fragments from Compositional Safety Analysis

A method that generates reusable safety case argument-fragments, referred to as “FLAR2SAF”, is presented in (Sljivo et al., 2016). The increased interest for reuse in the domain of safety-critical systems is discussed by the authors. They also argue that reuse of components is really meaningful only when evidence artefacts are reusable so that integration is achieved safely. However, safety is viewed as a system property and it is highly dependent on the context. The authors in (Sljivo et al., 2014) introduced the concept of ‘safety contracts’, which encapsulate safety behaviour of components in assumption/guarantee pairs. They have also provided a method to construct argument fragments from those contracts and ultimately support reuse of components; more specifically, commercial off-the-shelf software (COTS). The generation of these contracts is achieved from the model of a given COTS component following failure analysis via the Fault Propagation and Transformation Calculus (FPTC). The method combines the typical V-model, found in various safety standards, with a component-based development. Fig. 4.10 depicts this model as adapted from (Sljivo et al., 2016:4).

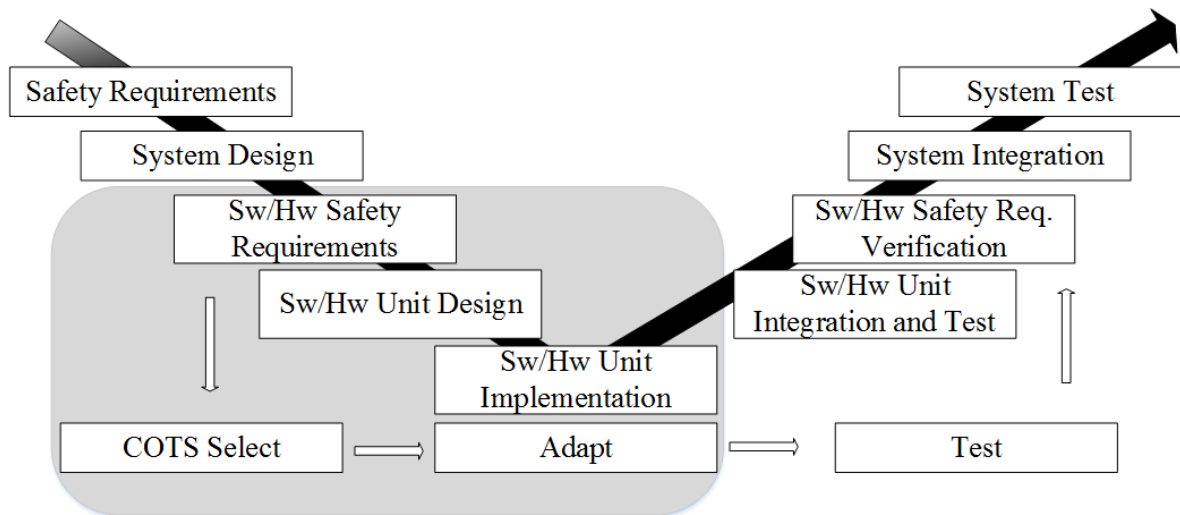


Figure 4.10: V-model combined with COTS-driven development (Sljivo et al., 2016:4)

The approach initiates in a top-down manner with a hazard identification by analysing the failure propagation on the given system architecture. If any system-level failure leads to intolerable hazards, it prompts the method to proceed with requirements formulation and decomposition to lower levels in the architecture. These requirements are represented as SILs, which act as a measure of quantifying risk reduction. Once all the requirements are decomposed, appropriately developed COTS can be selected if they meet those requirements. The method also allows for an adaptation stage in case those components do not fully satisfy the requirements. Additionally, the authors in (Sljivo et al., 2014) have also introduced a component metamodel that connects safety contracts with the appropriate evidence. This enables the reuse of evidence when the contracts are reusable on a given scenario.

The method described in this thesis portrays similarities to the method above; however, there are substantial differences. The proposed approach is applicable from the early stages of system design and requires less rigorous annotation on the system model compared to a formal method for software components. Moreover, it mainly focuses on the assessment and assurance of system safety instead of promoting the reusability of evidence artefacts or components. Finally, in this thesis the utilisation of SILs has a vital role in the argumentation strategy, whereas the method above uses the concept mostly as a quantifying factor to select the appropriate COTS.

#### 4.6.7 Automated Generation of Safety Cases for Modular Software Product Lines

The method presented in (Oliveira et al., 2015) enables the reuse of design elements for Safety Product Lines (SPLs) by automatically constructing modular safety arguments. To do so, the authors integrate compositional safety analysis, allocation of safety integrity requirements, assurance case techniques and variability management into SPL engineering processes. SPL is a development paradigm for software products, which share common characteristics and fulfil a similar purpose within a domain; they are typically developed from a set of basic assets in a prescribed way (Clements and Northrop, 2001). Successful use of SPL products requires variability management (i.e. presenting software assets), which refers to: a) how variability is modelled and b) what are the most suitable features for different system configurations. Naturally, SPLs for safety-critical systems demand even more effort in the sense that artefacts from safety assessment and assurance processes need to be traceable across the different SPL product versions.

The method and tool presented in (Oliveira et al., 2015) utilise the HiP-HOPS tool to automatically allocate requirements and in conjunction with a set of customised patterns, they manage to generate safety arguments. Note that to modify the approach for SPL products, it is important to determine the variation of features and context across the architecture and safety analyses. Hence, Oliveira et al. integrate the following information models and the argument notation metamodel: 1) the feature model, 2) the system model, 3) the graphical notation metamodel (D-Case GSN) and 4) the component failure model. The interconnection between these metamodels is achieved via a ‘model weaving’ approach. This method manages to semi-automatically produce evidence artefacts and argument structures, but it is heavily focused on the specialised infrastructure needed for the SPL products. On the contrary, the method presented in this thesis applies to a more general array of systems that follow the guidelines of functional safety standards.

#### 4.6.8 Generation of Model-Based Safety Arguments from Automatically Allocated SILs

A method for automatically constructing safety arguments for the civil aircraft industry is presented in (Sorokos, 2017). That approach follows the safety lifecycle prescribed in the ARP4754-A safety

standard and provides a means for semi-automatically generating argument structures from safety requirements. To do so, they utilise the MATLAB Simulink and HiP-HOPS tool for system modelling and safety analysis, respectively. After establishing an architecture and evaluating the results from the analyses, it is possible to allocate DALs to high-level system elements. By employing a Tabu search optimisation and certain rules for decomposition, the approach assigns DALs in an (almost) optimal way to the rest of the architecture. Then, with a user-defined argument pattern (appropriate for the system at hand) and an instantiation algorithm, the method produces a preliminary argument structure. The method discussed here shares many similarities with the proposed method in this thesis. Although the method presented in (Sorokos, 2017) is novel, it appears to be in a prototypical stage with no integrated environment for tool support. Further, the method is described in an implicit, ad-hoc manner; whereas, in this thesis an explicit metamodel is presented and materialised in a software tool. Finally, the method presented in this section focuses on the automatic generation of safety arguments from argument patterns for the aerospace industry. On the other hand, the approach of this thesis focuses on capturing converging features, shared within safety standards, and providing a common pattern and framework for producing safety arguments usable in cross-sector applications.

## Chapter 5 Design of Model-Connected Safety Cases

---

This chapter introduces all the structural entities, implementation details and design-focused decisions that led to the materialisation of the proposed method. Initially, the core metamodel and its elements are formulated in a diagrammatic form using the Unified Modelling Language (UML). Then, the chapter analyses the data structure, the Model Connected Storing Unit (MCSU), which was designed to be the foundation for model tethering and information exchange within the proposed method. In a later section, the transition from the proposed pattern metamodel towards the designed and implemented tool metamodel, which is the heart of the software tool, and additional elements are explained. Further, the chapter provides the application of the pattern generation mechanisms and demonstrates the core parts of the developed instantiation algorithm. Finally, the chapter ends with a summary that highlights the tool's importance in our approach.

### 5.1 The Metamodel

Our metamodel integrates elements found in various metamodels in order to incorporate all the capabilities necessary for supporting the method discussed in Chapter 4. The backbone consists of the HiP-HOPS metamodel, which enables the modelling of system architectures, failure behaviour annotation, system analysis techniques and optimal allocation of SILs. Arguably, an alternative or even a custom framework could have been used as a replacement for HiP-HOPS. However, development of a brand-new framework for safety assessment was beyond the scope of this thesis. Hence, one of the reasons behind the investigation on supporting tools for safety assessment that was presented in the Background chapter. The major benefits from integrating HiP-HOPS is twofold. First, HiP-HOPS supports the model-based paradigm and provides many features that are directly usable by the method. Secondly, as already mentioned in the Introduction (i.e. Section 1.7), HiP-HOPS is a mature tool being developed and supported over the past twenty years. This provided confidence that there are no issues related to the correctness of the analysis of a system or the allocation of requirements. However, regardless of the choice, all safety assessment tools have unique formats to input data before ultimately using them. That being said, the implementation of a

completely different metamodel on our software would require additional data transformations to interact with those tools – irrespective of the tool selected. This was the main reason why the proposed metamodel expanded on the HiP-HOPS’s metamodel. The key structural elements of HiP-HOPS that form the foundation of our metamodel are:

- a) **the model**, a top-level element that acts as a container for system architecture and knowledge.
- b) **the systems/subsystems**, which are intermediate placeholders for multiple interconnected components and containers of lines (that capture those interconnections via ports).
- c) **the components**, which are the core elements of the target system. These can be hardware, software, abstract functions or even processes. Note that components can also consist of subsystems (with the use of implementation elements).
- d) **the ports**, which represent component interfaces that enable the propagation of (failure) information towards the rest of the system.
- e) **basic events**, which are the base sources of failure for individual components (failure modes).  
For example, the component representing a valve may have a “stuckClosed” failure mode that can lead to an omission of output. Basic events can effectively represent any kind of failure (e.g. energy, information or logical) and can be accompanied with probabilistic data when necessary.
- f) **deviations**, failures that can propagate throughout the architecture. They capture the failure class (e.g. Omission) and the associated port. Based on these, there may be, for example, an ‘omission of output’ towards another component or a ‘commission of input’ from a linked component (of lower level).

The connection between these elements and potential architecture styles are depicted in Fig. 5.1.



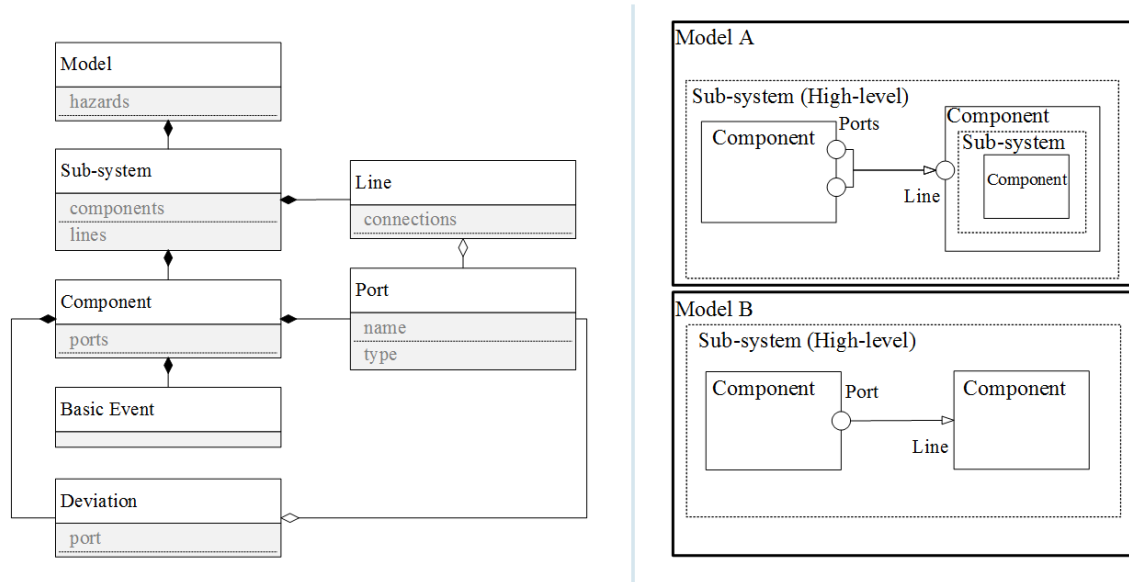


Figure 5.1 (Left) Simplification of core HiP-HOPS metamodel and (right) illustration of system model design strategies

On the left of the image, the aforementioned core elements that constitute part of the HiP-HOPS metamodel can be viewed in UML format. On the right, it is shown how an architecture can be visualised using different design strategies. Specifically, a model can have either a subsystem of multiple components or a more complex architecture of subsystems nested within components. In practice, the HiP-HOPS metamodel is more complicated than what was described so far. For example, it allows the specification of common cause failures and various types of deviations, the creation of different perspectives to distinguish and highlight the software from hardware aspects of a system, as well as the introduction of various component implementations to enable architectural optimisation.

The proposed metamodel has been integrated with elements from the GSN standard (ACWG, 2018) to encompass argument structure rules and semantics, as well as argument pattern metadata and modularity features. The CAE notation (CAE Framework, 2020) is a valid candidate and could have been used equally for the argument structures; however, due to our familiarity with the GSN and its extensions, this notation was used instead.

Finally, the metamodel has also been extended with the MCSU structure to support the linking of the architecture, argument pattern and standard-related processes within the instantiation algorithm.

Thus, allowing the semi-automatic generation and maintenance of safety arguments. Fig. 5.2 illustrates an overview of the metamodel relationships.

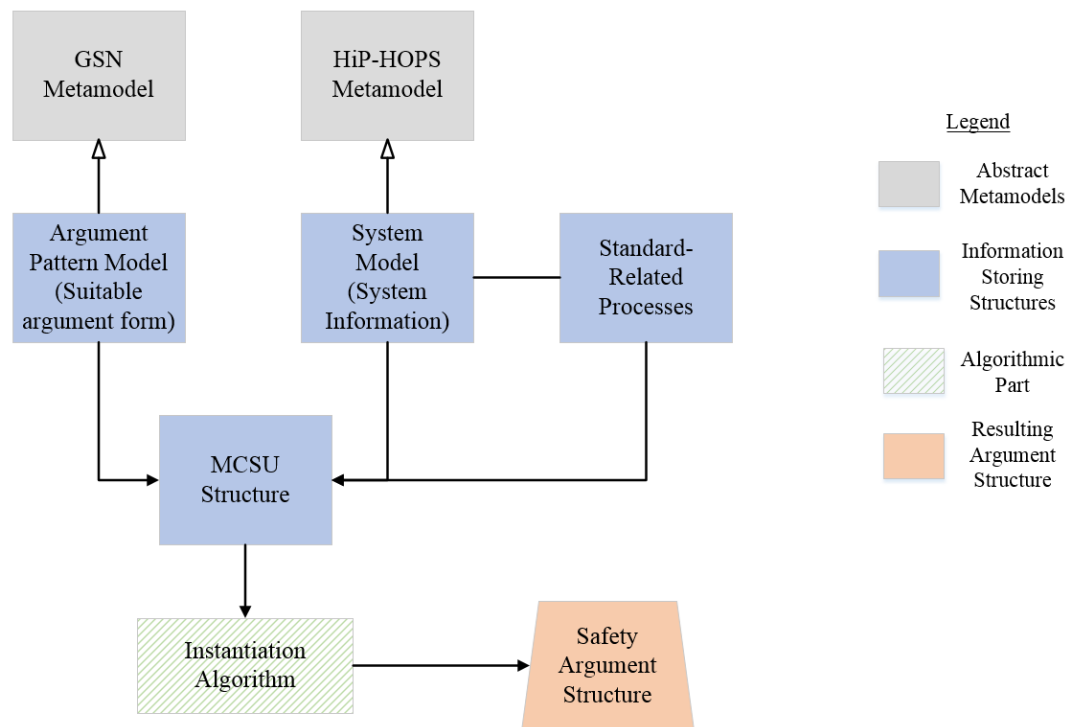


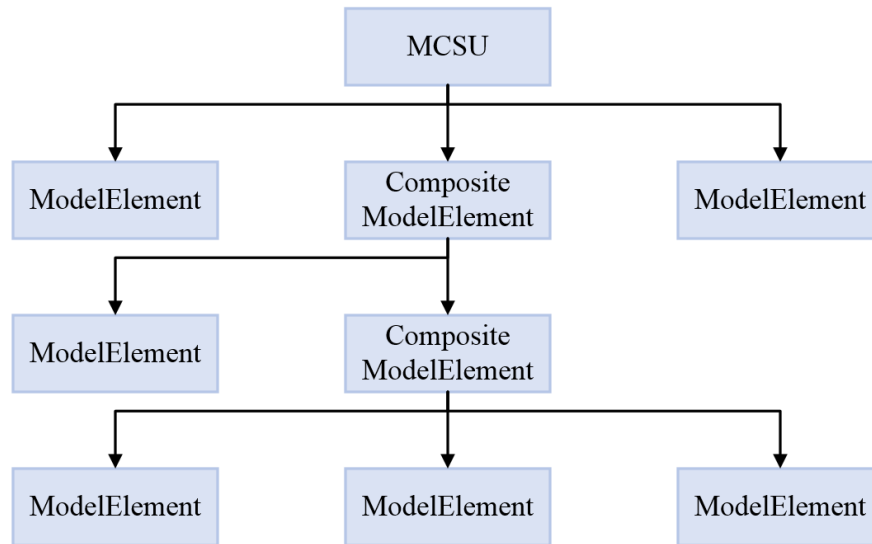
Figure 5.2: Overview of metamodel relationships

As shown in the figure above, standard-related process elements are connected with both the model and the MCSU. Before explaining the reason behind this, it is worth reminding that standard-specific safety assessment processes have an important role in safety assurance as they justify compliance with those standards. That being said, they provide not only a guide for system analysis activities but also help shape the argument pattern and consequently the final argument. Additionally, processes' outcomes can often be used directly as evidence for specific claims. For example, in the aviation domain, the FHA table could be used as evidence to support a claim arguing the correctness of DALs allocated to specific high-level functions. This means that assessment processes are essential for the concrete argument and therefore are embedded in the pattern and later utilised in the instantiation algorithm. Thus, to gain access to those processes, the outcomes are stored within the MCSU structure and the algorithm can retrieve this data with specific calls. The software tool allows the practitioner

to create those results in XML format and import the file or annotate the model accordingly. From the latter, the program can generate equivalent results and transform them to MCSU elements.

## 5.2 The Model Connecting Storage Unit

The instantiation algorithm requires information about the system architecture, assessment processes, system analysis results, allocated SILs and the argument pattern before generating the concrete argument. This information is captured in various models described within the aforementioned metamodels. Even though access to this information was possible through the model storing structures directly, it was more effective to organise the data slightly differently and under the same storage unit. One of the reasons for creating the MCSU can be traced back to the internal structure of the software tool. The metamodel elements (e.g. system architecture as described in HiP-HOPS) and their graphical counterparts along with element-specific operations (e.g. annotation dialogues) or visual behaviour are handled within certain management classes, referred to as ‘editors’. To operate on those metamodel elements, each editor acts as their higher-level container. This results in an additional layer of ownership on top of the HiP-HOPS’s metamodel, which already establishes system relationships in a complex way (i.e. subsystems, lines, components and implementations). Hence, fetching data during the instantiation could lead to performance drawbacks. Therefore, the MCSU was implemented with an adaptation of the Composite pattern (Gamma et al., 1994:183) as a means to alleviate this issue. The purpose of this pattern is to create a composition of elements where any element can include not only other entities, but other elements as well. Fig. 5.3 depicts the general strategy of the composition mechanism. Using this pattern, the system architecture is saved in the MCSU and each of the elements can include not only information, such as failure data, but all of its constituent elements (e.g. subsystem). Similarly, the argument pattern elements can include information, such as the name and description, as well as groups of other elements – assuming a supporting role.



*Figure 5.3 Example of the Composite mechanism*

During the modelling stage, to avoid rebuilding the library after every single change, either in the information or the structure, the program composes the MCSU just before the instantiation takes place. With this approach for information keeping, the program maintains good performance during the instantiation process due to significantly less function calls. Moreover, by preserving the metamodels discussed earlier, model transformations are avoided when using the HiP-HOPS engine. Under the MCSU, the system information, such as failure modes and SILs, and argument pattern parameters, such as element type and member-elements, can be treated more uniformly and accessing information or specific operations are easier to manage with the use of the Visitor behavioural design pattern (Gamma et al., 1994:366). This pattern utilises a double dispatch mechanism, which controls the programming operation based not only on the name of the request and the type of the object (i.e. caller), but also on the type of the ‘visitor’ called. It is useful for abstracting functionality from a group of classes (parent and derived) and allows to perform different operations to the elements of a specific structure via the implementation of different visitor subclasses. As a result, new additions in the structure of the group (in this case the MCSU) or new functionality might incur changes to the existing visitors’ interface and implementation or the creation of new visitors, respectively. For example, a visitor class responsible for structure checking can be implemented to perform model checking on the MCSU. The visitor will treat elements differently based on their type. In our case,

pattern elements would be checked based on the GSN rules (e.g. strategy cannot be connected directly with evidence), whereas system architecture would follow safety standard-specific rules (e.g. based on ARP4754-A, a high-level function cannot be directly supported by low-level items). In the case that a new type of MCSU element was added, then a new visitor subclass should be implemented to manage that type. The design of the core MCSU structure and the visitor class are shown in Fig. 5.4.

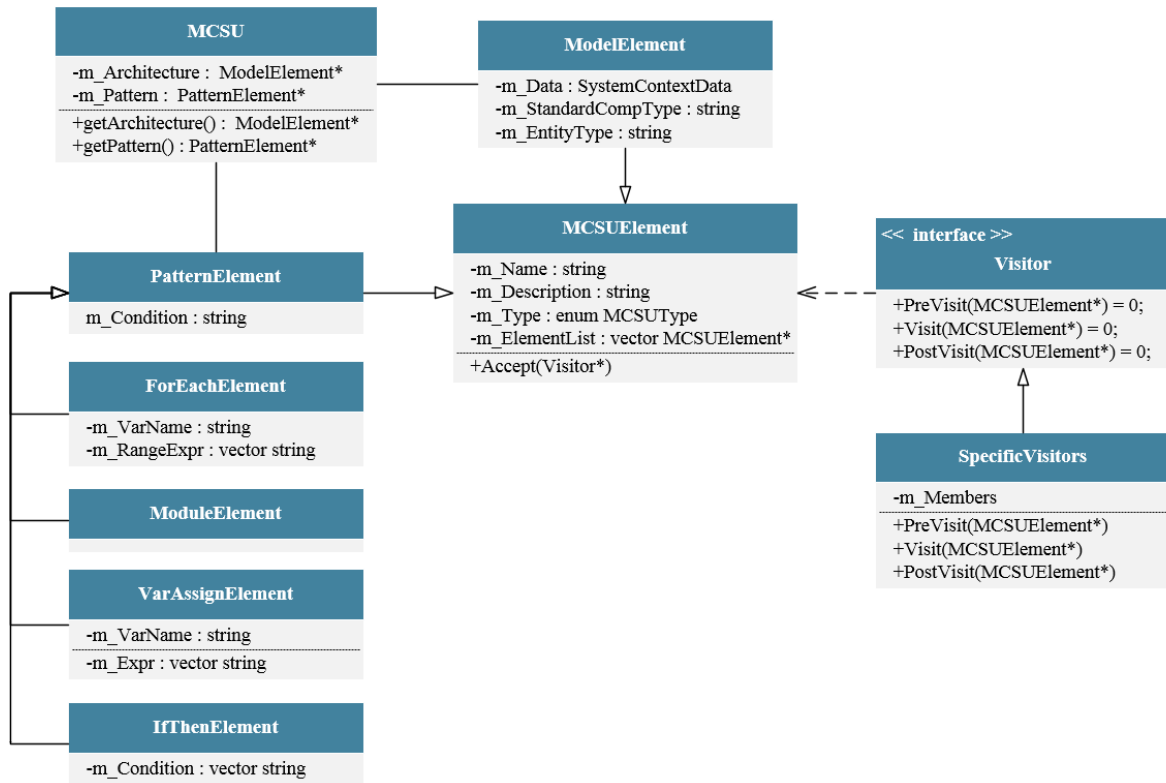


Figure 5.4: Core elements of the MCSU storage structure

- 1) **MCSU**: the metamodel data repository; enables code to incorporate model information.
- 2) **MCSUElement**: the basic unit that enables the composite design and allows building whole-part hierarchies. Basic parameters that are inherited by all the derived classes include:
  - a) 'm\_Name' is the parameter used by the pattern to call the element.
  - b) 'm\_Type' is used for type checking (e.g. ModelElement or TextElement).
  - c) 'm\_Description' is used differently based on the element type.

- d) 'm\_ElementList' is the underlying architecture that resembles relations of 'SupportedBy' and 'InContextOf' for argument elements, or 'PropertyOf' and 'ImplementedBy' for system elements (e.g. functions and subfunctions).
- 2) **ModelElement**: the unit that represents and captures information relative to the system architecture.
    - a) 'm\_EntityType' allows us to differentiate software from hardware components when needed (useful when composing safety cases for the ISO 26262).
    - b) 'm\_StandardCompType' is used for categorising model element types relatively to the target domain (e.g. aircraft functions, subfunctions, system and items for the aviation industry).
  - 3) **PatternElement**: the class for representing argumentation elements such as Goals (claims), Strategies (argument rationale), Solutions (evidence). In this category there are also entities for representing the GSN element and structural abstraction extensions, which facilitate the traversing of the model and guide the argument structure. Finally, the derived class 'ModuleElement' corresponds to the GSN's modular extension and helps create an additional layer of aggregation and essentially implement the modules (subgraphs) in the overall argument.
  - 4) **Visitor**: the interface used by the MCSU elements via the 'Accept()' function. The visitor's fundamental function is the 'Visit()', which is simply declared in the parent interface. Hence, achieving new functionality for the 'MCSUElements' can be accomplished by deriving new visitor classes and implementing the 'Visit()' function for each of the different structure types.

In reality, the MCSU has more elements than the ones shown here. There are elements that capture system analysis or assessment process results; these can further be supported by a 'PractitionerElement', which is another derived class from the 'MCSUElement'. This can be useful as registering and documenting the practitioner responsible for a work product might help build confidence in the results. For example, the analysis or review of results conducted by an experienced engineer are more trustworthy and could be used to generate process-based arguments for the safety case.

## 5.3 Tool Framework

### 5.3.1 The Implementation

Now that the fundamentals of both the method and the metamodel have been established, in this section we review the implementation details. The software tool that materialises the proposed method is developed in the C++ programming language, under the Qt framework (version 5). The language was chosen based on its strengths in performance (due to less memory overhead in the absence of virtual functions), scalability (good at working with small and large sets of data) and excellent community support. Regarding Qt, it is a great framework that provides multiple APIs and libraries for graphics rendering, quick generation of user interfaces, platform-specific dialogues and peripherals management. Moreover, the developers of Qt have engineered a lightweight mechanism similar to ‘callback’ functions, known as the ‘event system’. This allows to conveniently detect any activity (internal or external) and define response functions; this feature supports many prefixed events (e.g. application window resize), as well as customised events and responses. Arguably, using a modelling specialised framework, such as Eclipse, and language, such as Epsilon, could be more appropriate. This would allow for easier metamodel definition, optimised querying, modification and model transformations (Eclipse, 2020). Nevertheless, we opted for a Qt implementation as it provides a wide array of functionality useful for future work, as well as its cross-platform capabilities. Specifically, any programs developed in Qt are almost directly deployable in different hardware architectures and software platforms (Qt, 2020). This can be a strong asset for a safety assurance tool as it can be used as a solution for runtime safety assurance in embedded systems.

### 5.3.2 The Design

The tool architecture combines core classes provided by the Qt framework, the metamodels we utilise in our method and various utility classes. Generally within the Qt framework, one of the most notable classes is the ‘QGraphicsScene’, which acts as a container and management class for all types of 2D graphical items. To fully exploit the capabilities and performance provided by Qt, it was only natural to follow the scheme described in the documentation instead of creating everything from scratch.

Based on this idea of scenes, we have created the ‘Editor’ classes; each one of them is responsible for storing and graphically managing elements from specific metamodels whilst maintaining specialised behaviour for certain input. The basic tool framework is shown in the following figures. Starting from Fig. 5.5, which shows the higher-level (in terms of purpose) classes.

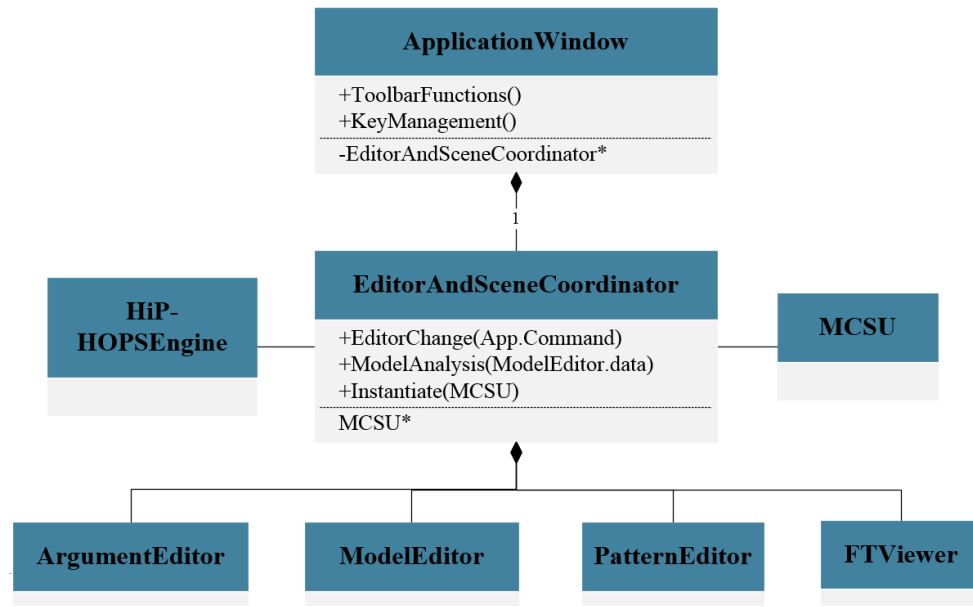


Figure 5.5 Tool high-level classes

The entity ‘ApplicationWindow’ is the class that creates the tool window and handles all the core actions and settings. As the foreground, it is responsible for the appearance of the tool, such as frame colours and toolbars, and basic functionality, such as saving and loading of projects. The editor classes (e.g. ‘ArgumentEditor’) derive from a base class that we have created, which consequently inherits core functionality for 2D graphics management from the ‘QGraphicsScene’. The main differences between each editor are: a) the graphical layout, b) various toolbars and tabs, and c) graphical entity interactions. In theory, one instance of the editor class would be sufficient for all of the above. In reality, however, such approach would be a performance bottleneck. A practitioner working on a safety case requires continuous availability of the resources and potentially reviews the system architecture, pattern and the argument models multiple times. As a result, to maintain good performance when searching for specific entities within (any of) the models, the size of editor’s items



list should be minimal (i.e. as few entities as possible). To counter this drawback it would be possible to define different lists, which would store different element types and graphical information, and each time render on the scene only the elements from the appropriate list. However, every time that the user switches from one view to another (e.g. system architecture to pattern structure), the editor instance would have to save any changes on the current scene and load the one requested along with the corresponding scene assets (e.g. toolbars and shapes) while readjusting certain options, such as zoom levels and font types. Further, any specific interactions, that vary from system to pattern editors, would require further type checking of the elements. Additionally, all this editor-specific behaviour would lead to an enormous (class) file, which can be hard to maintain as a developer. In contrast, with our design approach each editor has its own item list and layouts, and there is no need to manually keeping graphical information nor saving and loading assets during editor changes; as this is handled within the Qt framework (internally) by default when switching the main scene of the application. Hence, the instant the editor changes from a user action, the renderer class simply draws the items list of the new scene. Further, the individual class files are smaller and more manageable due to the reduced number of members, functions and variables, and there is no need for extended function overloading. Naturally, with our approach there is some repetitiveness of code between the editor classes, which translates to a bigger file size for the program. Arguably, this can be a drawback in certain applications where memory storage is a constraint; however, at the current implementation the size difference is almost negligible.

As shown in Fig. 5.5 above, the exchange of information between the program's actions is handled with a coordinator class, referred to as 'EditorAndSceneCoordinator', which is created by the 'ApplicationWindow' class during the program initialisation. For example, when the user decides to save a system architecture and clicks on the save button from the menu tab, the coordinator calls the appropriate editor to execute its overridden saving function. In addition, method specific actions, such as the instantiation, are handled through the coordinator, who has access to all the editors and the MCSU. Specifically, the coordinator calls the model editor and builds the 'ModelElement'

architecture and then passes it to the MCSU. It then does the same for the 'PatternEditor' and then uses the instantiation algorithm to parse through the data. Once the argument is created, it is the coordinator's responsibility to provide the 'ArgumentEditor' with the corresponding structure. In response, the 'ArgumentEditor' creates graphical items to reflect upon the argument structure, which are then displayed on screen for the practitioner to review and potentially tweak. The coordinator has also an established connection, through a member variable, with the HiP-HOPS engine, as a dynamic library, which allows sending and receiving information between the two. As a result, the system architecture is given to HiP-HOPS for analysis, which then provides the outcomes that are stored in the form of MCSU elements for later use. Before the allocation of SILs takes place, the user may choose the safety standard for the project, so that the coordinator will call the correct HiP-HOPS extension; ultimately leading to the appropriate decomposition rules applied.

Continuing the exploration of the tool's framework, Fig. 5.6 below depicts the design of 'ModelEditor'. The 'ModelEditor' essentially acts as a container for the system architecture, as described in the HiP-HOPS metamodel discussed in Section 5.1, and the 2D graphical counterparts. In addition, it defines functionality for failure annotation via various customised dialogues and various input and command interactions. The Renderer is a class provided by Qt for painting the items on the scene. The program creates a new renderer during initialisation and passes a reference in each editor via the coordinator. In response, the editors pass this reference to every newly created item (either shape or arrow) in order to allow for further customisation during the drawing process. Every second the scene is updated multiple times (on average at 60 frames per second), where the active editor is calling the render function on each item in its list.

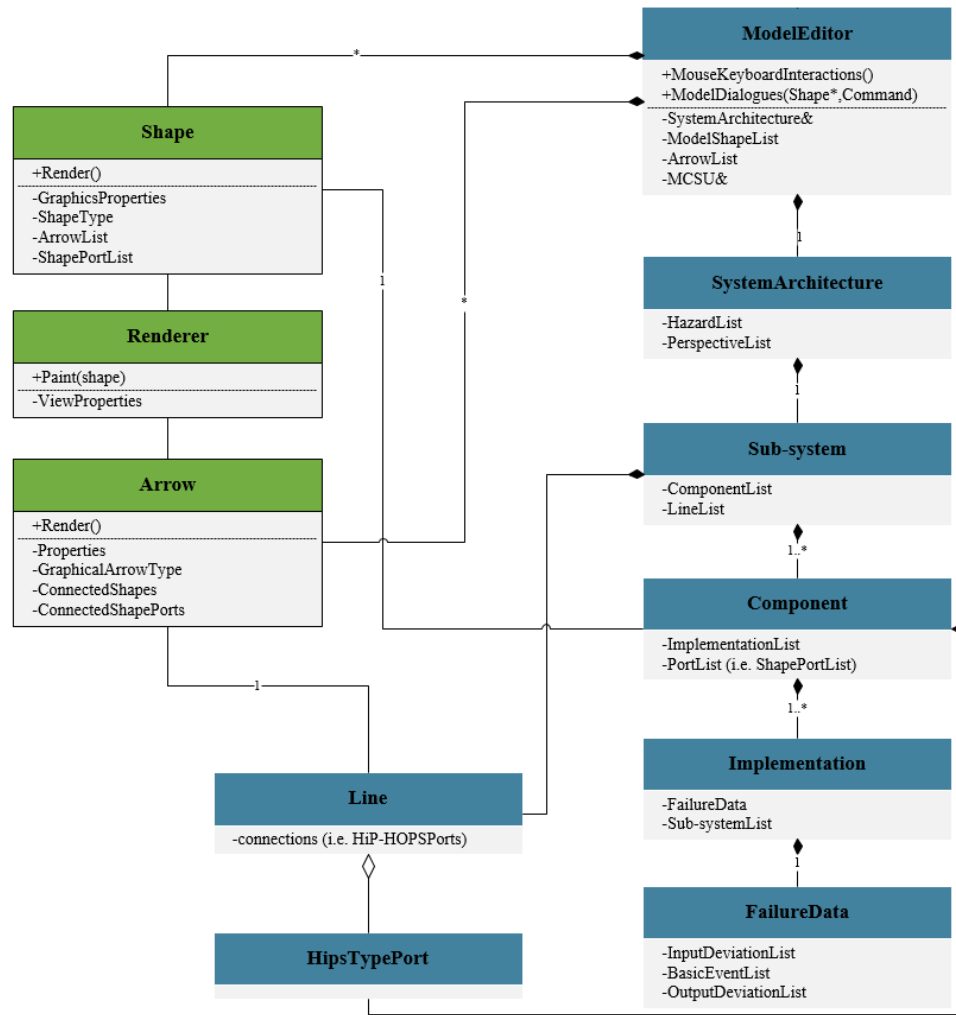


Figure 5.6 The 'ModelEditor' design

The Shape class derives from Qt base classes for creating simple shapes (e.g. a circle) and has been customised to increase complexity of shapes that correspond to specific notations, used in the industry, and alter the drawing mechanisms. The Arrow class also derives from these base Qt classes, but drawing has been customised to accommodate for changes in arrow types and capabilities. For example, there are arrows used for the argument generation that follow the GSN definitions, which means that head-types change based on connection type (e.g. 'InContextOf' connectors have a blank-coloured head). Regarding the capabilities, for user convenience we have added the ability to maintain the arrow on screen even when there is no connection. This way the user does not have to delete and re-create arrows when an alternate connection is needed; in contrast, this behaviour was not included within Qt. Finally, each time the user creates either entity graphically, the editor creates the

appropriate metamodel information. As a result, when the user places a new shape on the scene, the shape generates a new component which is added to the system architecture. With a certain action, the user can add a new component within another component, which effectively creates a new implementation and essentially upgrades the container-component into a subsystem. Similarly, when a connection is established with an arrow item, the tool produces a new line in the metamodel. The user can annotate each component with failure behaviour via customised dialogues that mirror those implemented in Simulink from the HiP-HOPS documentation. This ensures that most of the capabilities provided by the HiP-HOPS engine are available.

The ‘ArgumentEditor’ design, shown in Fig. 5.7, follows a similar philosophy, but reflects the GSN instead.

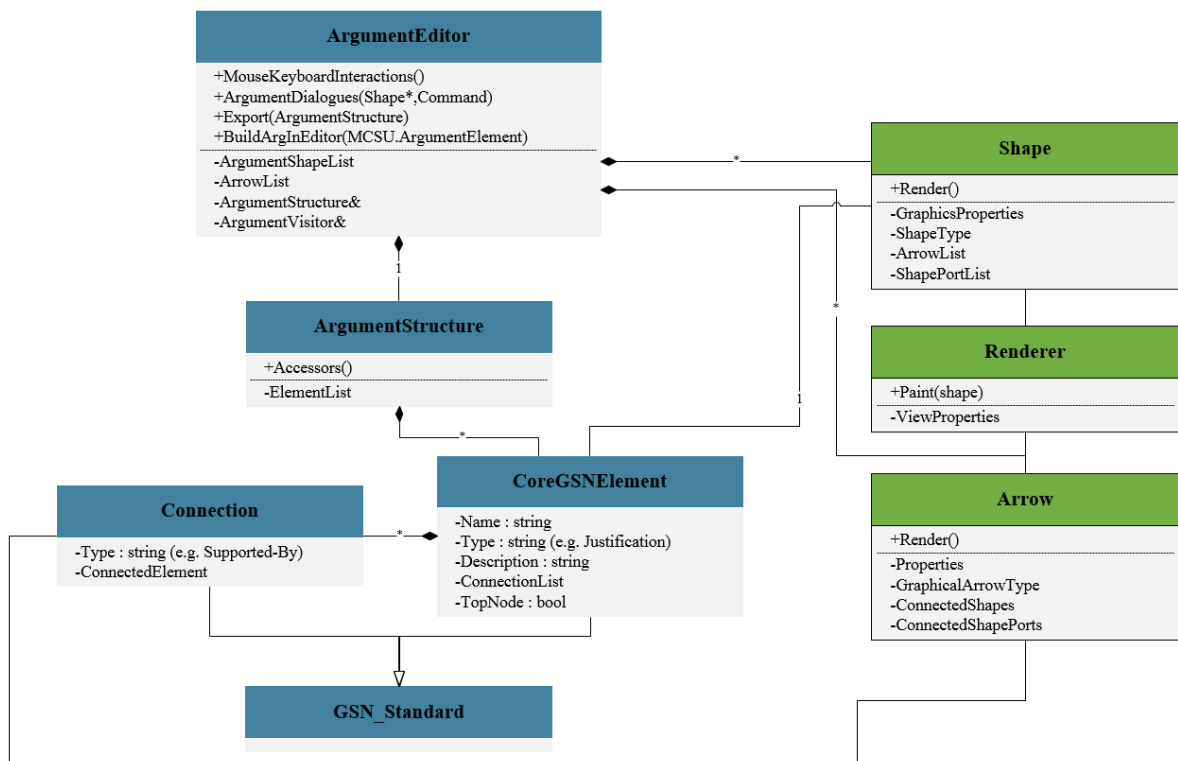


Figure 5.7 The ArgumentEditor design

As seen in the image, the Shape, Renderer and Arrow classes are again used in this part for the graphical representation of GSN elements. The ‘ArgumentEditor’ captures the GSN architecture as a set of interconnected items. Each item has its own connections list, which holds information about the connection type and the supporting elements (children-nodes), as well as the supported elements (parent-nodes). This editor allows the display of an argument structure provided by the instantiation process and also provides the capability to manually compose an argument. Moreover, the editor supports the ability to print the graph structure, as well as the export of the argument into XML files. Currently, export options are limited, but they can easily be extended to enable further customisation in the export syntax. Finally, the argument can be segmented via the module-elements, described in the GSN, with a similar mechanic used in the ‘ModelEditor’ for creating subsystems. Hence, it is possible to place a module-element on the scene and assign it with a subgraph.

The ‘PatternEditor’, shown in Fig. 5.8, is almost identical to the ‘ArgumentEditor’ architecture. There are two major differences: a) this editor incorporates the extensions of element and structural abstractions provided by the GSN standard and b) it implements both a graphical editor and a text editor. The first is important for composing the pattern structures with embedded model-based rules (e.g. via the one-to-many relationships), later used for guiding the argument structure. The second is important for convenience reasons. The user is permitted to either create the pattern using the GUI, by placing and annotating (e.g. the expression of the for-each elements) the elements in the appropriate order or by simply writing (or adding as an external source via an XML file) the pattern following a specific syntax, provided in Section 4 of this chapter.

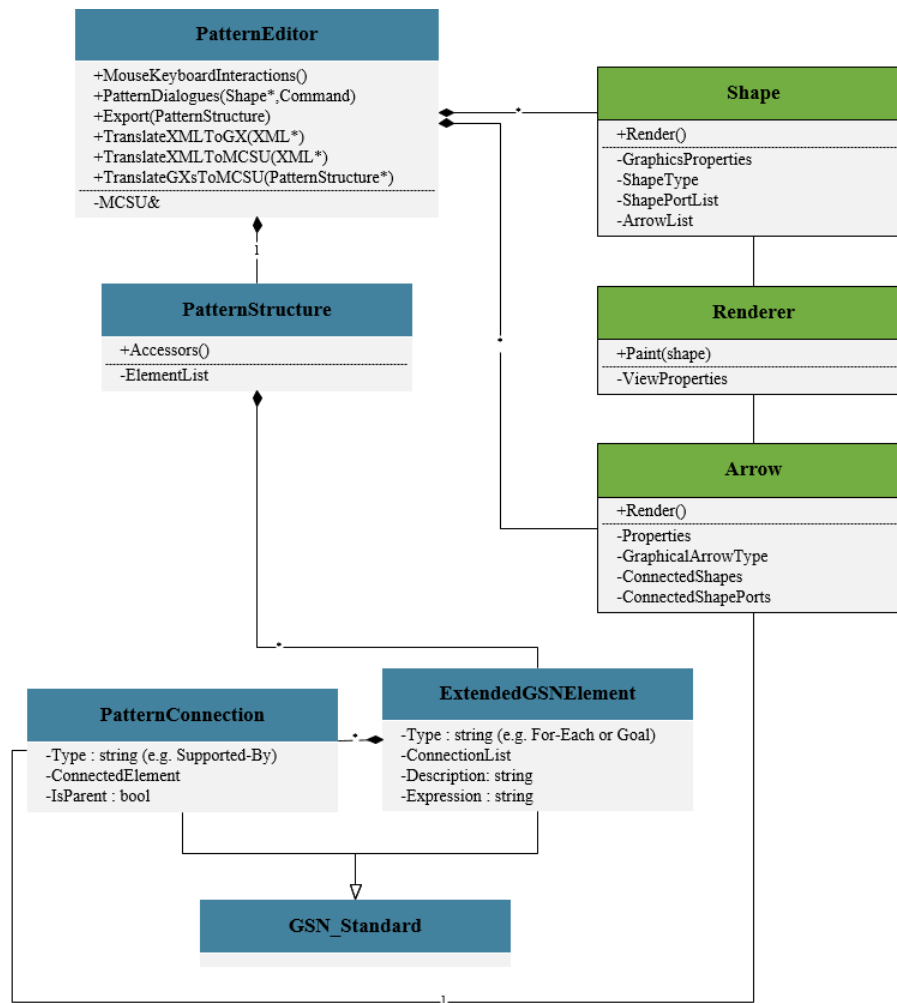


Figure 5.8 The PatternEditor design

Finally, Fig. 5.9 provides the design of the ‘FTViewer’. Note that even though it is referred to as a viewer, this is still an editor that allows the user to create custom fault trees and store them within specific components for potential use later as evidence artefacts. The ‘FTViewer’ core functionality is to create the system tree generated by the HiP-HOPS engine after the analysis is completed. This enables the engineer to have a more technical and direct view of the analysis results. The HiP-HOPS tool, by default, generates this tree and allows viewing via an html file (HyperText Markup Language). However, the layout was not very intuitive and therefore we provide a diagram instead. Similarly to the previously shown designs, each entity is graphically represented with the classes of Shape and Arrows.

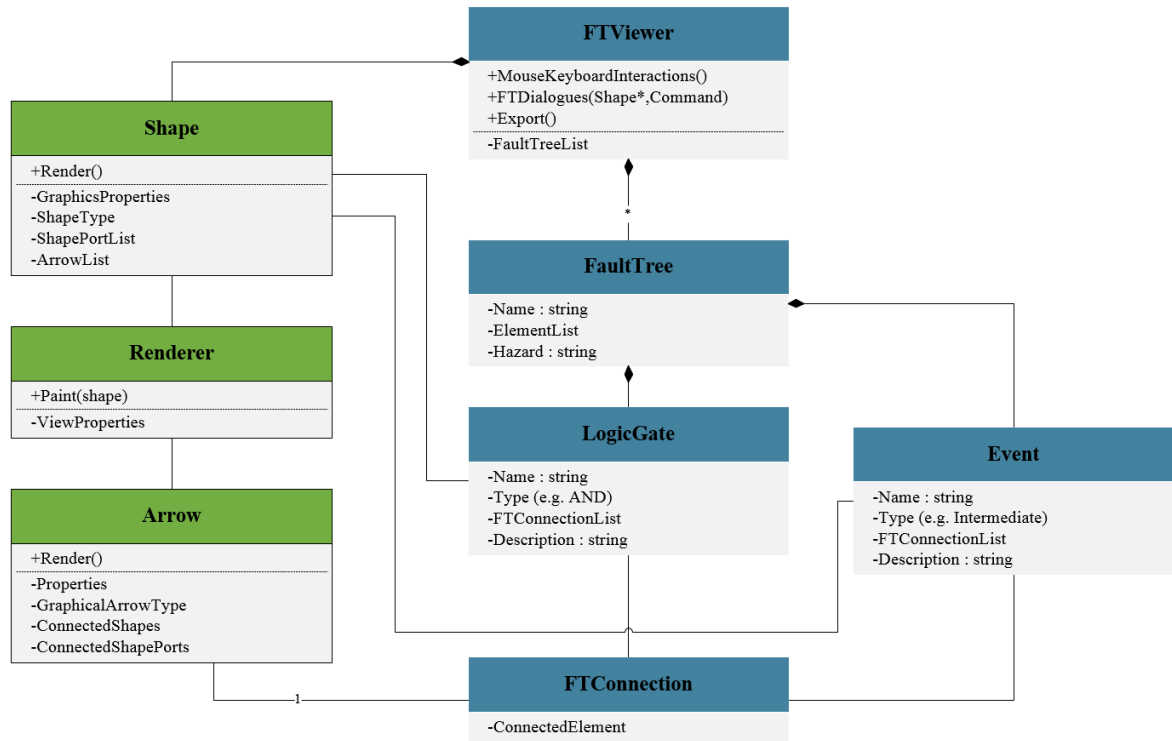


Figure 5.9 The FTVIEWER design

The 'FaultTree' class has a list of elements, which can be either 'Events' or 'LogicGates' and a hazard parameter. The gates can be of any type (e.g. AND), whereas the Events can either be intermediate or basic events (depicted as a rectangle or a circle respectively). The hazard is placed on top of the tree graph and all the constituent elements are structures below it. To establish a hierarchy, elements of either type contain a list of 'FTConnection' instances. The 'FTConnection' class stores the name and identity of just one connected element and is graphically represented with an 'Arrow'. Finally, we expand on the results, typically provided by HiP-HOPS, by allowing multiple trees for each subsystem. This feature can be useful as a justification element in the argument when performing PASA/PSSA during safety assessment in the aviation domain. To do so, the user can segment the model into various sub-models and use the HiP-HOPS engine to analyse them separately. The results are stored within the subsystem element and can be viewed in the 'FTVIEWER' if the user decides so.

With this section, the metamodel-related and framework's technical aspects conclude. Note that the tool includes other utilities, such as an XML parser and sound generation classes, which were excluded from this description for simplicity reasons and to emphasise on method-related aspects.

## 5.4 Pattern Generation and Processing

Despite our effort to keep the method as universal as possible, when composing safety arguments for different domains, there are still aspects that are standard-specific and need to be reflected on the argument structure in order to maintain compliance. To achieve this and control the resulting argument structure, the user has to provide a pattern to guide the process. It has to be suitable for both the safety standard of the target system as well as any further decisions made by the development's stakeholders.

As discussed earlier, the software allows the practitioner to use either a GUI-based or a document-based (i.e. XML) pattern generation process. The former helps developers who are familiar with graphical notations to use the GSN, whereas the document-based approach aims to be clear regardless of prior knowledge with notations and it is typically easier to modify. Once the pattern is complete, the program parses through and builds appropriate structures in the MCSU, which are later directly usable by the instantiation algorithm. Before delving into the specifics of the instantiation process, this section examines the two different ways for synthesising the pattern. This example serves an introductory role, so it will be easier for the reader to follow the pattern found in the case study of Chapter 6. Fig. 5.10 shows a structure of an abstract example in XML format.

```
<Goal type='GOAL'>
  <Name>G1</Name>
  <Description> System {S.Name} contribution to {S.H} mitigated via DAL {S.DAL}</Description>
  <Strategy type='STRATEGY'>
    <Name>A1</Name>
    <Description>Argument over contributors of system {S.Name} to {S.H}</Description>
    <forEach type='ForEach' var='C' ofType='ElementType' in='S.elementList'>
      <Goal type='GOAL'>
        <Name>{C.Name}Goal</Name>
        <Description>{C.Name} contribution to {S.H} mitigated via DAL {C.DAL}</Description>
      </Goal>
    </forEach>
  </Strategy>
</Goal>
```

Figure 5.10: Abstract example of a pattern in the XML-based syntax



In this scenario, the pattern mandates that the argument starts with a goal, which claims that system contribution to a certain hazard is mitigated via allocation of the appropriate SIL (DAL in this case). Then, the strategy is to expand to the underlying architecture of that system and showcase that the contribution of lower-level components to that hazard is indeed mitigated; again, via the allocation of DALs. The syntax is rather simple and follows the generic form of the XML; basically, a root element and its child elements. This hierarchy conveniently resembles that of a safety argument structure, where the top-level goal is represented as the root element, found in XML, and the rest of the argument elements are embedded as child elements (either for supporting or contextual purposes). The utilisation of attributes is mostly restricted in stating the type of GSN elements, which is used by the program to determine the appropriate graphical representation and the topology of elements on the scene. The only exceptions are the 'ForEach' and 'IfThen' elements, which capture various variables as attributes; utilised for traversing through model structures, and help building a model-based argument. For instance, the 'ForEach' element is used as a means to explore system hierarchy and data, such as supporting components contained within a subsystem or multiple hazards related to a function. Based on that information and user-defined behaviour provided through the pattern, the resulting argument changes significantly each time even when working with the same system. In the example shown in Fig 5.10, it helps parse the list of components found in System1 and when it finds a specific type, it creates a goal with the respective information. The 'var C' is essentially an iterator that points to an element, of type 'ElementType', from the parent's list; hence, 'var C' points to a different object during each iteration and its actual meaning remains within the scope of the 'ForEach' element (until the '</forEach>' tag is found). From the XML, either created within the tool or imported, the program will generate a graphical structure, which the user can view and interact with. Fig. 5.11 shows the graphical representation of the XML pattern presented in the previous figure (i.e. 5.10).

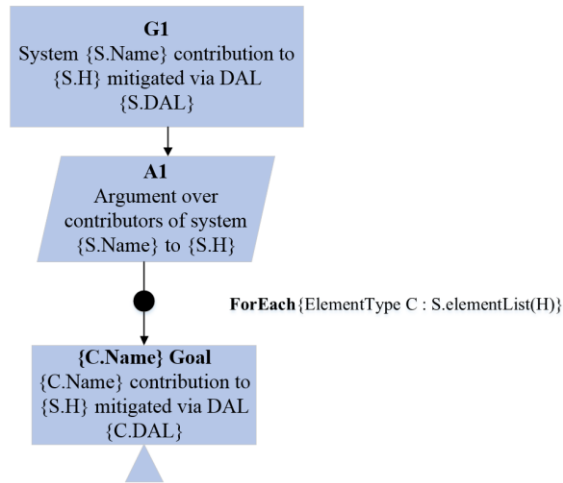


Figure 5.11: Graphical representation of the pattern example

As seen in the figure above, any information related to the system, such as element names, hazards and SILs, is handled through parameterised expressions. For instance, the ‘{S.Name}’ is the name of the system the program is examining at the time and the {S.DAL} is its allocated SIL (for the civil aviation industry). These expressions are instantiated, or simply replaced by the actual information, in the resulting argument. Fig. 5.12 provides the generated argument for our example.

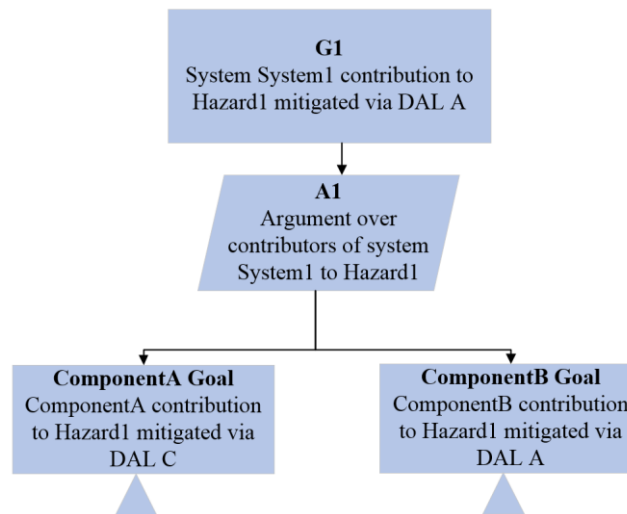


Figure 5.12: Resulting Argument Structure

Despite the seemingly minor discrepancies due to the size and complexity of this example, the argument is significantly different compared to the pattern. The parameterised variables have been replaced by the appropriate values and the ‘ForEach’ elements have been removed; as a result, the

strategy is now connected with suitable goals related to the components (i.e. ComponentA and ComponentB) that were found within the model in System1.

## 5.5 Pattern Instantiation Algorithm

The core design, storage units and the pattern syntax of the developed framework have been discussed in previous sections of this chapter. Now we examine the part of the software that implements the automatic instantiation process. The algorithm is presented in pseudocode and any language-specific details are omitted. The intention is to keep the code clear and readable independently of the reader's background in programming languages. The core parts of the algorithm are summarised below:

### **EditorAndSceneCoordinator::Instantiate (MCSU)**

- 1) If (PatternMCSU AND ModelMCSU exist)
  - a) Pick the top-level nodes of the model and pattern
  - b) Generate an empty top-level node for the argument structure
  - c) ParseAndCreate (pattern\_top, model\_top, argument\_top)
  - d) Pick argument editor and pass the completed argument structure
  - e) Request the argument editor to graphically create that structure
- 2) Else
  - a) Inform the user via dialogs that certain information is missing and stop the process

### **ParseAndCreate (pattern\_node, model\_node, argument\_node)**

- 1) If (pattern\_node is a basic node)
  - a) Instantiate title, content and other relevant information
  - b) For each child\_node in pattern\_node
    - i) If (child\_node is basic\_node)
      - (1) Generate an empty argument\_node\_child
      - (2) Set argument\_structure's\_last = argument\_node\_child
      - (3) ParseAndCreate (child\_node, model\_node, argument\_node\_child)

- (4) Update argument\_node for the new child
  - ii) Else if (child\_node is ForEach\_node or IfThen\_node)
    - (1) ParseAndCreate (child\_node, model\_node, argument\_node)
- 2) Else if (pattern\_node is ForEach\_node)
  - a) ParseForEach (pattern\_node, model\_node, argument\_node)
- 3) Else if (pattern\_node is IfThen\_node)
  - a) Pick the user-defined condition from the IfThen\_node
  - b) If (ProcessCondition(condition) is true)
    - i) Create an empty argument\_node\_child based on the child of IfThen\_node for true
    - ii) ParseAndCreate (child\_node, model\_node, argument\_node\_child)
    - iii) Update argument\_node for the new child
  - c) Else
    - i) Create an empty argument\_node\_child based on the child of IfThen\_node for false
    - ii) ParseAndCreate (child\_node, model\_node, argument\_node\_child)
    - iii) Update argument\_node for the new child

**ParseForEach (pattern\_node, model\_node, argument\_node)**

- 1) For Each (model\_node\_child in model\_node)
  - a) For Each (pattern\_node\_child in pattern\_node)
    - i) Create an empty argument\_node\_child
    - ii) ParseAndCreate (pattern\_node\_child, model\_node\_child, argument\_node\_child)
    - iii) Update argument for the new child
  - b) ParseForEach (pattern\_node, model\_node\_child, argument\_structure's\_last\_node)

Although we refer to the algorithm as one unit, it is basically implemented as three member functions (highlighted with bold fonts) within our editor manager class. The first function, **Instantiate()**, initiates the process by checking whether or not the user has provided all mandatory information (i.e. model information and structure, as well as the pattern structure) for the instantiation process. In fact, the

program not only searches for availability of information but also applies model-checking, for both the system architecture and the pattern, to make sure that certain rules were followed. For example, if the pattern has evidence directly under a strategy, this is considered to be an invalid structure and the program will prompt the user to review the pattern. If the information is substantial and correct, the program proceeds and creates the top-level element of the argument structure and calls the second function, 'ParseAndCreate()'. This function is recursive and is considered to be the major body of our algorithm for building an argument based on the provided pattern and system architecture. It requires only three parameters, the current nodes from the pattern, model and argument structures, respectively. The pattern structure itself acts as the general guide for the argument structure, whereas the parameterised expressions found in pattern elements, either as embedded text or other input methods, mandate the information needed from the model for the corresponding argument element. The model is used for either providing the information that will replace the parameterised expressions or for guiding the number of times certain parts of the pattern need to be repeated so that the resulting argument conforms to and argues about the whole system architecture.

In practice, the function starts by picking the top-level pattern, model and argument nodes. If the pattern-node is of a basic type (e.g. Goal/Claim), it instantiates the provided argument-node and then proceeds to look for child-nodes within that pattern-node. If the pattern-child-node is a basic node, it creates a new child for the argument and then the function is recursively called with these instances of child-nodes. Once the recursion is completed, the branch for that first child-node has been discovered and appropriate arguments have been created. Hence, the initial argument-node is updated for all these newly created elements (set as their parent-node). If the child-node of the pattern is either an 'IfThen-node' or a 'ForEach-node', the function does not create a new child-node for the argument, as these should be excluded from the argument structure; instead, it simply parses again with that pattern child-node (i.e. 'IfThen' or 'ForEach') and the same argument-node it had in the previous parse.

However, when the pattern-node itself is an 'IfThen-node' (in step 3 of 'ParseAndCreate()'), the function creates a new child-node for the argument, if the condition set by the user is true, and parses again with the newly generated argument child-node. This type of pattern-node (i.e. 'IfThen') enables the conditional creation of argument subgraphs and proper use allows the user to create complex argument structures. For example, assume a scenario where in the pattern an element 'Goal A' is connected with an element 'IfThen', which in turn is connected with an element 'Goal B'. In this case, the argument structure, supposing that the condition is true, will be an element 'Goal A' directly connected with an element 'Goal B'.

Finally, if the pattern-node is 'ForEach', the 'ParseAndCreate()' function calls another recursive function, the 'ParseForEach()'. This function is responsible for traversing through the model information according to the user-defined expressions found in the 'ForEach-node'. Then, it calls the 'ParseAndCreate()' for each child-node of the 'ForEach-node' to instantiate the newly created argument elements. Once that step completes, it then recursively calls itself to look deeper into the system architecture. This process essentially repeats the pattern elements that follow the 'ForEach' element and helps build an argument structure that covers the system architecture.

When these recursive functions are completed, the Instantiate() function passes the resulting argument structure to the corresponding editor for building its graphical representation. At that point, the user can view the generated structure and potentially apply changes either manually or through the pattern and the model and then request another instantiation. Note that various utility functions used in between or after the aforementioned steps are omitted either for simplicity reasons or they are not relevant to the method proposed in this thesis. For example, a set of functions that process input or text provided by the user, such as names and expressions, and translate it into useable parameters for the program. Other examples are the functions that handle the graphical topology of the argument structure and scaling values of 2D shapes based on the embedded text's size.

## 5.6 Summary

The previous chapter introduced the concept behind the proposed method and succinctly reviewed relevant approaches. It also explained the key features of the method and its metamodel in detail and provided an example of the method's application on an abstract system architecture. This chapter was focused mostly on elements of the metamodel and how these are connected. Then, it delivered the technical information of how the framework was designed and developed for implementing the method proposed in this thesis. Finally, the chapter concluded with a brief presentation of the instantiation algorithm (in pseudocode) responsible for the automatic generation of argument structures.

## Chapter 6 Case Study

---

Up to this point, the proposed method has been partially (through the DAL decomposition process in Chapter 3) or fully (i.e. example in Chapter 4) examined with the use of abstract, simple examples. However, it is important to gain a better understanding of the procedures involved and more sufficiently evaluate the outcome it produces. Therefore, this chapter introduces and applies the method on a more realistic example of larger scale than what has been demonstrated so far. Specifically, this case involves an abstract architecture of a brake system that can be found in the aviation industry and it is based on the technology and specification of the Boeing 787 Dreamliner. The reasons behind the choice of this specific case study are twofold. On one hand, this aircraft is considered representative as it is relatively modern and most of the braking technologies we examine can be found in other aircraft. On the other hand, Boeing 787 has moved into a complete electric braking system that facilitates our method, at this prototypical stage, by focusing the assessment on system interactions and systematic failures rather than requiring additional physical and statistical techniques for accounting random failures (e.g. due to material degradation). This case, as a more complex design, allows not only to inspect the produced argument for plausibility, but also helps to visualise how the method could perform in real-world applications. Moreover, this example allows for a more accurate evaluation of the implemented framework, in terms of performance, against more extensive and complicated system architectures. Before delving into the application of the method on this case study, it is necessary to review the technology behind braking systems, used in the aviation sector, as well as supplementary functionalities implicated in the design. Understanding how the system works allows us to have a clearer view, from an engineer's perspective, and better evaluate what sort of failures might pose a threat to the safety of an aircraft.

### 6.1 A brief history of brake technologies

The very first aircraft were not designed with a built-in brake system for landing. They rather relied mostly on the friction between the tail skid and the ground. Naturally, the lightweight airframe, the relatively low speeds and the soft ground found on the fields used for landing were significant factors



for making this possible. However, as the technology was maturing and speed and payload needs increasing, the design and instalment of a brake system became necessary. From the early days and until now, the main role of the braking system has remained the same; to decelerate the aircraft, completely stop its motion and prevent it from moving again over longer periods of time during parking. To accomplish that, independently of the underlying technology, the braking system applies friction on the wheels and the kinetic energy of the aircraft, during motion, is converted into heat and exerted.

The braking system can be viewed as three basic components: a) the operation part, b) braking unit and c) command transmission mechanism. Each one of these subsystems has undergone significant changes over the years to either improve the braking function of an aircraft, due to increased needs, or to enhance other aspects.

#### 6.1.1 Operation Part

The operation part of the aircraft brakes is the physical form of control given to the pilots for activating the brake function. Initially, pilots were using a lever that was activating all the brakes simultaneously, on all the wheels, resulting in a more symmetrical braking. Later, the wheel brakes were combined with the pedals responsible for controlling the rudder, since switches or levers were not convenient considering that the pilots need to control the motion in other axes, as well as using other functionalities. Moreover, the use of two different brake pedals, one for the left and one for the right wheels, allows for differential braking with the application of different pressure on each pedal. This effect is useful for enabling a supplementary way for directional control (i.e. vertical axis, aka yaw), which is especially useful in situations where the rudder is not performing well, due to low aircraft speed and strong winds, or where there is no nose wheel to assist in steering.

#### 6.1.2 Transmission/Actuating Part

The transmission refers to the mechanism that basically propagates the pilot's braking command to the braking unit. At first, this control input transmission was mechanical, typically a network of

braided steel cables. The pilots could use the brake lever or pedals to create enough tension on the cables and eventually transfer that force to the braking unit, which would start applying friction on the wheel. However, this proved to be ineffective for larger aircraft due to the limited braking power and the nature of the materials. Cables are susceptible to stretching or fraying; thus, potentially leading not only to ineffective braking but also to complete loss of braking. In addition to these safety concerns, cable-enabled brakes required frequent replacement and maintenance cost was too high. Note that the reason behind the high cost was not only related to the continuous need for replacement materials, but also to the time the aircraft would be unavailable for maintenance reasons. These factors led aircraft engineers to cast away this technology, especially for large aircraft. The most suitable substitute, and arguably the most prevalent even today in aviation, are the hydraulically activated brakes. Naturally, there are many variations with this technology as well. The most common, since they can be found in the automotive industry, are the independent systems that use master cylinders and a separate reservoir (either remote or built-in to the master cylinder) for hydraulic fluid. The master cylinder can be viewed as a converter of force into hydraulic pressure. Once the pilot applies pressure on each of the pedals, a pushrod moves the main piston and a spring within the corresponding cylinder's bore. This entraps the hydraulic liquid into smaller space, thereby increasing pressure, which further forces other pistons within the brake assembly. The latter brake pistons push a set of brake pads or linings against a brake rotor (firmly attached to the wheel rim) and through friction the wheel slows down. The more pressure the pilot applies on the pedal, the more the pressure increases throughout the system and therefore, the linings are pushed onto the rotor with greater force. However, the braking power of these systems is still limited and mainly effective in isolation for smaller aircraft. Higher performance and larger aircraft are equipped with power brake actuating systems, which use the aircraft's general hydraulic system for applying brake pressure. The amount of fluid and force required simply cannot be handled by a master cylinder. In power brake systems, the master cylinder is replaced with hydraulic pumps and meter valves. The pumps provide a constant stream of fluid throughout the linkage and towards a set of valves (Aeronautics Guide, 2020). The meter valves are responsible for adjusting the control valves, which in response regulate and release

the right amount of fluid to ultimately apply the intended pressure at the brake unit. In braking systems that incorporate redundancy, there may be intermediate valves for selecting which pump's pressure is suitable based on the mode of operation (e.g. normal or alternate mode). This type of braking system is well-known since it resembles the Wheel Brake System (WBS) described in the ARP4761 (SAE, 1996:192) safety standard. Note that even though the design of WBS is used mainly for illustration and research purposes, there are brake systems that work in a relatively close fashion in the aviation industry, such as the Boeing 737 (B737 Technical Site, 2020).

Despite the sufficient performance of hydraulic brakes, engineers still pursue new means to optimize the braking system. One of the main concerns when designing an aircraft is weight. Independently of the raw power of the engines an aircraft is equipped with, its weight is correlated with the fuel consumption. Thus, a weight increase would inevitably lead to higher cost per flight, more emissions to the environment, and potentially heavier strain on the main frame. With that in mind, the pipes, fluid, and actuators required to form a hydraulic brake system contribute a considerable amount of added weight. As a result, the most recent adaptation of the actuation system revolves around electricity and electromechanical brakes, which require mostly lightweight wires and equipment. The use of electricity is not an entirely new concept as digital parts are already installed in various commercial aircraft for utility functions. For instance, Brake System Control Units (BSCUs) are often utilized for sending the control input from the pedals to the meter valves or providing feedback to other systems (e.g. control system state in the cockpit). Additionally, many hydraulic pumps or the anti-skid systems use electricity as a source of power. However, in the context of electrically activated brakes, the electricity is used for the whole operation of the braking system. The pilots press the pedals, similarly to the other systems, and send an electronic signal to the BSCU. Then, the signal is transmitted to other components and finally arrives to the electric brake actuators (EBAs), which are special kinds of pistons, that electromagnetically move and press against the brake discs. Modern commercial aircraft, such as the Boeing 787, already use this technology. Current research also explores the feasibility of regenerative brakes for aircraft (EPSRC, 2010). This technology is

estimated to be able to generate energy from the braking during landing, store it and then use it for the taxiing phase. It is a promising mechanism that already finds use in the automotive industry; however, at the time of writing this thesis, no such system is installed in any of the operating commercial aircraft.

### 6.1.3 Brake Units

The brake unit, also referred to as the brake assembly, exists on the aircraft's main wheels; although, a few aircraft with a nose wheel also have brakes installed there. Initially and until the early 1940s, aircraft were using drum brakes. These had a cylindrical structure that would move along with the wheel and a set of pads (linings) and springs enclosed within that structure. During a brake command, the linings would apply pressure towards the cylinder's inner walls and through friction the wheel would start to decelerate. However, this type of brake was quickly replaced by the disc brake, which was better at handling the generated heat; thus, leading to more consistent braking performance. The disc brake involves a disc (or rotor) firmly attached on the wheel, often via bolts, and an immobile calliper with typically hydraulic-powered pistons and embedded pads. Once a braking command is provided, the pistons installed on the calliper close and attempt to lock the disc in place by creating friction, which result in slowing down the rotating wheel. When the brake is released, a spring within the assembly pushes back the piston and the friction on the disc terminates. This technology on the brake assembly is still dominant in modern aircraft for its performance. For example, the ability of the carefully designed rotors to repel water, when landing in wet runways, is far greater than any previous mechanism and they are also less likely to lock in place during heavy usage. Despite their success, there have still been advances, such as dual-disc or even multiple-disc brakes. Therefore, each aircraft is equipped with the appropriate disc brake based on its weight, speed, region of operation to account for weather conditions and load capacity. Moreover, over the years the materials used for the disc brake and the rotors have varied greatly, from steel-based to a mixture of carbon and other composites. The latter have succeeded in decreasing the overall weight and are able to withstand and more effectively drive away the heat generated from friction compared to the steel-based variant.

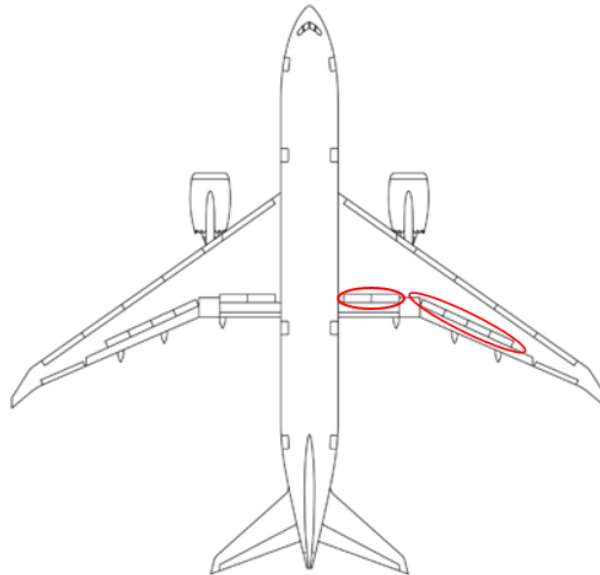
As mentioned earlier, these are important factors in aviation, and as a result high-performance and large aircraft opt for these materials.

## 6.2 Auxiliary Braking Mechanisms

Most commercial aircraft have additional systems to assist in the braking function, especially during the landing phase. The most common ones are the spoilers and the reverse thrust. In the next section, these two systems are briefly reviewed since the Boeing 787, whose system architecture this case study is based on, incorporates these supplementary components.

### 6.2.1 Spoilers

Spoilers are relatively large panels installed on the upper side of the wings as shown in Fig. 6.1.

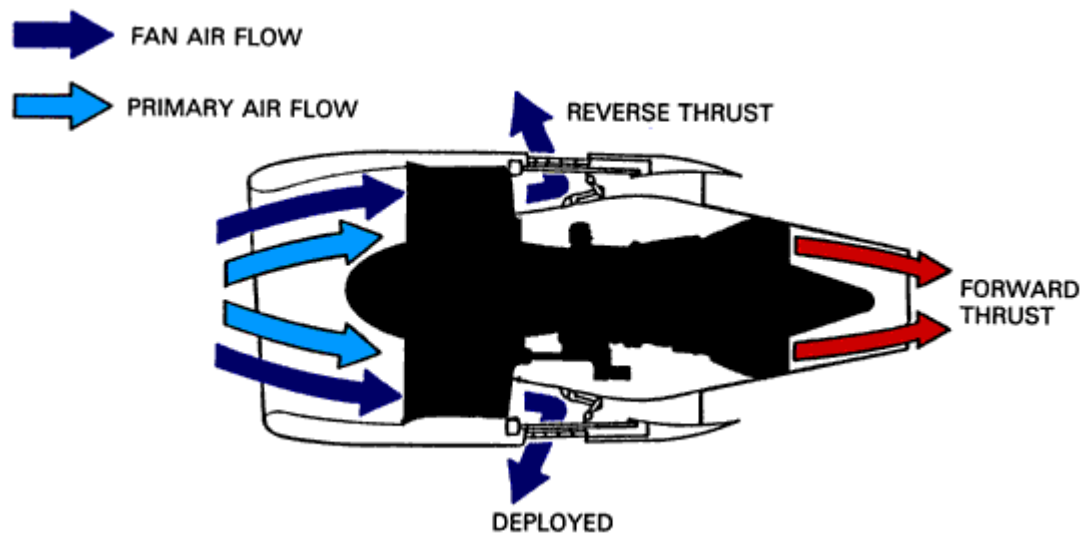


*Figure 6.1 Wing Spoilers*

They are typically manufactured out of composite materials, often found as carbon laminate. Spoilers play a supporting role in braking function by intervening with the airflow over the wing. This reduces the lift generated by the wings and as an effect, all of the aircraft weight is directed towards the wheels and that leads to more effective braking. In addition, spoilers help increase aerodynamic drag that assist in slowing down the aircraft. These effects can be useful not only during the landing phase, but also in the case of a rejected take-off. Modern aircraft have incorporated automated mechanisms for the use of spoilers, but pilots are able to activate them manually during unexpected events.

### 6.2.2 Reverse Thrust

The reverse thrust is yet another mechanism for supporting the wheel brakes. The air, which normally escapes at the backend of the engine, can be diverted towards the frontend; thus, creating the opposite effect similarly to the reverse gear in automobiles. However, the most common use for the reverse thrust is for deceleration purposes, during landing or rarely during flight, rather than enabling a reverse motion to move the aircraft from the gate. To engage the reverse thrust of the Boeing 787 during landing, the pilot pulls two levers as soon as the wheels touch the ground. This displaces a set of blockers within the engine and opens a side door. The air is therefore unable to escape from the back and instead is forced to pass through the door and towards the opposite direction. This process is visualised in Fig 6.2 with the air flow indicated with arrows.



*Figure 6.2 Reverse Thrust (Aerospaceweb.org, 2018)*

During normal operation, the air (in red arrows) escapes from the backend, whereas during reverse thrust, the air (in blue arrows) is directed to the side and applies force forward. Most aircraft have two different settings for the reverse thrust, the idle reverse and the max reverse. The former is used during normal landing whereas the latter is mostly reserved for either hot or high-elevation airfields. Despite the ability of reverse thrust to help in slowing down the aircraft, it should be noted that its efficiency varies based on the aircraft speed. Therefore, it is typically engaged at relatively high speeds and the pilots terminate its operation when the aircraft reaches a relatively low speed, so that the aircraft can

continue its course on the runway, as it is entering the ‘Taxiing’ phase, and also prevent reverse thrusters from throwing debris in the engine intakes.

## 6.3 Additional Functions

Apart from the physical systems in place for enabling and maintaining effective braking, engineers have implemented and installed several functions, which engage mostly automatically, for smoothing the braking and increasing efficiency as well as guaranteeing a safer operation. The functions discussed in this section are also installed in the Boeing 787 aircraft, and therefore might be useful to quickly review them.

### 6.3.1 Anti-skid Protection

Flights are frequently realised during unfavourable weather conditions, such as heavy rain. In the scenario where the runways are humid and therefore slippery, the wheels might start skidding when braking is engaged. Naturally, this may also happen in cases where the pressure applied to the brakes is too high. To avoid these incidents and more generally improve the efficiency of the brakes, anti-skid protection systems are employed. To stop the wheel from skidding over the surface, the anti-skid controller determines the wheel’s rotation rate from the aircraft speed, which is received as input. Once the wheel stops spinning abruptly and the ratio between the rotation rate and the aircraft speed deviates, the system detects that skidding occurs. To prevent skidding from continuing, the anti-skid system takes control over the brakes and releases the wheel until it reaches the appropriate rotation rate. Note that this process initiates in fractions of a second.

### 6.3.2 Temperature Indication and Fuse Plugs

Since braking is accomplished mostly through friction at the brake assembly, it is natural to presume that temperatures could rise high and not only affect the brake performance but may also affect the condition of the tyres. Therefore, pilots monitor the temperature from the control deck in the cockpit. The indication is shown in numerical values, in the range of 0-4.9 for normal levels, whereas everything above requires further attention. Moreover, to prevent the tyres from overheating and

exploding, leading to loss of control, engineers have placed a physical solution in the form of fuse plugs. These are constructed from materials vulnerable to heat, so that when the temperature rises above a certain threshold, the plugs melt and the trapped air within the tyres is released safely.

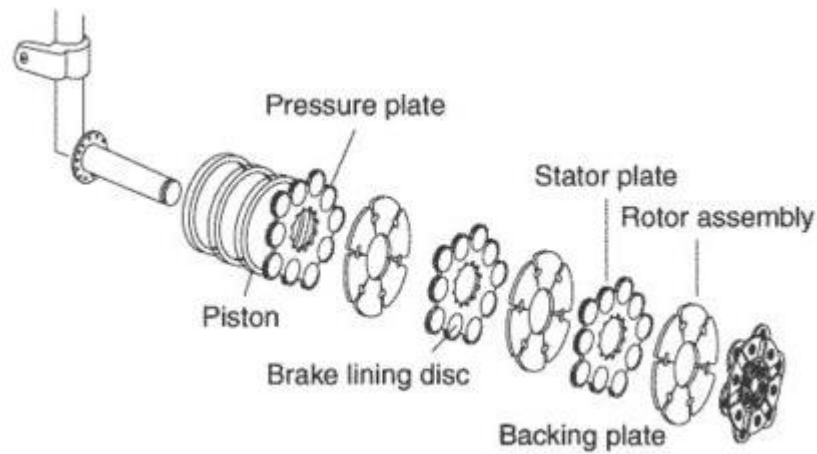
### 6.3.3 Autobrake

Another useful function of a contemporary braking system is the autobrake. Arguably, braking in a straight line with low speed and normal weather conditions can be quite easy for experienced pilots; however, braking during landing with adverse conditions (e.g. strong winds) can be a challenge. The pilots need to maintain appropriate angles in all three axes, initiate brake and other functions whilst still travelling at over 160mph (Boeing, 2016). As mentioned earlier, the vertical axis is handled through the rudder pedals, which are also used for braking (toe brakes). Thus, to facilitate the braking during landing, the braking system is equipped with an automated brake function. In fact, the moment the aircraft touches the ground, the autobrake engages and attempts to slow down the aircraft at a modifiable pre-defined rate. Finally, except for promoting a safer braking, the autobrake is calculated to operate in a manner that reduces the wear of the wheel assembly (brake pads and tyres).

## 6.4 Boeing 787 Dreamliner - Brake System

Now that various brake technologies, and how they operate, have been reviewed, this section discusses the brake system examined in this case study. The system architecture is based on Boeing 787 Dreamliner, a series of aircraft manufactured in the late 2000s and completed its first flight in 2009 (Boeing, 2020). The different models vary in size and capabilities but follow a similar architecture. For example, the 787-8 has the capacity of 242 passengers and can take off with 227 tons of weight whereas the 787-10 can transfer up to 330 passengers and 250 tons respectively (Boeing site, 2020). To decelerate, and eventually stop, an aircraft that averages 180 tons whilst travelling (on ground) at an average of 170mph requires great braking power. To achieve such a feat, the Boeing 787 is equipped with 8 different braking assemblies, each per main wheel. As for the brake assembly, it features a modernised edition of the multi-disc brake, the segmented rotor-disc brake. Fig 6.3 illustrates a generic design of this technology.



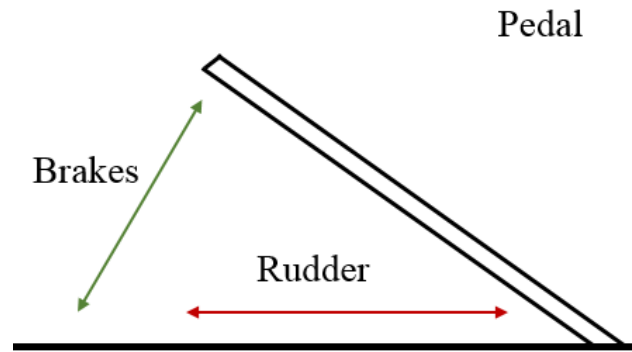


*Figure 6.3 Generic Segmented Rotor-Disc Brake (Academic, 2020)*

Specifically, the Boeing's brake incorporates a total of 5 rotors paired with stators. The rotors are discs with small openings (for water and heat to escape) attached to the rotating wheel. The stators are immobile flat plates covered in isolated blocks, for better heat management, with adhered brake linings. The stators are placed adjacently to the rotors. Once the brakes activate, the pistons in the brake assembly push a special plate, which correspondingly applies pressure to the stack of rotors and stators; thus, creating an immense amount of friction suitable for heavy-duty braking. As for the actuation means, Boeing 787 has discarded the traditional hydraulic system for braking, and instead uses electricity to power the braking mechanism. In fact, it is the first commercial aircraft to incorporate a fully electromechanical brake system. This not only reduces the weight of the aircraft, by replacing hydraulic pipes and fluid with wires, but also facilitates the effort of maintenance. Moreover, electric brakes allow for a better on-board monitoring of the brakes (e.g. for wear) and other health reports. Note that as for most of the aircraft components, there are at least two suppliers for the brake system, Goodrich/UTC and Safran Landing Systems (Goodrich, 2020; Safran, 2020). The client can choose before purchasing, but due to the lack of details with regards to the variations of the architecture of the brake assembly between the two suppliers, this case study considers them to be identical. As for the operation mechanism, the 787 aircraft utilises the standard pedals, which combine the rudder control with the brakes. To handle the rudder the pilots push the pedals forwards

(using the heel), whereas to activate the brakes they press the top part (toe brake) towards the ground.

Fig. 6.4 indicates that motion with a green arrow for the brake and a red arrow for the rudder control.



*Figure 6.4 Pedal Motion Command*

The Boeing 787 incorporates two separate processing BSCUs, left and right, which manage the left and right groups of wheels (4 wheels per group) respectively. Their main role is to process input (from pedals) and send the corresponding commands to the appropriate Electronic Brake Actuator Controller (EBAC). In addition, they receive feedback from the sensors of brake assemblies and send various messages (e.g. warnings) to the cockpit or other systems (e.g. autobrake). Sensor feedback often includes temperature, lining wear, pressure and wheel rotation rate for the anti-skid system. The EBACs are responsible for interpreting the signals from BSCUs and controlling the brake rate. They also send information about their state back to the BSCU for health monitoring. Each EBAC utilises a separate Electric Brake Power Supply Unit (EBPSU) as a source of power. Finally, each brake assembly has four Electric Brake Actuators (EBAs), engaged through the EBACs, that apply pressure on the stack of rotors and stators and decelerate the wheel. Fig. 6.5 showcases a simplified version of 787's brake system architecture.

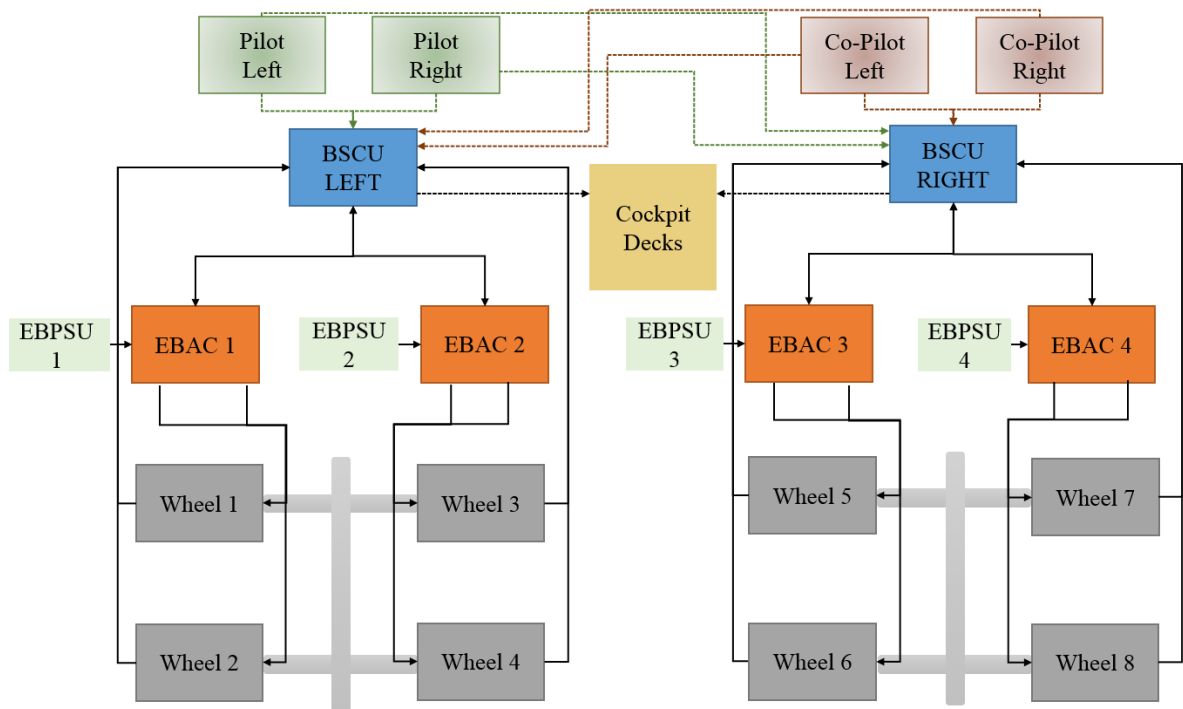


Figure 6.5 Base Architecture of 787 Brake System(Maré, 2017:187)

Note that both pilot's and co-pilot's pedals communicate with both BSCUs, as the left and right pedals are responsible for commanding the left or right columns of wheel-brakes, respectively.

The Boeing 787 Dreamliner can be considered as one of the safest aircraft, as to the best of our knowledge there have been no serious crash accidents ever reported in media since its release, also mentioned in an article by (Zetlin, 2019). Its innovative systems and specifically the electric brakes have lessened the mechanical complexity substantially. The advanced monitoring systems allow for continuous fault detection and wear status. Most brake components, such as the actuators, are modular and allow for quick service. In addition, the replacement of hydraulic brakes eliminates a large portion of random hardware failures from material degradation. Instead, with electric brakes, most of the failures are systematic, being either related to software or hardware systems. As explained in Section 3.2, these failures can be addressed with DALs according to the ARP safety standard. The latter was of major importance when reviewing various aircraft as candidates for this case study. Generally, to truly evaluate the brake system, 787 aircraft have undergone rigorous testing processes. The software of the EBAC has been validated and verified by the suppliers Goodrich and Safran as well as other

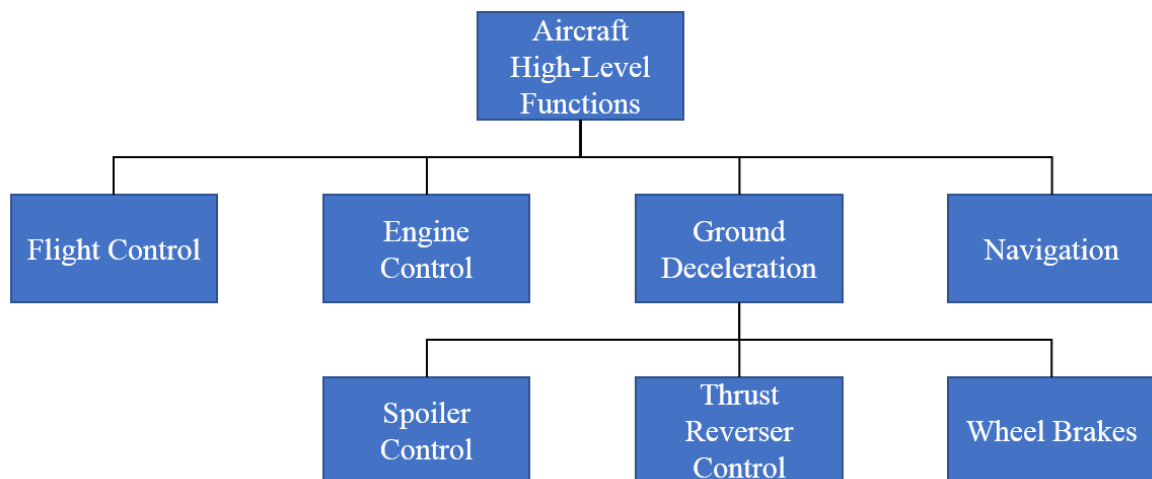
third-party companies. There were no details on the methodology, but effort was focused on developing a software specification in line with the functional specification and then verifying that the implementation follows the software specification. This was presumably achieved via feeding the input into the software and then evaluating the output against the specification (i.e. black-box testing) (Nidhra, 2012). As for the brake system as a whole, including the hardware and software interfaces interaction, there have been tests on all components individually, system integration tests, as well as tests on the actual aircraft. Regarding the latter, stakeholders persisted on testing vital systems, such as the braking, in normal operation and under certain restrictions. For example, the assessment of the behaviour in the case of a rejected take-off was extensive. The reason behind this decision was the fact that the aircraft is far heavier during take-off compared to the landing phase, where most of the fuel is consumed, and runway distance is constrictive. To further intensify the process, the brake linings are typically overused and one of the brake assemblies shut off. In fact, the 787 is safe to operate, assuming sufficient condition of the equipment, even with only six functional brake assemblies.

The novelty of this brake system is that it reduces the amount of common cause failures, for software and hardware, compared to a hydraulic setup. A fluid leak could potentially affect an entire wheel brake. Instead, the 787 establishes four individual actuators per wheel whilst performance remains relatively unaffected even with three operating EBAs. It is also equipped with a different EBAC, supported by its own software, per two wheels. This means that failure of one EBAC will only affect those wheels and not the entirety of the brake system. Further, these EBACs are paired with a separate power unit for autonomy. Considering that the aircraft is operational with only six functional brake assemblies, it means that even if one EBAC fails for some reason, the aircraft should face no serious danger. Finally, each wheel assembly has its own sensors, meaning that absence of feedback, from one or two wheels, should not impact the information received in the flight deck to the point of compromising safety during braking. Finally, the existence of other deceleration means, such as the

spoilers or engine thrust reversers, supported by different systems, further decrease the likelihood of common errors. All of the above indicate that functional independence is supported to an extent.

## 6.5 Function Modelling

Despite having explained the base system architecture, to properly demonstrate the method it is helpful to follow all the steps explained in Chapter 4. The process would normally begin with the function modelling phase. Thus, all aircraft functions should be listed and further refined into subfunctions. However, this case study would be too extensive and would elude its demonstrating purpose. Hence, the focus is given to the base brake system. Fig. 6.6 shows some of the high-level functions of Boeing 787. In the figure, there is use of the term ‘ground deceleration’ as this is the common expression in aviation for the brake system. The diagram below is presented in the simplest way possible. In reality, although the spoilers and reverse thrust as subfunctions contribute towards the ground deceleration function, they are controlled via the flight control and engine control systems, respectively. Thus, they incorporate their own set of subsystems such as power supplies, actuators and controllers. Finally, feedback from sensors within each of the ground deceleration subfunctions may also be directed to other systems as well.



*Figure 6.6 High-level Aircraft Functions*

## 6.6 Hazard Identification and Analysis

This step of the method involves the identification of hazards, criticality and functional DAL for each of the functions. The results of the FHA are shown below in Tables 6.1 and 6.2.

Table 6.1 Ground Deceleration FHA page 1

<b>FHA Name</b>	Function FHA Boeing 787	<b>Aircraft Phase</b>	Landing & Rejected Take-off
<b>Function Name</b>	Ground Deceleration	<b>Hazard Classification</b>	Catastrophic
<b>Hazard Name</b>	Loss of Deceleration Capability	<b>DAL</b>	A
<b>Hazard Description</b>	Unanticipated loss of deceleration capability	<b>System Requirements</b>	Likelihood of occurrence less than 1E-9 per flight hour
<b>Hazard Effect</b>	Pilots unable to stop the aircraft	<b>Verification Method</b>	Aircraft Fault Tree

Table 6.2 Ground Deceleration FHA page 2

<b>FHA Name</b>	Function FHA Boeing 787	<b>Aircraft Phase</b>	Landing & Rejected Take-off
<b>Function Name</b>	Ground Deceleration	<b>Hazard Classification</b>	Major
<b>Hazard Name</b>	Loss of Autobrake	<b>DAL</b>	C
<b>Hazard Description</b>	Unanticipated loss of automatic brake	<b>System Requirements</b>	-
<b>Hazard Effect</b>	Automatic system fails to operate, and pilots switch to manual mode. Slow reaction time might lead to overrun and minor injuries	<b>Verification Method</b>	Software and Hardware Testing

For simplicity purposes, the case study examines only two functional failures for the ground deceleration aircraft-level function. The first table refers to the loss of any deceleration capability,

where a high-speed overrun with multiple injuries seems unavoidable; hence, the hazard is classified as catastrophic. The other example is about the autobrake function, and as seen in the second table, the impact and therefore the classification, is not as great. The reason behind this is that experienced pilots, with appropriate warning signal, are likely to react in time. Even in the scenario that switching to manual control is slow, a potentially medium-speed overrun is deemed unlikely to cause fatal or major injuries to the crew or passengers.

## 6.7 Modelling of System Architecture

Once the functions are modelled, the process continues with the design of the underlying system architecture that supports these functions. Normally, the ground deceleration function is directly supported by three systems; however, due to space limitations, only the wheel brake is presented here.

Fig 6.7 illustrates in a diagrammatic view its system and subsystems.

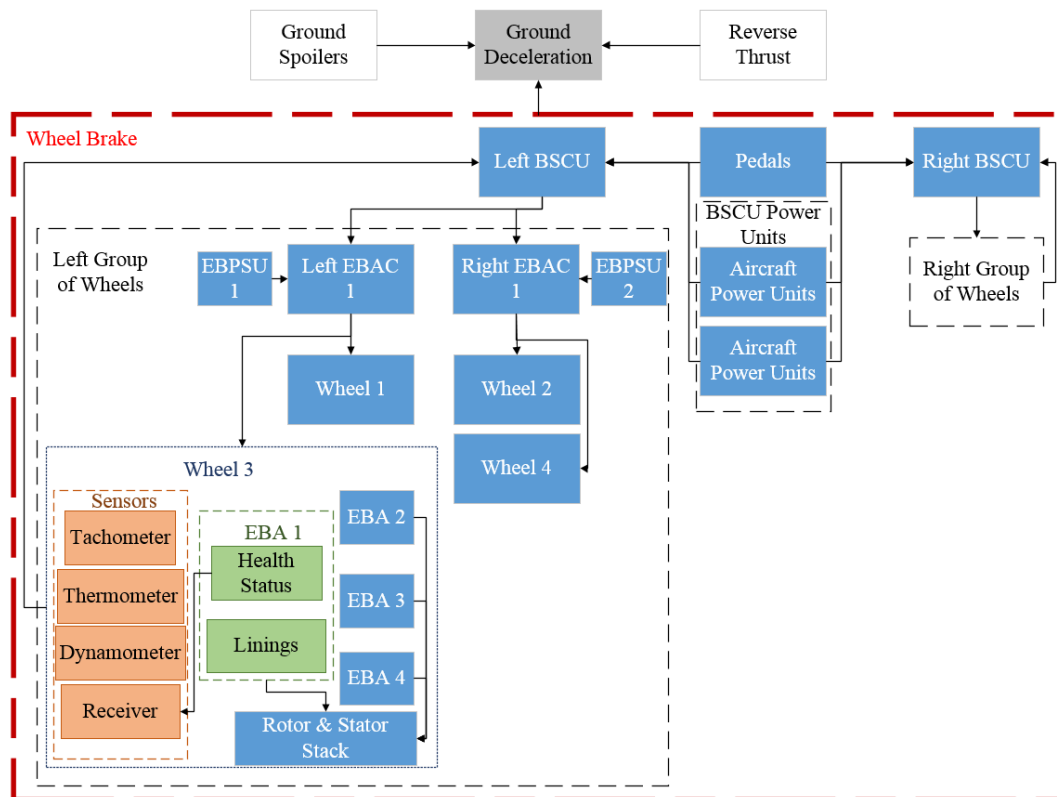


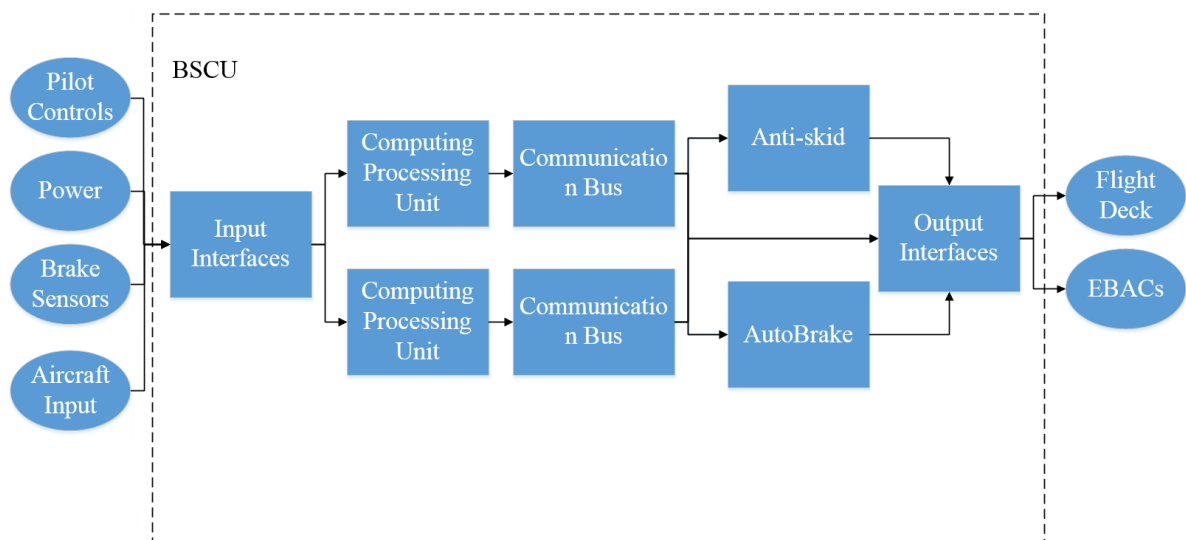
Figure 6.7 Wheel Brake System Model

The diagram above begins by depicting the aircraft-level function ‘Ground Deceleration’ as the grey-coloured rectangle and its supporting systems, ground spoiler, reverse thrust and wheel brakes. The architecture of the latter is highlighted with red lines. The model provided here is essentially a more detailed description of the original architecture presented in Section 6.4 of this chapter. The pedals send input to the BSCUs, which calculate the appropriate command and send the signal to the EBACs. For clarity and due to space restrictions, only the ‘Left BSCU’ is further shown in Fig. 6.7. Although omitted here, the ‘Right BSCU’ features the exact same underlying architecture, but the commands are directed towards the right group of wheels. Both BSCUs are powered via two individual power units. Note that related information about the 787’s real solution with regards to the power units was limited. Therefore, to compensate, it is assumed that one power unit provides power to both, and in case of failure the other one takes over for redundancy purposes. The wheel assemblies receive input from the EBACs, as discussed earlier. ‘Wheel 3’ is further refined into its components; other wheel assemblies are supported similarly by their separate subsystems. These subsystems include:

- A series of sensors
  - Thermometer, for temperature indications to the cockpit and BSCU subsystems
  - Tachometer, for measuring rotation rate (useful for anti-skid)
  - Dynamometer, for determining overall pressure
  - Receiver, for receiving warning signals for lining wear
- 4 Electric Actuators (EBAs)
  - Linings, mechanical component
  - Health status indicator and individual brake force monitoring
- The mechanical part of the brake, which is the stack of rotors and stators

Each wheel constantly sends feedback, during take-off, before landing and during taxiing, to its respective BSCU. This information is either used internally for the BSCU’s subsystems or is sent to the flight deck to provide status update to the pilots. Information about the technical details, regarding the BSCU, were limited. Therefore, Fig. 6.8 showcases a fictitious model.





*Figure 6.8 Abstract BSCU model in detail*

The BSCU is modelled to have various input and output interfaces and a number of subsystems, as follows:

- Inputs
  - Pilot Controls, input via pedals, levers and switches (e.g. for brake control or autobrake activation)
  - Power, electricity from the BSCU power supplies via wires
  - Brake sensors, the warnings and signals received from the wheel assembly
  - Aircraft input, values and properties, such as aircraft speed (useful for anti-skid)
- Outputs
  - EBACs, brake commands to the brake actuators based on BSCU calculations
  - Flight deck, warnings and indications to the cockpit for informing the crew
- CPUs, two CPUs with their own separate software and communication buses, for redundancy purposes
- Anti-skid & Autobrake, responsible for the functions discussed in Section 6.3. There was no clear information, whether or not they are solely software or incorporate their own hardware

Finally, the BSUs typically include additional subsystems, such as the parking brakes. The Boeing 787 implements constant brake monitoring even when parked. The main reason is to automatically adjust the parking brakes, based on brakes' temperatures, and achieve improved power efficiency.

## 6.8 Failure Analysis and Annotation

The next step in the method is to identify the failure behaviour based on the description of the system. Once this information is clear, the system architecture can be modelled within our software tool based on the metamodel explained in Chapter 5. Then, all individual functions and systems can be annotated with that failure behaviour information along with system-level failures (i.e. hazards). The hazards used in this case are the ones already described in the FHA, in Section 6.6.

The annotation process is managed through defining the failure logic on each of the subsystems. However, it is easier to convey the behaviour, for each of the hazards, using fault trees. Fig. 6.9 demonstrates the failure behaviour of the aircraft as a whole.

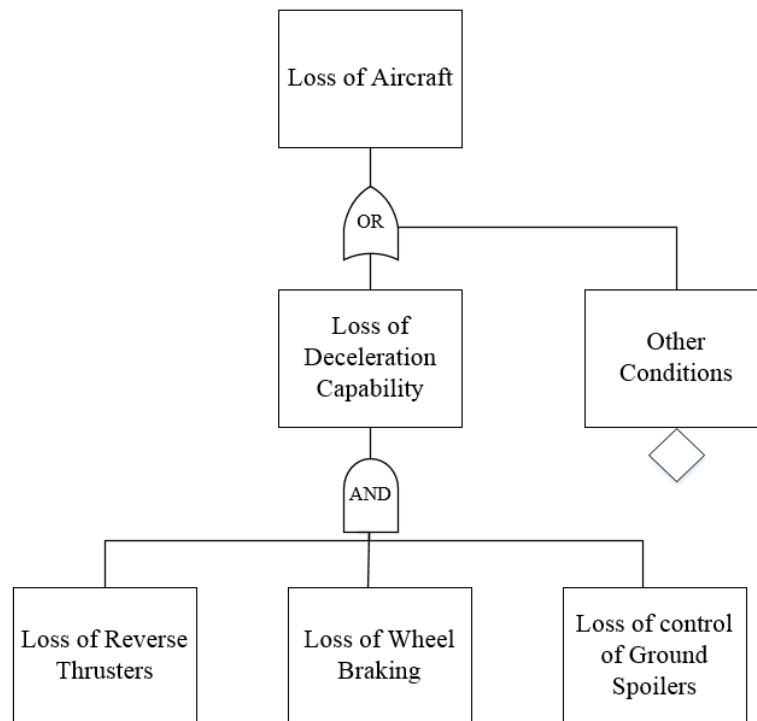


Figure 6.9 Overall aircraft failure behaviour

The hazard for the top-level system, the aircraft, would be ‘loss of aircraft’. There are many different conditions that could potentially result in such a scenario. However, in this diagram only the loss of deceleration capability is considered, as the rest of conditions remain undeveloped. The ‘Loss of Deceleration Capability’ as a hazard has many different facets. For example, the anticipated loss of deceleration capability or the inadvertent deceleration after an attempt for landing or a rejected take-off. In any of these scenarios, the combination of failures of multiple systems, responsible for braking, would need to be considered. However, despite the reverse thrusters’ ability to quickly decelerate the aircraft during landing, they are not always required. In fact, in many cases the aircraft should be able to land safely even with inoperative thrust reversers as specified in Master Minimum Equipment Lists (MMELs) provided by regulatory bodies such as the FAA and International Civil Aviation Organisation (ICAO, 2010:ATTC-31). Note that reverse thrusters are not to be confused with redundancy braking mechanisms as they have more of a supportive role and regulations state that, under normal conditions, the aircraft should still be able to decelerate even without using all of its braking capabilities. Therefore, in the following tree, only the loss of wheel braking is considered for one of the possible scenarios of ‘Loss of Deceleration Capability’. Fig. 6.10 illustrates the hazard examined earlier during FHA (table 6.1).

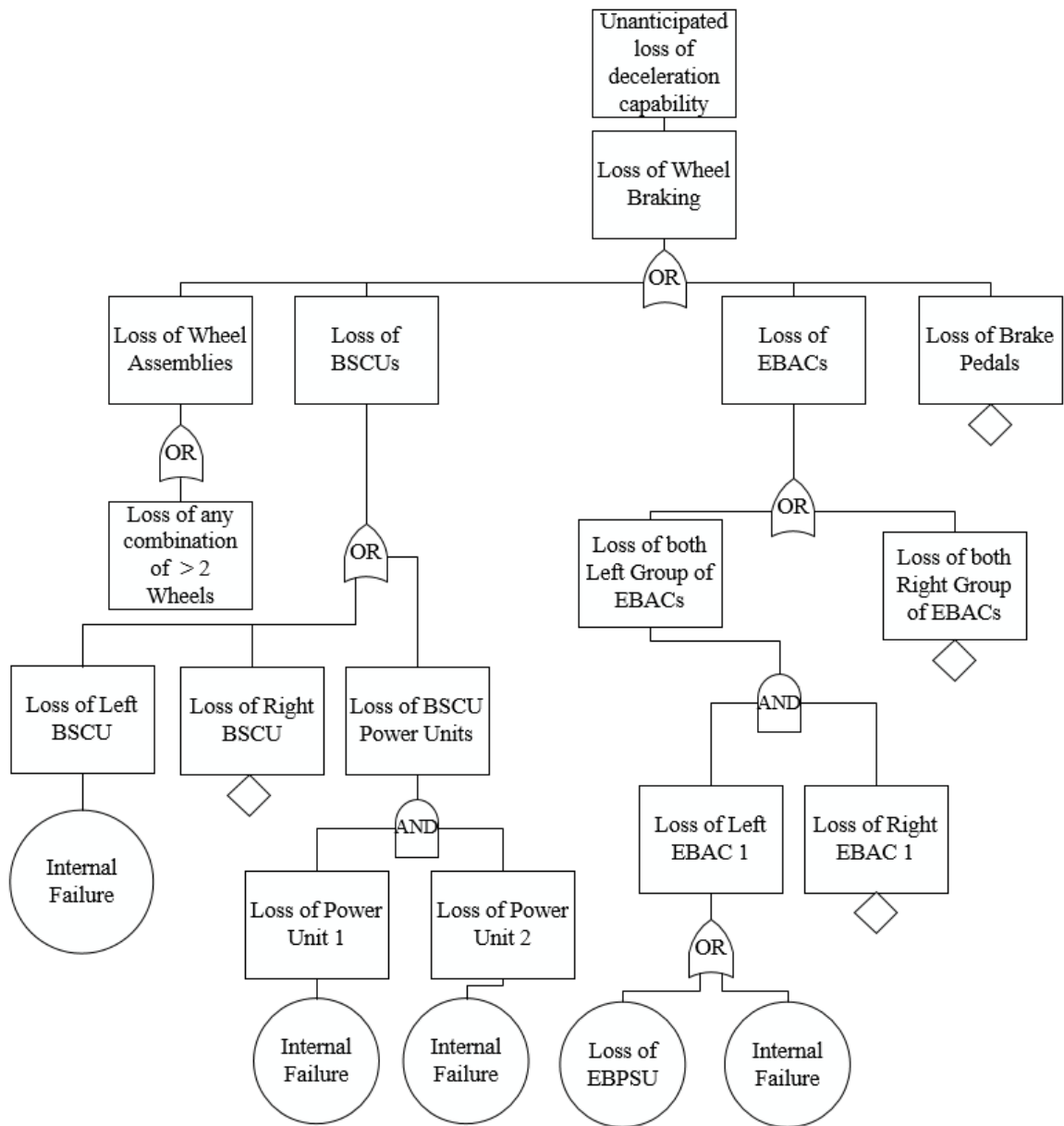


Figure 6.10 Analysis Loss of Wheel Brake System

One of the failure conditions that could be classified as ‘Loss of Deceleration Capability’ is the ‘Unanticipated loss of deceleration capability’. Most of the elements that support the ground deceleration function and its supporting system, the wheel brake system, are assessed. The ‘Loss of Wheel Braking’ failure may happen under any of the following circumstances:

- Loss of Brake Pedals
  - This can happen only if both the pilot’s and co-pilot’s pedals fail

- Loss of any groups of EBACs
  - Loss of both EBACs, either via loss of power or internal failure (e.g. SW or HW)
- Loss of BSCUs
  - Any internal BSCU failure (internal failures of CPUs or SW systems are simplified)
  - Both of BSCUs' power supplies fail
- Loss of Wheel-Brake Assemblies
  - Any combination of 3 or more assemblies (e.g. Wh1 & Wh3 & Wh8 from Fig. 6.7)

These are the major possibilities, although some of the details have been omitted. For example, the wheel assembly might fail due to more than one of the actuators (EBAs) failing to operate. In addition, the pilot's and co-pilot's pedals failure has been simplified. Theoretically, if the left and right pedals from the pilot and co-pilot (or vice versa) respectively are functional, under absolute coordination between the two, successful braking might still be possible. Finally, internal failures for computer-based components might be due to software or hardware failures, communicate bus failures or even damaged wiring. With all the information provided, it is possible to start the automatic analysis, using the integrated HiP-HOPS engine.

## 6.9 DAL Allocation and Decomposition

The results produced with the previous step are the fault trees, for every top-level system failure, as well as the FMEA tables. More importantly, the minimal cut sets are acquired and later used by the algorithm, discussed in Chapter 3, for the automatic allocation of DALs. The cost parameter used for each DAL are the same shown in Table 3.8. In practice, the engineers would have calculated the cost for every level of stringency per system configuration. Finally, it should be noted that the allocation of the overall system is derived after accounting for all potential hazards. As the guidelines recommend, the most stringent requirement per component overrules lower values. Since the hazard in Fig. 6.10 was identified as catastrophic during FHA, the top-level system is assigned with DAL A. The other hazard was classified as Major, meaning that DAL C would be sufficient. The rest of the values from one of the optimal allocations are shown in Fig 6.11 below.

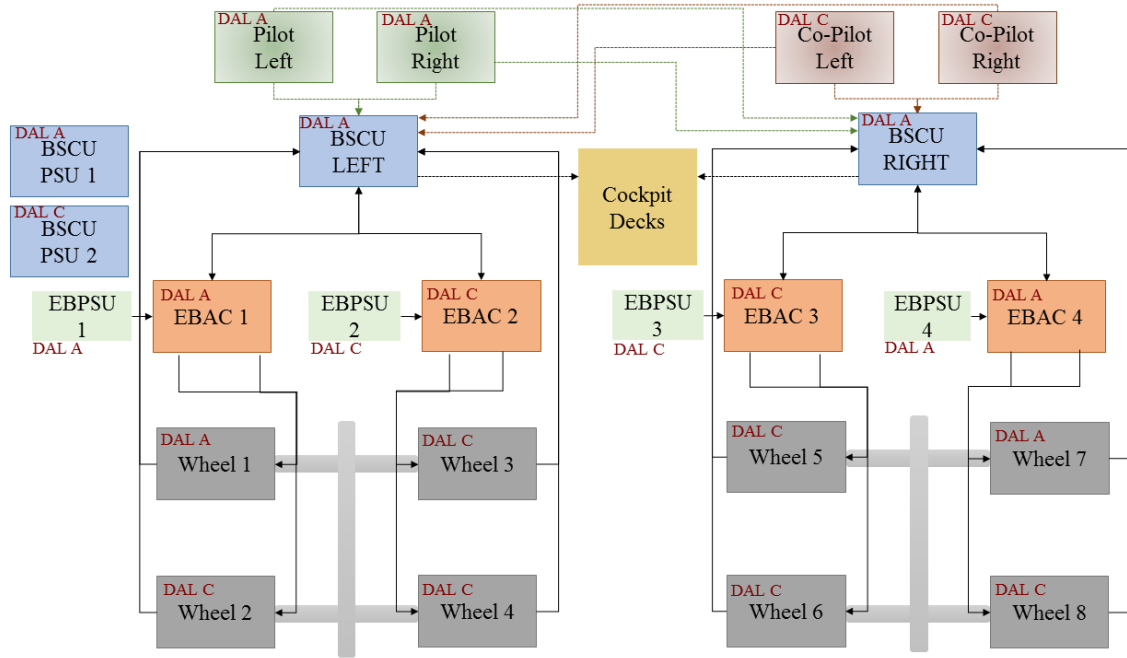


Figure 6.11 DAL allocation for Wheel Brake

The power supplies support both BSCUs and there is redundancy, so the decomposition rules apply. The allocation on the wheel assemblies is mainly for the electronic parts, as mechanical parts use a different method for requirements; although for the purpose of a simplified example, even the stack of rotors could be considered as an electronic function. Finally, there were many allocations with similar total cost due to the absence of individualised costs, but only one presented here.

## 6.10 Pattern Preparation

With all the previous information and the DALs already allocated, the software only needs an argument pattern to automatically produce an argument of safety. Following the general guidelines of ARP4754-A and the context provided in Chapters 3 and 4, it should be expected that a pattern covering all these aspects would result in numerous sets of large diagrams. Therefore, only certain parts are presented in this section. Fig. 6.12 presents an overview of the pattern.

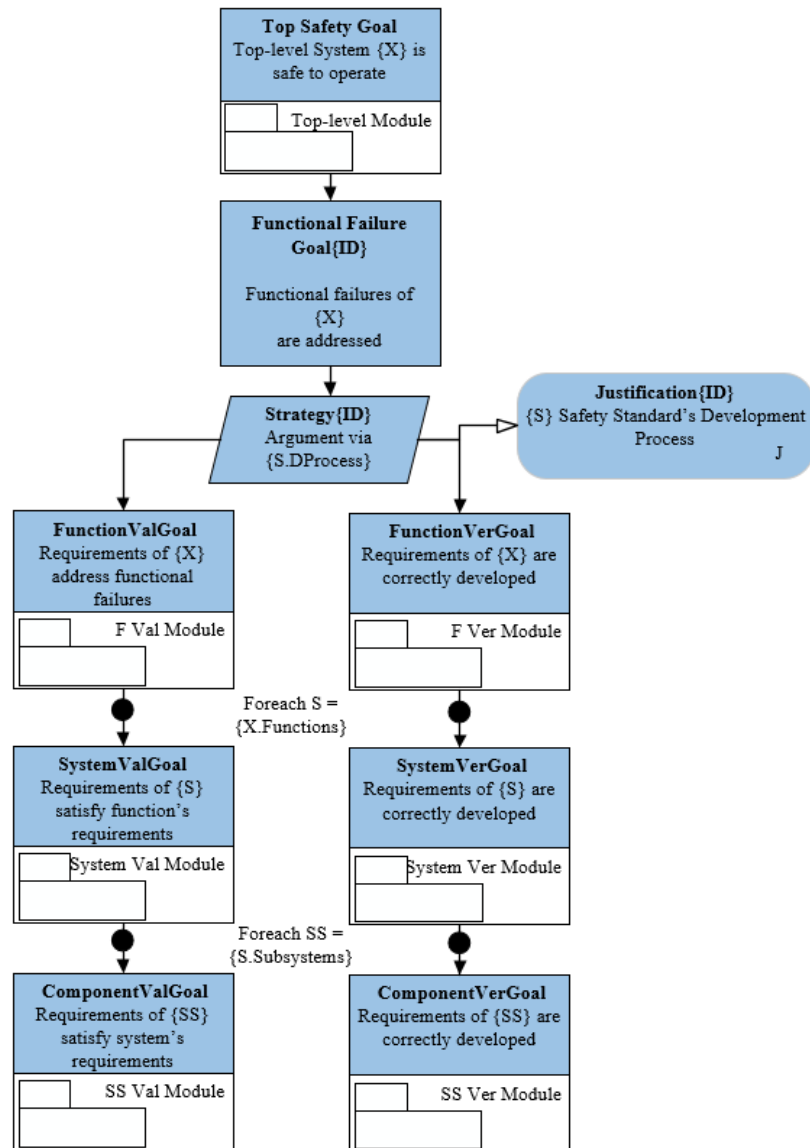
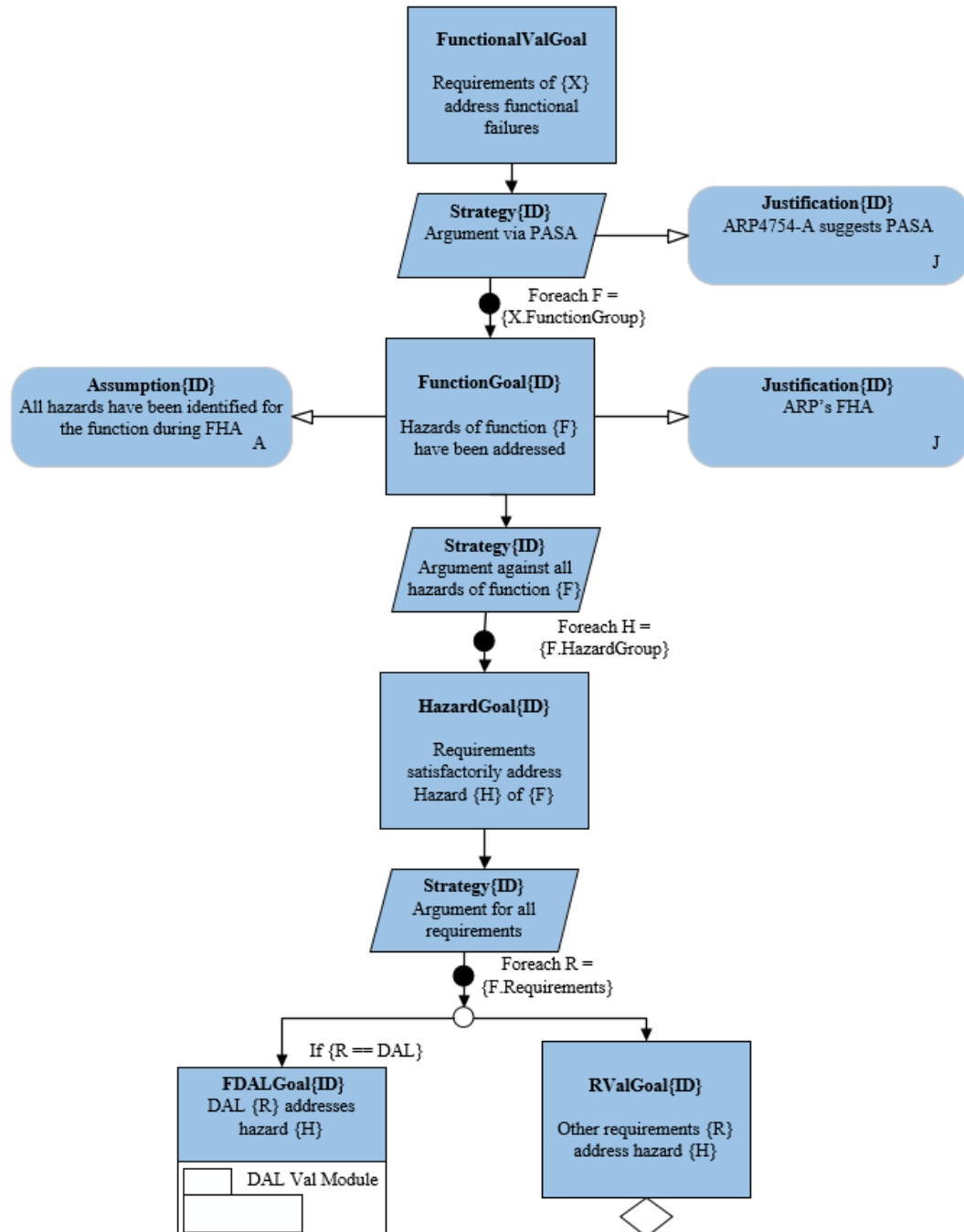


Figure 6.12 Pattern Overview

The argument is split into various subgraphs encapsulated in modules. The top module argues that the top-level system (i.e. the aircraft) is safe since the development followed the corresponding safety standard's guidelines. The node below supports the top-level goal by establishing that all functional failures, identified during design, have been addressed. To properly accomplish this task, all functions need to be validated and verified by following the 'V-model' lifecycle. Hence, functional requirements need to be validated for correctness, during the design stage, and verified for correct development, during the implementation phase. This process has to be repeated, and therefore argued,

across the entire architecture. This leads to further claims about the systems which support functions, and subsystems or items that support their parent systems in response. Fig. 6.13 shows a more detailed claim for function validation.



*Figure 6.13 Function Validation Claim*



The top-node of this subargument claims that aircraft functional requirements address all the underlying hazards, assuming a perfect identification during the most recent FHA. The process is justified by the guidelines, whereas the claim is further supported by traversing throughout all hazards of all the functions of the aircraft. For each hazard per function, a claim is made that the function requirement, whether DAL or not, satisfactorily addresses the hazard. If the requirement is a safety integrity level (DAL), then the claim is supported by the ‘DAL Val Module’ sub-argument. In any other case, the argument would be supported by the ‘RValGoal’ node, which is left undeveloped in this case study. Fig. 6.14 illustrates the ‘DAL Val Module’ in a mixed argument that contains nodes and modules for simplicity.

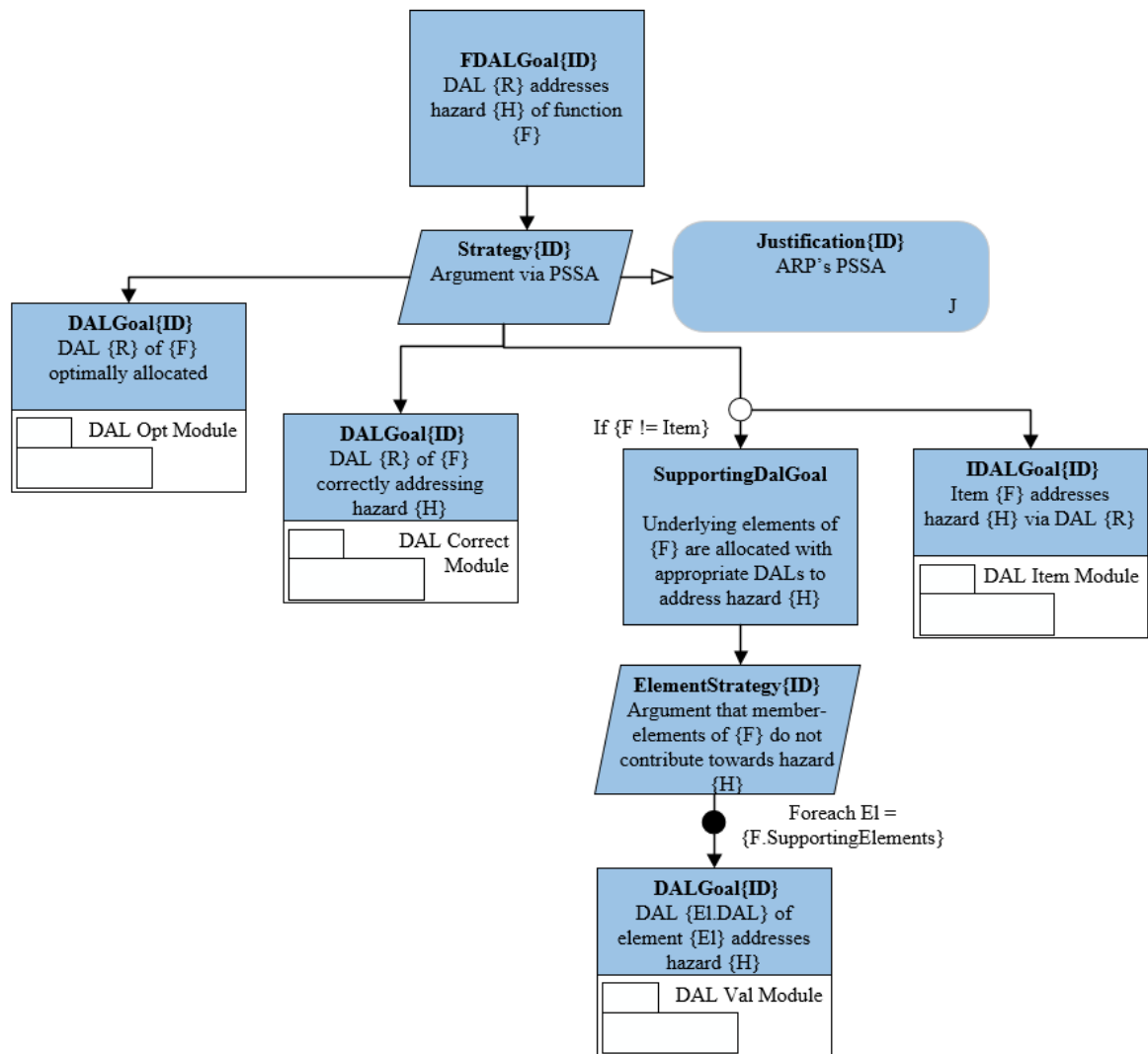


Figure 6.14 DAL Validation Argument Overview

The top node on this argument claims that the allocated DAL of a system element is suitable for the hazard of its parent-node. The PSSA is utilised, as described in the ARP4754-A, to examine causes of failure that might result in the hazard occurring. The supporting nodes argue that the DAL is correctly allocated and optimal for the current architecture. The ‘DAL Opt Module’ is responsible for arguing the optimality, based on evidence from the optimisation algorithms (e.g. Tabu search). On the other hand, the ‘DAL Correct Module’ sub-argument provides further claims and evidence that the DAL is correct. To do so, it utilises the FFSs where the element appears for the current hazard and argues about its correctness. Then, it separates the elements based on the strategy selected during the optimisation process and creates the specific claims. This is achieved by iterating over the FFS members until everything is sorted. The ‘SupportingGoal’ is responsible for dividing the elements into items (i.e. no further subsystems) and systems (or functions). If the latter is true, then it uses the same module, from Fig. 6.14, in a recursive manner to allow the pattern to look deeper into the architecture.

Finally, the verification part of the argument modules in Fig. 6.12 works in a similar fashion except for the application of ASA and SSA, instead of the PASA and PSSA (featured in the validation). Moreover, it uses every system element as input to examine if the implementation respects the functional requirements.

## 6.11 Argument Structure

The final stage in the method is to call the instantiation algorithm, with the results from previous steps as input, and the safety argument will be created and presented automatically. The argument is considerably larger in size than the pattern. Hence, only two fragments are demonstrated and discussed in this section. Fig. 6.15 depicts the argument for the top-level module, earlier seen in Fig. 6.12. The reason is that the details of this sub-argument structure were skipped during the pattern description. Moreover, it is important as this part of the overall argument, claims that the top-level system, in this case Boeing 787, is safe to operate by following the corresponding guidelines during design and development.

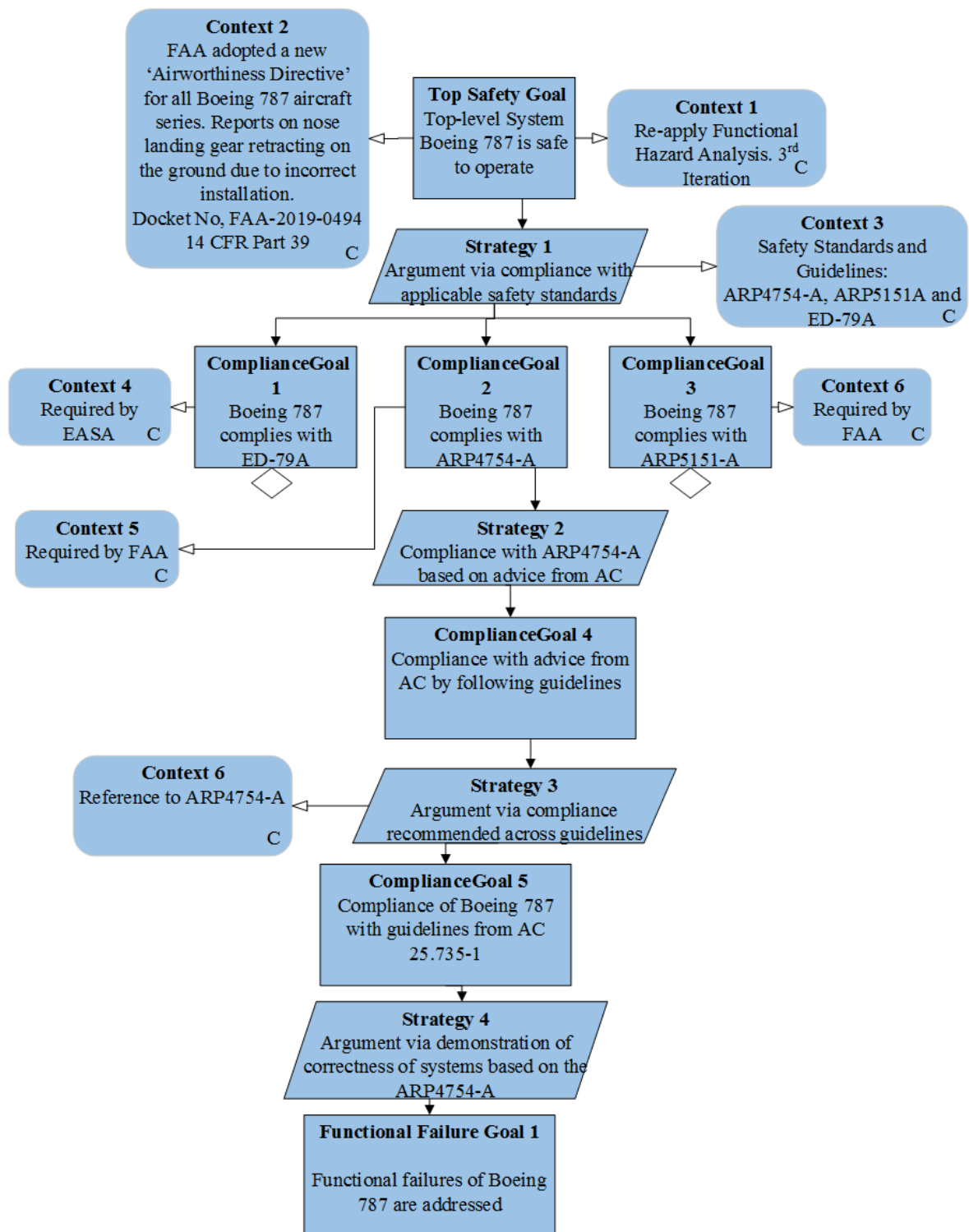


Figure 6.15 Top-level Argument - Aircraft safety due to compliance with safety standards

Note that airline companies operate this type of aircraft all over the world, so compliance with EASA, for operation within European boundaries, is mandatory. However, the guidelines are similar to the FAA's and to simplify the argument, other standards remained as undeveloped nodes.

The argument presented in Fig. 6.16 below establishes that the DALs allocated to the left BSCU and its supporting systems are sufficient for addressing the hazard (i.e. Unanticipated Loss of Deceleration Capability). Normally, this would involve its internal architecture, from CPUs and communication buses to software. Instead, the architecture was limited to the underlying systems, EBACs and power units; according to the model shown in diagram of Section 6.9.

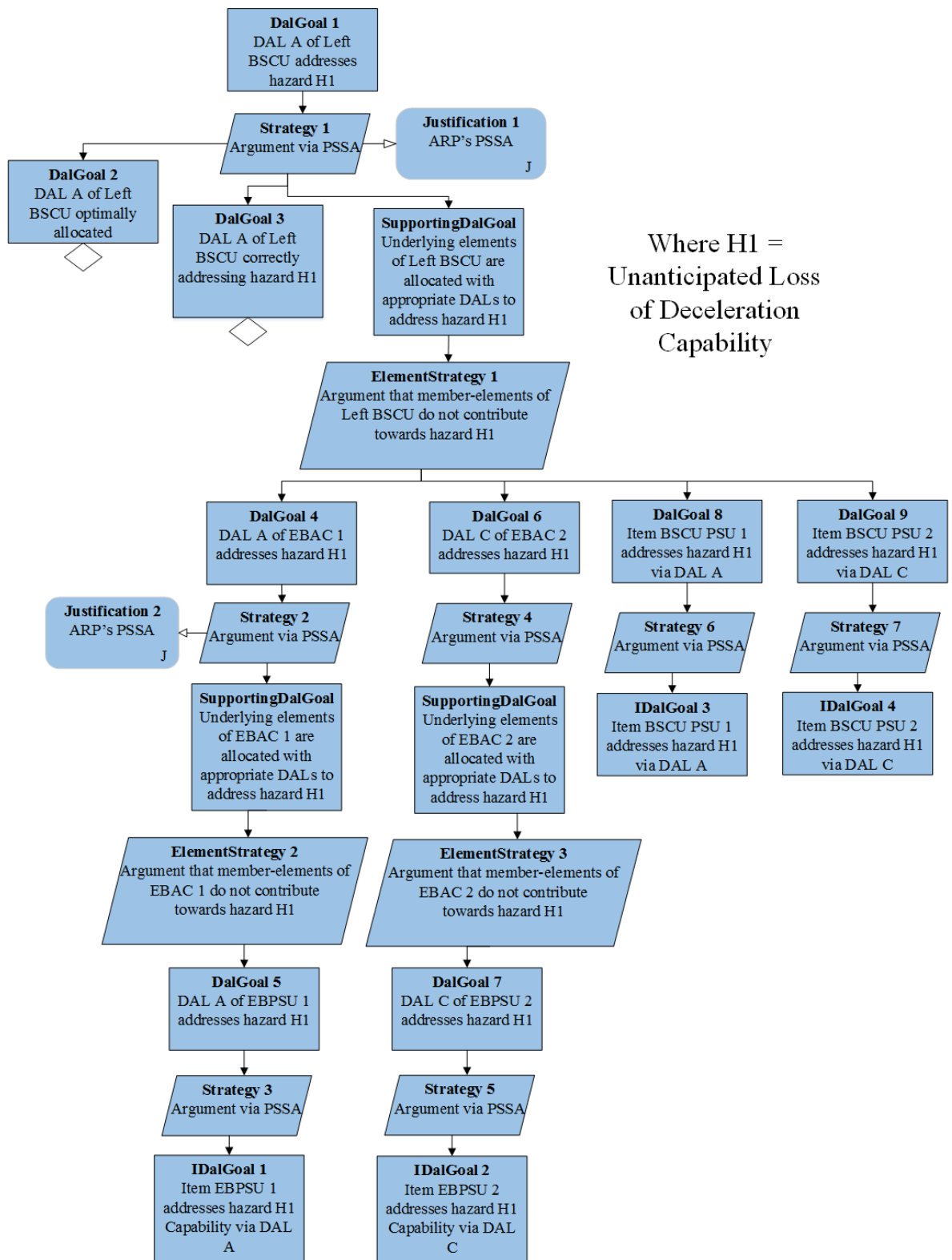


Figure 6.16 Left BSCU's supporting systems

## 6.12 Handling Modification

Following an evolutionary development lifecycle, changes are bound to happen. The safety standards prescribe that design, implementation, validation and verification should be performed in an iterative manner. This philosophy is also shared among safety engineers; therefore, safety processes should also follow this scheme. In this case study, it was demonstrated that even by just analysing only a portion of one system for two hazards and presenting excerpts of the results, composing the pattern and arguments can still be a cumbersome task. Even today, a large portion of the processes involved in the certification are handled manually or with ad-hoc software. The method and framework presented in this thesis, however, utilise algorithms to automate many of the steps towards safety assurance. The next step is to review how the approach handles change.

The most common type of changes is the alteration of the system design. During the design phase, engineers might conclude that a certain system or component does not serve its intended purpose well or maybe a component's suppliers changed the price dramatically and the alternatives have different specifications or internal architecture. In aviation, even when the aircraft is in active use, the FAA (or other overseeing bodies), might request a hardware or software alternation through directives. This often happens when various reports reach the aviation administrations and it is determined that the aircraft might violate certain regulations. For example, through an Airworthiness Directive (AD) in 2019, FAA requested that Boeing 787-8 and 787-9 should install a hydraulic tubing, a check valve and new flight control software (FAA, 2019).

Earlier it was stated that Boeing 787 is safe to operate with only six functional brake assemblies, and one dysfunctional EBAC would not have a major safety impact on the braking of the aircraft. However, if the brake assemblies were to be replaced with less effective alternatives for reducing cost, this probably would not have been the case anymore. Assuming that even one EBAC malfunction could be catastrophic, the system needs to be re-evaluated. That means that the practitioner, using our method, has only to re-annotate the system model as there were no actual changes in the aircraft's structure. The rest are handled by the automatic analysis and instantiation of

the tool. Fig. 6.17 illustrates the changes on the argument fragment, earlier shown in Fig. 6.16. The changes in the argument are highlighted in red colour. As expected, the EBAC 2 and its PSU are both assigned DAL A, since each of them failing could cause a cascade of failures leading to the hazard H1 of the ‘Ground Deceleration’ function.

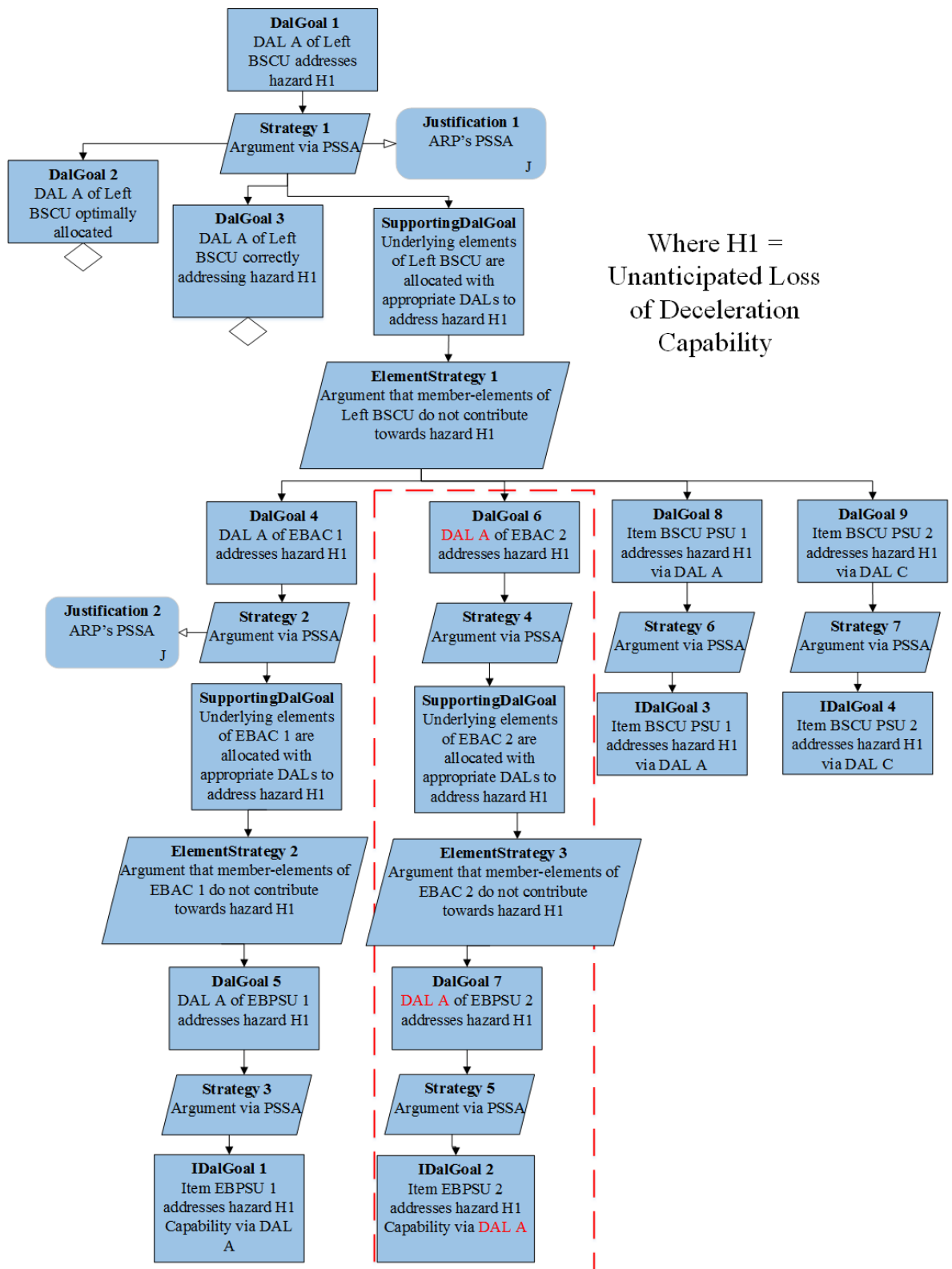


Figure 6.17 Altered Argument



Another type of change could happen via modifications to one of the guidelines or due to a customisation of the method for application in other domains. For example, the ARP4754-A might alter the ‘V-model’ lifecycle and add one more stage in between the validation and verification. Assume a scenario where the ARP incorporates a ‘Data-Driven Simulation’. In this case, arguing safety would not be sufficient with the previous argument structure. However, the practitioner could alter the pattern and future arguments would still be compliant. Fig. 6.18 demonstrates how the overview of the pattern would change. The new nodes are placed within a red container for clarity.

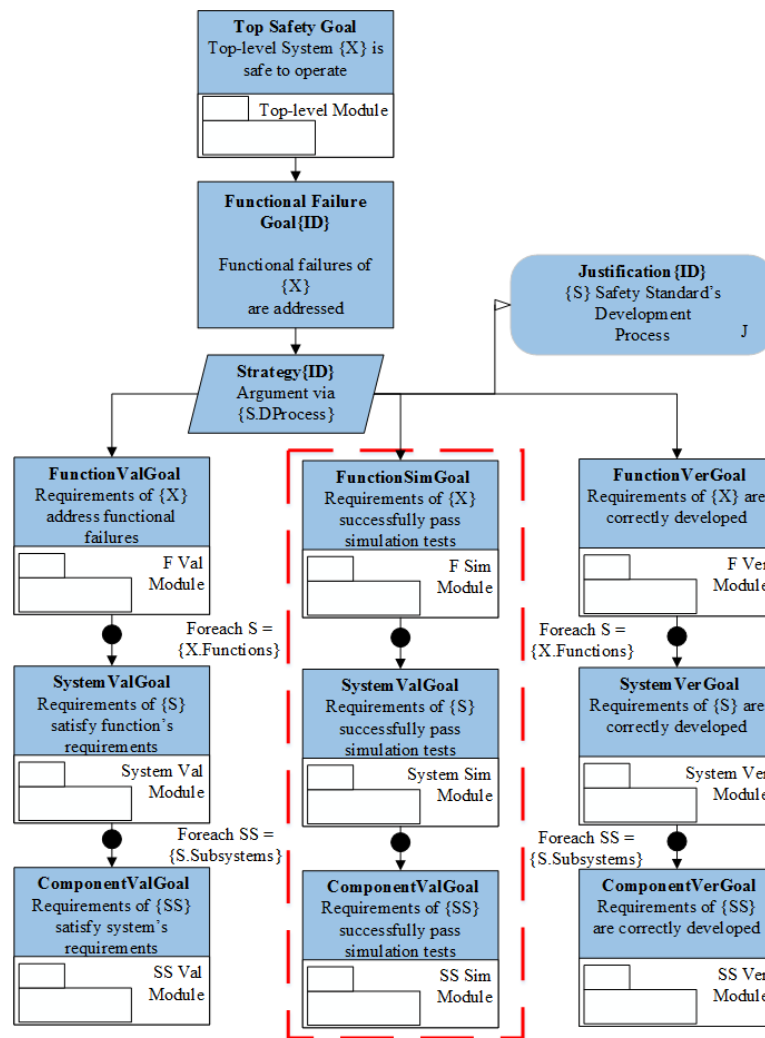


Figure 6.18 Altered Pattern Overview

There can be other forms of changes from the guidelines that require a slightly different approach. For example, currently the ARP4761 recommended practice guide suggests FTA or Markov Analysis

as viable techniques for quantitative analysis of failure conditions during SSA or PSSA (ARP4761, 15-16). Thus, the analysis part of the method incorporates FTA. However, if a new version of the guidelines find that these techniques are inadequate, the method would produce arguments with weak evidence according to the guidelines. In such a scenario, the practitioner can either use results from other tools and add the results as evidence in an ad-hoc manner or implement these new techniques in the tool. Unfortunately, this would not be enough as the pattern would still require changes to update the argument structure and various context or justification elements about the validity of the results (e.g. DALs). However, such drastic changes would rarely happen in a short amount of time, because they would require multiple reports from different sources to support them.

### 6.13 Method Evaluation and Framework Testing

Evaluating the method presented in this thesis would essentially mean to inspect all its individual steps and results for the following aspects:

- Correctness of the safety argument
- Cross-sector applicability
- Benefits against current approaches

One meaning for “correct” safety arguments is whether or not they are comprehensive and convincing. For example, providing pure evidence from analyses with no proper wording, or rationale, is certainly not sufficient. Various groups of experts and regulatory authorities as well as other stakeholders are responsible for examining a safety case, either for certification or consultancy purposes. All these parties are well aware of the safety assessment processes and the development lifecycle. Therefore, by following the rationale described in safety standards, it becomes easier to convey the argument of safety. The method of this thesis, as shown in this chapter, follows the logic provided in ARP4754-A to build a safety argument for a commercial aircraft. This ranges from incorporating functional design and recommended system analysis techniques, such as FTA, to safety processes, such as PSSA. However, when composing a safety case, either traditionally or with

notations, it is important to maintain cohesion and allow for a smooth transition between claims. To accomplish this, one must be careful about preserving the following characteristics:

- **Appropriateness**, so that the most suitable child-nodes support a claim; no more, no less. By discarding unnecessary elements, this trait helps maintain clarity in the argument.
- **Association**, so that all supporting claims are directly related to their parent-claim.

By reviewing the examples presented in this thesis, to the best of our knowledge, these features have been maintained to a reasonable extent. Appropriateness has been maintained throughout the argument, as there are no apparent instances of unnecessary redundancy, neither in the form of repetitive claims and strategies nor as multiple instances of an architectural element found in different levels. Association is also conserved by following the system architecture. Most of the claims are supported directly by subclaims generated through the system model and in proportions that the guidelines encourage through the safety processes. That being said, the resulting arguments are not error-proof since most of the information about the aircraft and its architecture are informal. Naturally, most of the general information come from official sources, but not in the degree of detail that would allow for a safety case. An industrial case study would be more suitable for such an investigation.

The cross-sector applicability is yet another difficult criterion for evaluating our method. Throughout the thesis, it is mentioned that our approach can be applied in various industries. After discussion in earlier chapters, and most notably in Chapter 3, it is apparent that this is a plausible argument. However, to prove that it is feasible, it would require all the aforementioned safety standards, meticulous research and probably further consultation from industry experts. Unfortunately, safety standards are not free of charge; in fact, they are quite expensive and, luckily, due to previous possession of the specific ARP and ISO standards by University colleagues, their detailed examination in this thesis was possible. Moreover, the amount of time thoroughly researching all this material and attempting to communicate with experts was beyond the scope of this thesis. Finally,

even with all the above covered, there would still be vulnerable spots due to the lack of details an industrial case study would offer.

The remaining evaluation means, ‘benefits against current approaches’, is more manageable. Comparing the arguments from the example in Chapter 4 and the case study, it is obvious that the argument in the case study is considerably larger. The number of system elements between the two architectures (i.e. simple example from Chapter 4 and case study) is substantial. In addition, the argument elements generated seem to not share a proportional factor. In fact, it is obvious from the results that the argument for the case study is in the thousands of argument nodes whereas the other example has only a page-long structure. This leads us to the need for a scalable solution. Even today, many projects handle model information and analysis with ad-hoc software solutions and maintain safety arguments manually. This results in great costs in terms of time and effort. The method presented in this thesis, is still tied with manual work in creating the model, identification and classification of hazards, annotating the model with failure behaviour and finally creating the argument strategy via the pattern. However, the analysis, requirement allocation, argument generation, and maintenance are all handled automatically. This means that even if not all the process is automated, we still have an advantage over manual approaches in time. For a realistic example (i.e. industrial case study), we can presume that the complexity of the systems architecture and the number of elements involved will be vast. Hence, the benefit from automation will be even greater. Naturally, a benefit in scalability of the implemented solution is dependent on the actual software engineering framework. To this end, the thesis discussed some of the challenges and provided a blueprint that materialises that framework as well as a software tool solution. As an example of how scalable that framework is, the examples from Chapters 4 and 6 were tested in that tool and the results are compared in Table 6.3.

Table 6.3 Framework Execution Time

Example in:	Chapter 4	Chapter 6
<b>Model Elements</b>	5	152
<b>Pattern Elements</b>	10	137
<b>Argument Nodes</b>	21	93,404
<b>Avg. Execution Time</b>	1.3secs	357-381secs

During this testing, the processes involved in measuring the execution time were: a) the analysis process via HiP-HOPS, b) the optimal allocation of DALs (including registering them to the appropriate element), c) the instantiation process, and d) the exporting of the argument structure in a file for storing. The code used, and related information, are provided in Appendix II. The metrics shown in the table above showcase that regardless of the significant scaling of argument nodes, the average execution time of the implementation holds well. The latter is considered to be reasonable time only when comparing with manual methods that are typically employed based on the literature. This allows for an easier evaluation in terms of practical time during the design and development of a safety-critical system.

Another benefit of the approach is the reduction of errors produced with the human factor. Certainly, when dealing with a simple model, errors are limited and can easily be identified. However, locating errors within such immense arguments is close to impossible. Automation helps alleviate these errors. Arguably, software often is not error-free, but there are various testing processes and formal methods that can be applied on software tools that can offer rigorous evaluation and reduce or eliminate these errors. The software tool presented in this thesis has not been tested with formal methods or any other advanced testing techniques, however this was out of the scope of this thesis. The method presented does have a sound foundation that relies on assessment and assurance tasks advocated by the standards. The supporting tool developed was meant to be only an auxiliary mechanism to test the automatic parts of the method and help run examples to examine the generated argument structure.

Last but not least, another benefit compared to current (and mostly manual) approaches, is maintainability. Due to the iterative nature of modern development, decisions, assumptions and knowledge are constantly changing. The goal of safety engineers is to keep up with all these changes, re-assess the situation and maintain acceptable levels of safety whilst able to provide convincing arguments. Earlier in this chapter, it was discussed how the method handles changes. Architectural and knowledge-based changes require some effort either from the safety engineer, or rarely from a systems developer, but it can still be considered negligible compared to manual approaches or ad-hoc solutions.

## 6.14 Summary

This chapter demonstrated the method via a more detailed case study. Initially, information about the system in question was reviewed, to understand the various technologies of brakes and how they operate. Then, more information about a modern commercial aircraft was given, to help comprehend how that specific design works. The method was applied on the case study's system and its resulting safety arguments were evaluated with respect to their quality against specified criteria. Finally, the chapter concludes with a discussion on some metrics taken related to the execution time and explains their meaning.

## Chapter 7 Conclusions

---

### 7.1 Synopsis

In multiple occasions, in Chapters 1 and 4, it is mentioned that the evolutionary and fast-paced nature of modern development, for safety-critical systems, requires further attention. Dealing with systems whose failure could cause a catastrophic event requires continuous evaluation and assurance. Modern safety standards support this notion and have provided guidelines that help engineers prove compliance. However, following these guidelines does not directly translate to certification, as it is the developers' responsibility to assess their products and then convince the authorities. Safety cases serve this purpose, to communicate clear and convincing arguments that all related activities and safety processes have been performed according to the guidelines. Thus, explicitly expressing that the system, in its current state, is acceptably safe. To accomplish this task, safety assessment often begins during the early stages of design and development. Unfortunately, the increased integration and convoluted architecture of modern systems along with the complication of presenting coherent and satisfactory arguments makes this process arduous. Considering a few iterative cycles of design, validation and verification, it is logical to assume that errors and postponements are likely to happen.

There have been advances to ease the process of assessment activities and argument structure, via ad-hoc automation and graphical notations, respectively. These solutions are undeniably helpful, but a more systematic approach is required, as even with notations the safety argument can grow large in size (as seen in Table 6.3 of the previous chapter). In (Yamamoto & Matsuno, 2013) and (Sun et al., 2014), the authors explain the difficulties in producing assurance and safety arguments. The reasons vary from the multiple interrelations between system elements that need to be reflected on the argument to the confusion with regards to the notation syntax. For example, many inexperienced practitioners find the elaboration of strategy nodes to be difficult. As a result, creating an uncluttered argument structure, for a system of a considerable size, while preserving correctness and claim cohesion is still a challenge. In the aforementioned publications, it is stated that the emergence and use of patterns have been helpful for the argument decomposition process. In addition, the engineers

are able to store successful argument structures either for future reference or, when appropriate, for direct re-use. To create a pattern, a successful argument is typically deprived of any details and is utilised as a blueprint for future arguments. Unfortunately, the cohesion-related issues are more difficult to address. Towards this end, recent research is focusing on model-based processes, so that the system architecture helps maintain order and reduce ambiguity in the argument structure. The downside with many approaches is that the production and maintenance of arguments is handled manually; thus, the patterns are employed mostly for guidance purposes and use of the model-based paradigm is limited to the analysis and generation of evidence artefacts. A plausible way to simultaneously reduce errors and accelerate the process whilst maintaining consistency is by introducing systematic approaches that can also support automation.

The method presented in this thesis takes concrete steps towards the alleviation of these issues of modern safety assurance. The research hypothesis that by connecting the system models with the safety analysis and a pattern that describes the target argument structure whilst automating a substantial part of the process described in the standards has been verified. Overall, the examined examples were similar to those provided in the guidelines and the semi-automatically generated arguments by the method were found to be appropriate. Further, the time benefit compared to manual approaches means that it is more practical to assess and argue about the system even from the early stages of development without significant drawbacks; thus, enhancing the design iteration with safety-related feedback. The focus of this thesis was mostly the aviation and automotive industries and the approach and tool, with a few enhancements, could be used directly in such applications. However, industry professionals from other domains, that also use the concept of SILs, could benefit from this work if changes to compensate for the potential differences are accounted for and implemented. In Section 1.10, research objectives were set for guiding the progress towards this end.

- To examine safety standards for converging elements that could infer a pattern for safety assessment across the relevant domains.



This objective revolves around the investigation of safety activities whilst looking for commonalities across safety standards. A critical review of the literature in safety assessment and assurance, metamodels, argument patterns and conventional practices in safety case productions was presented throughout Chapters 2, 3 and partially 4. Through this process, it became evident that there are commonalities that support the elicitation of a common pattern for arguing safety across standards and therefore industries.

- To operationalise the pattern with the development of a metamodel for connecting safety arguments with system models.

The produced metamodel was designed, based on the pattern identified in Chapter 4, and presented in detail in Chapter 5. It incorporates all the common elements found in safety standards and its structure is based on a modified version of the HiP-HOPS metamodel, the GSN metamodel and a structure that allows the arguments to be connected with the models which they represent.

- To implement the metamodel in a software supporting tool for demonstrating feasibility.

The design process, considerations and related code fragments were demonstrated in Chapter 5. Completion of this objective led to the creation of a software tool. This allowed for testing the feasibility of the approach, as the application of the method on various examples was facilitated. Finally, through this objective, it was possible to evaluate the performance of the software engineering concept, which highlighted the importance of automation when producing safety arguments.

- To evaluate the approach through a case study.

The case study was presented in Chapter 6. Sufficient research was conducted regarding brake systems in aviation for better understanding the implications of the architecture and its operation. Then, the method was applied on the system and a brief evaluation, based on defined criteria, on the resulting argument was provided. Finally, software performance metrics were presented in Table 6.3. This case study allowed for the evaluation of the approach in the aviation industry. Although the results were useful for demonstrating the feasibility of the method and, to a certain extent, the

performance of the tool, the evaluation was modest. Therefore, future evaluations should focus on exposing the proposed method to third parties that would enable examination of case studies, not only larger in scale and complexity, but in other domains as well.

## 7.2 Contributions

The thesis provides an approach usable from the early stages of development, that follows the safety assessment activities in compliance with safety standards and generates an argument of safety about the system in question. The process entails the following:

- **functional design and analysis**, to enable early assessment
- **analysis methods**, advocated by safety guidelines, with a **model-based approach**
- **requirements allocation**, as part of the supporting evidence
- **argument patterns**, for guiding the argument

The above encompass a major part of the actions necessary for supporting safety assurance and lead to a preliminary safety case. Major contributions of the method involve:

- a) **the incorporation of automation.** The semi-automatic nature of the approach makes it adaptable to iterative lifecycles as less effort is required to produce the deliverables. Moreover, the deployment of the model-based paradigm not only enables part of the automation, but also enhances the traceability within the argument. The latter facilitates the linkage between information artefacts and assessment artefacts and makes it easier for the engineers to detect connections between low-level and high-level requirements. In our case, the method automatically handles system analysis techniques, requirement decomposition and allocation as well as pattern instantiation, that ultimately results in the safety argument. User interaction is minimal as it takes place mostly during the annotation, system and pattern design; thus, errors are minimised.
- b) **a functioning software engineering framework that materialises the method.** The framework is adaptable; therefore, it can be used directly by engineers for practical work or

as an exemplar for developers. In the latter case, it could prove to be helpful in understanding the technical details of the approach and also help them build their customised versions.

Last but not least, in Chapters 3 and 4, the thesis provided reasonable indications that the method can be utilised as a guide for producing safety arguments in other industrial domains of safety-critical systems. The rationale behind this concept focuses on the converging common elements found in safety standards that involve the safety integrity levels as a form of requirement abstraction. Despite the plausibility of the information provided, it is required to further solidify that argument. To do so, more standards, such as the CENELEC in the railway domain, need to be thoroughly examined. Undertaking a number of industrial case studies for the corresponding domains, would certainly be helpful in proving feasibility.

The conception of the model-connected safety cases required significant effort in researching the literature. Most importantly, the operationalisation of the method required the integration of elements from various areas. Specifically, assessment activities and concepts in safety standards, safety analysis techniques, modelling and metamodels, argument patterns and software engineering. Furthermore, existing methods and tools as well as current practices required investigation. This process led to a critical review in Chapters 2, 3 and 4 that helped in sculpting the proposed method and it was also essential in its realisation.

### 7.3 Limitations and Future Work

To establish a more sustainable approach for safety assessment and assurance, there are certain aspects that require further attention. These can be divided into technical (i.e. framework-related) and theoretical (approach-related); however, here we focus on the theoretical which have a more direct impact on the method, rather than the technical which can differ depending on the implementation.

From a theoretical standpoint, the method can be reinforced so that the generated safety argument is more compelling. As discussed in Section 1.5, the nature of the arguments presented in this thesis are product-based and therefore cover safety issues related directly with the system, focusing on its

functions and supporting systems. However, safety standards, such as the ISO26262, also examine other aspects that indirectly affect the system (Birch et al., 2013). For example, any sort of factors related to the work products, such as the processes themselves or the competency and technical expertise of the practitioner. These are commonly referred to as process-based arguments and are utilised for strengthening the confidence in evidence artefacts and claims. This is important since claiming that a system is safe because it was assessed via technique ‘X’, can sometimes be insufficient evidence, especially for certification purposes. It has been highlighted throughout the literature review that many of the analysis techniques are subject to the practitioner’s background and understanding. Additionally, a new analysis technique always requires to be validated before being accepted. Hence, the method could be modified to incorporate this type of arguments too. At the moment, the MCSU structure, shown in Chapter 5, could store and handle such information, but neither the method nor the underlying framework support directly the necessary activities. Consequently, this could be excellent material for future work, as it further supports the safety case, and it is relevant for the safety standards discussed in Chapter 3 that guide the development in their respective industries (i.e. aviation and automotive).

One of the current limitations of this method is that software-related assurance is implicit. This means that the system safety arguments imply that both hardware and software parts are safe due to the appropriately allocated requirements. Even when following the appropriate documents, such as the DO-178C in aviation, the rationale behind certain activities is not always clear. Hence, creating the proper reasoning related to these activities, within the argument pattern or safety argument can be challenging. In (Holloway, 2015), it is mentioned that despite empirical evidence showing adequacy of implicit safety cases, the reasons are still obscured. This leads to uncertainty when attempting to predict future safety due to earlier reasoning being hard to interpret. A collaboration between the FAA and NASA through the Explicate ’78 project, has provided excerpts of an explicit assurance case based on the implicit rationale, related to the purpose, found in DO-178C. As a result, customising the pattern, for the aviation-related arguments, to include this explicit strategy could help

communicate not only the safety activities and results during development, but also the purpose behind each activity and the reason why this activity is adequate; thus, increasing confidence in the argument.

Another avenue that requires further work would be the expansion of the method towards the inclusion of additional safety standards. Based on the discussion in Chapter 3, many standards make use of the concept of SILs. However, it is also mentioned that there are differences in the guidelines related either to the assurance process lifecycle, definitions or even requirement equivalency, especially when dealing with the software of systems (Blanquart et al., 2018). Arguably, the requirements optimisation algorithm and rulesets as well as the pattern are adaptable; hence, they could be customised to compensate for these discrepancies. Therefore, as part of covering more domains, further work should focus on customising the method to incorporate more processes for producing appropriate evidence, such as dedicated software analyses.

The method currently allows for a limited set of system analysis techniques and therefore results may not be adequate for any given situation. For instance, under the ISO 26262, verification of components assigned with ASIL D requires artefacts from formal verification approaches. As future work, the method and framework could integrate more techniques to account for the analysis of more complex system behaviour, in general, or when they are appropriate for the situation. For example, the inclusion of Markov Analysis would allow for accurately predicting the transitions of a system between its states. On the other hand, the incorporation of formal methods would address verification of component designs, when suggested in the guidelines, and solidify the verification part of the argument produced with this method.

Finally, the recent increase of interest in Cyber-Physical-Systems (CPS) allows us to presume that a stronger connection between physical and digital entities, in a co-operative manner, in various domains, such as the Health Care and Automotive, is impending. These domains require dependable systems to avoid incidents that would result in harm or large-scale catastrophes. Considering that any connection between two or more CPS would result into a distinct system, predicting the amount of

different potential configurations would be infeasible. Therefore, evaluating and assuring dependability or safety for these systems is not possible with conventional methods. The DEIS project has proposed and implemented an innovative approach, the Digital Dependability Identities (DDIs), for addressing dependability assurance of CPS (Wei et al., 2018). Considering the increasing integration of systems into systems of systems, further work could focus on the implementation of a framework for producing design-time and runtime DDIs (i.e. dynamic safety assurance). Enabling such capabilities could be achieved either by mirroring certain aspects found in the publications of the DEIS project or by customising the framework to allow interoperability between the tools. As mentioned in Section 4.6.1, the DEIS metamodels, inherit various elements from the Structured Assurance Case Metamodel (SACM). Thus, to follow either of the two strategies, the metamodel presented in this thesis should be modified by including SACM elements that describe all the necessary properties and model interrelations.

All of the above limitations can potentially translate to future work for the method itself. On a more technical note, there is room for experimentation that could help in different ways. One example is the transition of the tool towards a web-based solution. Instead of burdening the user with executables or IDEs and source code, the tool could be converted into a service available on the web. This way, it would be easy to use from any device and could promote it into a safety assurance hub. For example, the website could serve as a repository, where safety engineers exchange argument patterns, analysis methods, and evidence artefacts related to safety-critical systems.

Last but not least, although the program seems to produce the correct outcomes, it is almost certain that there might be traces of errors or undefined behaviour. To safeguard against such errors, it would be beneficial to create a formal specification (for important parts) of the software and then apply formal testing methods; thus, ensuring that the output is correct, based on the specification we provided, across the input domain. This will help build confidence both in the supporting tool during the assessment and the resulting argument during the assurance phase (through a process-based argument added to the rest of the argument structure).

## References

---

- Aboudolas, K., Papageorgiou, M., Kouvelas, A. & Kosmatopoulos, E. (2010) A rolling-horizon quadratic-programming approach to the signal control problem in large-scale congested urban road networks. *Transportation Research Part C Emerging Technologies*, 18(5), 680-694.
- Academic (2020) *Aviation Dictionary*. Available online: [https://aviation\\_dictionary.enacademic.com/6005/segmented-rotor\\_brake](https://aviation_dictionary.enacademic.com/6005/segmented-rotor_brake) [Accessed 15/06/2020].
- ACWG (2018) *Goal Structuring Notation Community Standard Version 2*. The Assurance Case Working Group (ACWG). Available online: <https://scsc.uk/scsc-141B> [Accessed 19/4/2019].
- Aeronautics Guide (2020) *Aircraft Brakes*. Available online: [https://www.aircraftsystemstech.com/p/aircraft-brakes\\_9081.html](https://www.aircraftsystemstech.com/p/aircraft-brakes_9081.html) [Accessed 12/04/2020].
- Aerospaceweb.org (2018) *Thrust Reversing*. Available online: <http://www.aerospaceweb.org/question/propulsion/q0008a.shtml> [Accessed 11/08/2020].
- Alana, E., Naranjo, H., Yushtein, Y., Bozzano, M., Cimatti, A., Gario, M., de Ferluc, R. & Garcia, G. (2012) Automated generation of FDIR for the compass integrated toolset (AUTOGEF). *DASIA 2012 Data Systems in Aerospace*. Drubrovnik, Croatia, 14-16 May 2012, ESA SP 701.
- AltaRica Project (2020) *Methods and Tools for AltaRica Language*. Available online: [https://altarica.labri.fr/wp/?page\\_id=23](https://altarica.labri.fr/wp/?page_id=23) [Accessed 01/09/2018].
- Arnold, A., Gerald, P., Griffault, A. & Rauzy, A. (2000) The Altarica Formalism for Describing Concurrent Systems. *Fundamenta Informaticae*, 34, 109-124.
- Arnold F., Belinfante A., Van der Berg F., Guck D. & Stoelinga M. (2013) DFTCalc: A Tool for Efficient Fault Tree Analysis. In Bitsch F., Guiochet J. & Kaâniche M. (eds) *Computer Safety, Reliability, and Security*. SAFECOMP 2013. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 8153.
- Azevedo, L. S., Parker, D., Walker, M., Papadopoulos, Y. & Araùjo, R. E. (2013) Automatic Decomposition of Safety Integrity Levels: Optimization by Tabu Search. *SAFECOMP 2013 - Workshop CARS (2nd Workshop on Critical Automotive applications: Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security*. Toulouse, France, September 2013.
- Azevedo, L., Parker, D., Walker, M. & Araùjo, R. E. (2014) Assisted Assignment of Automotive Safety Requirements. *IEEE Software*, 31(1), 62-68.

B737 Technical Site (2020) *Landing Gear*. Available online:

<http://www.b737.org.uk/landinggear.htm#Brakes> [Accessed 05/08/2020].

Bäck T., Fogel D. B. & Michalewicz T. (1997) *Handbook of Evolutionary Computation*. Oxford University Press.

Baufreton, P., Blanquart, J.P., Boulanger, J.L., Delseny, H., Derrien, J.C., Gassino, J., Ladier, G., Ledinot, E., Leeman, M., Quéré, P. & Richque, B. (2010) Multi-domain comparison of safety standards. *The 5th International Conference on Embedded Real Time Software and Systems (ERTS2 2010)*. Toulouse, France, 19-21 May 2010.

Bell, R. (2014) IEC61508: Assessment, Certification and other Assurance Measures. *Engineering Safety Consultants*. November 2014.

Bellman, R. (1957) *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Bieber, P., Delmas, R. & Seguin, C. (2011) DALculus - Theory and Tool for Development Assurance Level Allocation. *30<sup>th</sup> International Conference on Computer Safety, Reliability, and Security*. SAFECOMP 2011, Naples, Italy, September 19-22, 2011, 43-56.

Birch, J., Rivett, R., Habli, I., Bradshaw, B., Botham, J., Higham, D., Jesty, P., Monkhouse, H. & Palin, R. (2013) Safety Cases and Their Role in ISO 26262 Functional Safety Assessment. In *Bitsch F., Guiochet J. & Kaâniche M. (eds) Computer Safety, Reliability, and Security*. SAFECOMP 2013. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 8153.

Bittner, B., Bozzano, M. & Cimatti, A. (2017) Timed Failure Propagation Analysis for Spacecraft Engineering: The ESA Solar Orbiter Case Study. *International Symposium on Model-Based Safety Assessment 2017*. Trento, Italy, 11-13 September 2017, 255-271.

Blanquart, J-P., Ledinot, E., Gassino, J., Baufreton, P., Boulanger, J-L., Brouste, S., Camus, J., Comar, C., Quere, P. & Ricque, B. (2018) Software Safety - A Journey Across Domains and Safety Standards. *9<sup>th</sup> European Congress on Embedded Real Time Software and Systems*. Toulouse, France, January 2018.

Bloomfield, R. & Bishop, P. (2010) Safety and Assurance Cases: Past, Present and Possible Future – an Adelard Perspective. In Dale, C. & Anderson, T. (eds) *Making Systems Safer. Proceedings of the Eighteenth Safety-Critical Systems Symposium*. Bristol, UK, 9-11 February 2010, 51-67.

Boeing (2016) *FAA Reference Code and Approach Speeds for Boeing Aircraft*. Available online: <http://www.boeing.com/assets/pdf/commercial/airports/faqs/arcandapproachsheets.pdf> [Accessed 04/08/2020].



- Boeing (2020) *Boeing 787 Dreamliner Completes First Flight*. Available online: <https://boeing.mediaroom.com/2009-12-15-Boeing-787-Dreamliner-Completes-First-Flight#:~:text=SEATTLE%2C%20Dec.,Paine%20Field%20in%20Everett%2C%20Wash.> [Accessed 04/08/2020].
- Bozzano, M., Cimatti, A., Katoen, J.P., Katsaros, P., Mokos, K., Nguyen, V., Noll, T., Postma, B. & Roveri, M. (2014) Spacecraft Early Design Validation using Formal Methods. *Reliability Engineering & System Safety*, 132, 20–35.
- Bozzano, M. & Villafiorita, A. (2006) The FSAP/NuSMV-SA Safety Analysis Platform. *International Journal on Software Tools for Technology Transfers (STTT) – Special Section on Advances in Automated Verification of Critical Systems*, 9(1), 5-24.
- British Standard (2001) *BS IEC 61882: Hazard and Operability Studies (HAZOP Studies) – Application Guide*. UK: BS IEC.
- Brown, K. N. & Miguel, I. (2006) Uncertainty and Change. In Rossi, F., van Beek, P. & Walsh, W. (eds) *Handbook of Constraint Programming*. Foundations of Artificial Intelligence, 2(21), 731-760.
- CAE Framework (2020) *CAE Concepts*. Claims Arguments Evidence. Available online: <https://claimsargumentsevidence.org/notations/claims-arguments-evidence-cae/> [Accessed 19/04/2020].
- Carlson, Carl S. (2014) Understanding and Applying the Fundamentals of FMEAs. *2014 Annual Reliability and Maintainability Symposium*. Colorado Springs, USA, 27-30 January 2014.
- Čepin, M. & Mavko, B. (2002) A Dynamic Fault Tree. *Reliability Engineering & System Safety*, 75(1), 83–91.
- Clements, P. & Northrop, L. (2001) *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, Boston, MA, USA.
- DEIS (2020) *Project Information*. DEIS Project. Available online: <http://www.deis-project.eu/project-information/> [Accessed 19/04/2020].
- de la Vara, J. L., Genova, G., Alvarez-Rodriguez, J. M. & Llorens, J. (2016) An Analysis of Safety Evidence Management with the Structured Assurance Case Metamodel. *Computer Standards & Interfaces*, 50, 179-198.
- Denney, E., Naylor, D. & Pai, G. (2014) Querying Safety Cases. *SAFECOMP 2014, 33rd International Conference on Computer Safety, Reliability, and Security*. Florence, Italy, 10-12 September 2014, 294-309.

Despotou, G., Apostolakis, A. & Kolovos, D. (2015) *Assuring Dependable and Critical Systems: Implementing the Standards for Assurance Cases with ACedit*. Available online: DOI: 10.13140/2.1.5085.8245 [Accessed 01/09/2018].

Dugan, J.B., Bavuso, S.J. & Boyd, M.A. (1992) Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3), 363–377.

Department of Defence (2001) *Systems Engineering Fundamentals*. Fort Belvoir, Defense Acquisition University Press. Available online: [https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-885j-aircraft-systems-engineering-fall-2005/readings/sefguide\\_01\\_01.pdf](https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-885j-aircraft-systems-engineering-fall-2005/readings/sefguide_01_01.pdf). [Accessed 06/07/2019].

Dugan, J., Sullivan, K. & Coppit, D., (2000) Developing a Low-Cost High-Quality Software Tool for Dynamic Fault-Tree Analysis. *IEEE Transactions on Reliability*, 49(1), 49-59.

EASA (2011) *CS-25/Amendment 11: AMC 25.1309 System Design and Analysis*. European Aviation Safety Agency (EASA).

EAST-ADL (2017) *About EAST-ADL*. Available online: <http://www.east-adl.info/Specification.html> [Accessed 12/02/2018].

Eckberg, C. R. (1964) *WS-133B: Fault Tree Analysis Program Plan*. Seattle, WA: The Boeing Company.

Eclipse (2020) *Eclipse documentation - Current Release*. Available online: <https://help.eclipse.org/2020-06/index.jsp> [Accessed 18/04/2020].

Elsayed, A. (2013) *Reliability Engineering*. Hoboken, New Jersey, USA: Wiley.

E.P.PLANS (2006) *Failure Mode and Effect Analysis – Methodology*. Available online: <http://www.epplans.com/external/menu23/menu1.html> [Accessed 12/02/2018].

Epsilon (2020) *The Epsilon Object Language*. The Eclipse Foundation. Available online: <https://www.eclipse.org/epsilon/doc/eol/> [Accessed on 07/04/2020].

EPSRC (2010) *Feasibility Study of Energy Recovery from Landing Aircraft*. Engineering and Physical Sciences Research Council. Available online: <https://gow.epsrc.ukri.org/NGBOVViewGrant.aspx?GrantRef=EP/H004351/2> [Accessed 05/08/2020].

Ericson, C. (1999) Fault Tree Analysis - A History. *17<sup>th</sup> International System Safety Conference* 1999. Orlando, Florida, USA, 1999.

FAA (1988) *AC 25.1309-1A - System Design and Analysis – Document Information*. Washington, DC, USA: Federal Aviation Administration (FAA).

FAA (2006) *NAS System Engineering Manual*. USA: Federal Aviation Administration (FAA).

FAA (2018) *FAA Regulations. All Current & Historical Regulations*. Available online: [https://www.faa.gov/regulations\\_policies/faa\\_regulations/](https://www.faa.gov/regulations_policies/faa_regulations/) [Accessed 20/02/2020].

FAA (2019) *14 CFR Part 39. Airworthiness Directives; The Boeing Company Airplanes*. Available online: [https://rgl.faa.gov/Regulatory\\_and\\_Guidance\\_Library/rgad.nsf/0/a89e21a1caa5e0b0862583ee005345b1/\\$FILE/2019-08-05.pdf](https://rgl.faa.gov/Regulatory_and_Guidance_Library/rgad.nsf/0/a89e21a1caa5e0b0862583ee005345b1/$FILE/2019-08-05.pdf) [Accessed 13/06/2020].

Feiler, P. H., Gluch, D. P. & Hudak, J. J. (2006) The Architecture Analysis & Design Language (AADL): An Introduction. In U.S. Department of Defense (ed) *Performance-Critical Systems*. Carnegie Mellon University.

Feiler, P. & Rugina, A. (2007) *Dependability Modeling with the Architecture Analysis & Design Language (AADL)*. Software Engineering Institute Report.

Fenelon, P. & McDermid, J.A. (1992) *New Directions in Software Safety: Causal Modelling as an Aid to Integration*. High Integrity Systems Engineering Group, Department of Computer Science, University of York.

Fenelon, P. & McDermid, J.A. (1993) An Integrated Tool Set for Software Safety Analysis. *Journal of Systems and Software*, 21(3), 279-290.

Fosler-Lussier, E. (1998) *Markov Models and Hidden Markov Models: A Brief Tutorial*. International Computer Science Institute, Berkeley, California.

Fogel, L. J., Owens, A. J. & Walsh, M. J. (1966) *Artificial Intelligence Through Simulated Evolution*. Wiley.

FTDSolutions (2016) *Reliability Workbench: Product Description*. Available online: <http://ftdsolutions.com/product/reliability-workbench/> [Accessed: 01/01/2020].

Ge, X., Paige, R. F. & McDermid, J. A. (2009) Probabilistic Failure Propagation and Transformation Analysis. In Buth, B., Rabe, G. & Seyfarth, T. (eds) *Computer Safety, Reliability, and Security. 28<sup>th</sup> International Conference. SAFECOMP 2009, Hamburg, Germany, 15-18 September 2009*.

- Ge, X., Paige, R. F., & McDermid, J. A. (2010). Analysing System Failure Behaviours with PRISM. *SSIRI-C 2010: The Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement Companion*. Los Alamitos, Singapore, 9-11 June 2010, 130-136.
- Ghanem, B., Cao, Y. & Wonka, P. (2015) Designing Camera Networks by Convex Quadratic Programming. *Computer Graphics Forum*, 34(2), 69-80.
- Gill, P. E. & Wong, E. (2015) Methods for convex and general quadratic programming. *Mathematical Programming Computation*, 7, 71-112.
- Glover, F. (1989) Tabu search: Part I. *ORSA Journal on Computing*, 1(3), 190-206.
- Goodrich (2020) *Goodrich 787 Electro-Mechanical Brake Selected by Airlines around the Globe*. Available online:  
[http://www.goodrich.com/cap/Documents/SYSTEM%20FACT%20SHEET\\_Wheels%20and%20Brakes%20787%20Product%20Fact%20Sheet.pdf](http://www.goodrich.com/cap/Documents/SYSTEM%20FACT%20SHEET_Wheels%20and%20Brakes%20787%20Product%20Fact%20Sheet.pdf) [Accessed 14/06/2020].
- Grunske, L., Kaiser, B. & Papadopoulos, Y. (2005) Model-Driven Safety Evaluation with State-Event-Based Component Failure Annotation. *8<sup>th</sup> International Symposium on Component-Based Software Engineering*. CBSE 2005. St. Louis, MO, USA, 14-15 May 2005, 33-48.
- Hawkins, R., Clegg, K., Alexander, R. & Kelly, T. (2011) Using a Software Safety Argument Pattern Catalogue: Two Case Studies. *30<sup>th</sup> International Conference on Computer Safety, Reliability, and Security*. Naples, Italy, 19-22 September 2011, 185-198.
- Holland, J. H. (1975) *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.
- Holloway, C. M. (2015) *Explicate '78: Uncovering the Implicit Assurance Case in DO-178C*. Safety-Critical Systems Symposium 2015. NASA Langley Research Center Hampton, VA, US.
- HSE (1994) *The Railways (Safety Case) Regulations 1994 – Guidance on Regulations*. UK: Health and Safety Executive.
- HSW (1974) *Health and Safety at Work etc. Act 1974 c.37*. UK: UK Public General Acts.
- IEC (1998) *CEI IEC 61508-1: Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems*. International Electrotechnical Commission, Geneva: IEC.
- IEC (2010) *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems*. International Electrotechnical Commission (IEC).

IEC (2011) *IEC 61513: Nuclear Power Plants – Instrumentation and Control Important to Safety*. General Requirements for Systems. International Electrotechnical Commission (IEC).

IEC (2020) *Electropedia: The World's Online Electrotechnical Vocabulary*. Available at: <http://www.electropedia.org/iev/iev.nsf/6d6bdd8667c378f7c12581fa003d80e7?OpenForm> [Accessed: 05 March 2020].

Zetlin, M. (2019) In a Shocking Development, Safety Questions Arise About Another Boeing Jet. American and United Both Fly Them. *Inc.com*. Available online: <https://www.inc.com/minda-zetlin/boeing-787-dreamliner-safety-issues-north-charleston-plant-debris-planes.html> [Accessed 03/05/2020].

ISO (2011) *ISO 26262: Road vehicles - Functional safety*. International Organization for Standardization (ISO).

Isograph (2017) *Fault Tree Analysis*. Available online: <https://www.isograph.com/software/reliability-workbench/fault-tree-analysis-intro/> [Accessed on: 15 September 2017].

Joba, S. (2015) *Are You Modelling: Visualizing Safety Cases - Tim Kelly on GSN (Goal Structuring Notation)*. Available online: <http://areyoumodeling.com/2015/02/23/gsn/> [Accessed on 25/08/2016].

Johannessen, P., Grante, C., Alminger, C. & Eklund, U. (2001) Hazard Analysis in Object-Oriented Design of Dependable Systems. *2001 International Conference on Dependable Systems and Networks (DSN'01)*. Goteborg, Sweden, 1-4 July 2001, 507-512.

Joshi, A., Heimdahl, M., Miller, S. & Whalen, M. (2006) *Model-Based Safety Analysis*. Washington, DC, USA: National Aeronautics and Space Administration (NASA).

Kaiser, B., Liggesmeyer, P. & Mäkel, O. (2003) A New Component Concept for Fault Trees. *Eighth Australian Workshop on Safety-Related Programmable Systems (SCS'03)*. Canberra, ACT, Australia, 9-10 October 2003, 37-46.

Kelly T. P. (1998) *Arguing Safety – A Systematic Approach to Managing Safety Cases*. Thesis. University of York.

Kelly T. & Weaver R. (2004) The Goal Structuring Notation – A Safety Argument Notation. *Dependable Systems and Networks 2004 Workshop on Assurance Cases*.

Kelly, T. (2008) Are 'Safety Cases' Working? *Safety Critical Systems Club Newsletter*, 17(2), 31-3. Available online: <https://www-users.cs.york.ac.uk/~tpk/2008scscarticlekelly.pdf> [Accessed 10/10/2019].

- Lawley, H. G. (1974) Operability studies and hazard analysis. *Chemical Engineering Progress*, 70, 46-56.
- Leveson, N., Dulac, N., Marais, K. & Carroll, J. (2009) Moving Beyond Normal Accidents and High Reliability Organizations: A Systems Approach to Safety in Complex Systems. *Organization Studies* 30(2-3), 227-249.
- Leveson, N. G. (2011a) *Engineering a Safer World: Systems Thinking Applied to Safety*. Cambridge, Mass: MIT Press.
- Leveson, N. G. (2011b) The Use of Safety Cases in Certification and Regulation. *Journal of System Safety*, 47(6).
- Li, S. & Li, X. (2014) Study on Generation of Fault Trees from Altarica Models. *Procedia Engineering*, 80, 140-152.
- Linzey, W.G. (2006) *Development of an Electrical Wire Interconnect System Risk Assessment Tool*. Federal Aviation Administration (FAA). Available online: <http://www.tc.faa.gov/its/worldpac/techrpt/artn06-17.pdf> [Accessed 10/08/2019].
- Maguire, R. (2006) *Safety Cases and Safety Reports*. Farnham: Ashgate Publishing Ltd.
- Mancuso, A., Compare, M., Salo, A. & Zio, E. (2019) Portfolio optimization of safety measures for the prevention of time-dependent accident scenarios. *Reliability Engineering and System Safety*, 190.
- Maré, J.-C. (2017) Aerospace Actuators 2: Signal-by-Wire and Power-by-Wire. In *Systems and Industrial Engineering – Robotics Series*. Wiley-ISTE, 187.
- Marsh, W. (1999) SafEty and Risk Evaluation using bayesian NETs: SERENE. SERENE Partners. Available online: <http://www.eecs.qmul.ac.uk/~norman/papers/serene.pdf>
- Matsuno, Y., Takamura, H. & Ishikawa, Y. (2010) A Dependability Case Editor with Pattern Library. *IEEE 12th International Symposium on High Assurance Systems Engineering*. San Jose, CA, USA, 3-4 November 2010.
- Matsuno, Y. (2014) A Design and Implementation of an Assurance Case Language. *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Atlanta, GA, USA, 23-26 September 2014, 630-64.
- Matsuno, Y. (2017) D-Case Communicator: A Web Based GSN Editor for Multiple Stakeholders. In: Tonetta S., Schoitsch E. & Bitsch F. (eds) *Computer Safety, Reliability, and Security. SAFECOMP*

2017 Workshops, ASSURE, DECSoS, SASSUR, TELERISE, and TIPS. Trento, Italy, 12 September 2017, 64-69.

Microsoft (2020) Acquiring High-Resolution Time Stamps. Available online: <https://docs.microsoft.com/en-gb/windows/win32/sysinfo/acquiring-high-resolution-time-stamps?redirectedfrom=MSDN> [Accessed 17/08/2020].

MoD (1996a) *00-56 Safety Management Requirements for Defence Systems*. UK: Ministry of Defence.

MoD (1996b) *JSP 430: Ship Safety Management System Handbook*. UK: Ministry of Defence.

MoD (1997) *00-55 Requirements of Safety Related Software in Defence Equipment*. UK: Ministry of Defence.

Möhrle, F., Zeller, M., Höfig, K., Rothfelder, M. & Liggesmeyer, P. (2016) Automating Compositional Safety Analysis Using a Failure Type Taxonomy for Component Fault Trees. In Walls, L., Revie, M. & Bedford, T. (eds) *Risk, Reliability and Safety: Innovating Theory and Practice*. ESREL 2016. Glasgow, Scotland, 25-29 September 2016.

Mosleh, A., Fleming, K.N., Parry G.W., Paula H.M., Rasmunson D.M. & Worledge D.H. (1989) *Procedures for Treating Common Cause Failures in Safety and Reliability Studies: Analytical Background and Techniques*. USA: Nuclear Regulatory Commission, NUREG/CR--4780, 2.

Nidhra, S. (2012) Black Box and White Box Testing Techniques – A Literature Review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2), 29-50.

Oliveira, A., Braga, R. T. V., Masiero, P. C., Papadopoulos, Y., Habli, I. & Kelly, T. (2015) Supporting the Automated Generation of Modular Product Line Safety Cases. *Theory and Engineering of Complex Systems and Dependability: Proceedings of the Tenth International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)*. Brunów, Poland, 29 June - 3 July 2015, 319-330.

OMG (2018) *Structured Assurance Case Metamodel (SACM)*. Object Management Group, Version 2.0. Available online: <https://www.omg.org/spec/SACM/2.0> [Accessed 05/03/2020].

OMG (2019) *Structured Assurance Case Metamodel (SACM)*. Object Management Group, Version 2.1. Available online: <https://www.omg.org/spec/SACM/About-SACM/> [Accessed 05/03/2020].

Papadopoulos, Y. & Maruhn, M. (2001) Model-Based Automated Synthesis of Fault Trees from Matlab-Simulink Models. *International Conference on Dependable Systems and Networks*. Goteborg, Sweden, 1-4 July 2001, 77-82.



- Papadopoulos, Y. & McDermid, J. A. (1999) The Potential for a Generic Approach to Certification of Safety-Critical Systems in the Transportation Sector. *Reliability Engineering and System Safety*, 63(1), 47-66.
- Papadopoulos, Y., Walker, M., Parker, D., Rüde, E., Hamann, R., Uhlig, A., Grätz, U. & Lien, R. (2011) Engineering Failure Analysis and Design Optimisation with HiP-HOPS. *Engineering Failure Analysis*, 18(2), 590–608.
- Parker, D. J. (2010) *Multi-Objective Optimisation of Safety-Critical Hierarchical Systems*. PhD thesis. The University of Hull.
- Point, G. & Rauzy, A. (1999) AltaRica: Constraint automata as a description language. *European Journal on Automation*, 33(8-9), 1033-1052.
- Qt (2020) *About Us*. The Qt Company. Available online: <https://www.qt.io/company#why> [Accessed on 10/05/2020].
- Rao, K. D., Rao, V. S., Verma, A. K. & Srividya, A. (2010) Dynamic Fault Tree Analysis: Simulation Approach. In Faulin, J., Juan, A. A., Martorell, S. & Ramírez-Márquez, E. (eds) *Simulation Methods for Reliability and Availability of Complex Systems*. Springer London, 41-64.
- Rausand, M. (2014) *Reliability of Safety-Critical Systems: Theory and Applications*. Wiley Publishing.
- Rechenberg, I. (1965) *Cybernetic Solution Path of an Experimental Problem*. Farnborough, UK: Royal Aircraft Establishment, 1112.
- Retouniotis, A., Papadopoulos, Y., Sorokos, I., Parker, D., Matragkas, N. & Sharvia, S. (2017) Model-Connected Safety Cases. 5<sup>th</sup> International Symposium, IMBSA 2017. Trento, Italy, 11-13 September 2017, 50-63.
- Robens, Lord, Beeby, G.H., Robinson, S.A., Shaw, A., Windeyer, B.W., Wood, J.C. & Wake, M. (1972) *Safety and Health at Work: Report of the Committee, 1970-72*. London, UK: HMSO.
- RTCA (2011) *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. USA: Radio Technical Commission for Aeronautics (RTCA).
- Rushby, J. (2015) *The Interpretation and Evaluation of Assurance Cases*. Menlo Park, CA, USA, Stanford Research Institute (SRI) International.
- SAE (2010) *ARP4754-A: Guidelines for Development of Civil Aircraft and Systems*. Society of Automotive Engineers (SAE).



SAE (1996) *ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. Society of Automotive Engineers (SAE).

SAE (2010) *ARP4754-A: Guidelines for Development of Civil Aircraft and Systems*. Society of Automotive Engineers (SAE).

Safran (2020) *Wheels and Brakes: Boeing 787 Dreamliner Brake*. Available online: <https://www.safran-landing-systems.com/wheels-and-brakes/products/boeing-787-dreamliner-brake> [Accessed 14/06/2020].

Sljivo, I., Gallina, B., Carlson, J. & Hansson, H. (2014) Generation of Safety Case Argument-Fragments from Safety Contracts. In Bondavalli, A. & Di Giandomenico, F. (eds) *33rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2014)*. Florence, Italy, 10-12 September 2014, 170–185.

Sljivo, I., Gallina, B., Carlson, J., Hansson, H. & Puri, S. (2016) A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis. *Journal of Systems and Software*, 131, 570-590.

Sommerville, I. (2011) *Software Engineering*. Boston, MA, USA, Pearson Education, Inc.

Sorokos, I., Papadopoulos, Y., Azevedo, L., Parker, D. & Walker, M. (2015) Automating Allocation of Development Assurance Levels an extension to HiP-HOPS. In López-Mellado, E., Ramírez-Treviño, A., Lefebvre, D. & Ortmeier, F. (eds) *5<sup>th</sup> IFAC International Workshop on Dependable Control of Discrete Systems: DCDS 2015*, 48 7, 9-14.

Sorokos, I., Papadopoulos, Y., Walker, M., Azevedo, L. & Parker, D. (2016) Chapter 11 - Driving Design Refinement: How to Optimize Allocation of Software Development Assurance or Integrity Requirements. In Mistrik, I., Soley, R., Ali, N., Grundy, J. & Tekinerdogan, B. (eds) *Software Quality Assurance*. Elsevier, 237-250.

Sorokos, I. (2017) *Generation of Model-Based Safety Arguments from Automatically Allocated Safety Integrity Levels*. PhD Thesis. The University of Hull.

Sullivan, K.J., Dugan, J.B. & Coppit, D. (1999) The Galileo Fault Tree Analysis Tool. *Digest of Papers: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. Madison, WI, USA, 15-18 June 1999, 232–235.

Sun, L., Silva, N. & Kelly, T. (2014) Rethinking of Strategy for Safety Argument Development. In Bondavalli, A., Ceccarelli, A. & Ortmeier, F. (eds) *SAFECOMP 2014 Workshops: ASCoMS, DECSoS, DEVVARTS, ISSE, ReSA4CI, SASSUR*. Florence, Italy, 8-9 September 2014, 384-395.

Talbi, E. G. (2009) *Metaheuristics from Design to Implementation*. New Jersey, USA: Wiley Publishing.

Toulmin S. E., Rieke R. & Janik, A. (1984) *Introduction to Reasoning*. New York, USA: Macmillan Publishing Company.

Toulmin, S. E. (2003) *The Uses of Argument*. New York, USA: Cambridge University Press.

USA Department of Defense (1949) *MIL-P-1629: Procedure for Performing a Failure Mode Effects, and Criticality Analysis*. Washington, DC, USA: Department of Defense.

USA Department of Defense (1980) *MIL-STD-1629A: Procedures for Performing a Failure Mode, Effects and Criticality Analysis*. Washington, DC, USA: Department of Defense.

USA Department of Defense (2012) *MIL-STD-882E: System Safety*. USA: Department of Defense Standard Practice.

USA Government (1964) *e-CFR Title 14: Aeronautics and Space*. Available online: [https://www.ecfr.gov/cgi-bin/text-idx?c=ecfr&tpl=/ecfrbrowse/Title14/14tab\\_02.tpl](https://www.ecfr.gov/cgi-bin/text-idx?c=ecfr&tpl=/ecfrbrowse/Title14/14tab_02.tpl) [Accessed: 05/03/2020].

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

Vesely, W. E., Goldberg, F. F., Roberts, N. H., Haasl, D. F. (1981) *Fault Tree Handbook*. Washington D.C., USA: US Nuclear Regulatory Commission.

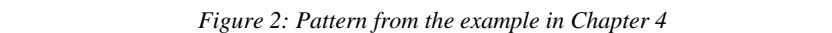
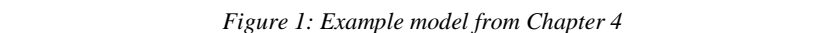
Villemeur, A. (1992) *Reliability, Availability, Maintainability and Safety Assessment, Volume I, Methods and Techniques*. Chichester, UK: John Wiley & Sons.

Vogelstein, J. T., Conroy, M. J., Lyzinski, V., Podrazik, L. J., Kratzer, S. G., Harley, E. T., Fishkind, D. E., Vogelstein, J. R. & Priebe, C. E. (2015) Fast Approximate Quadratic Programming for Graph Matching. *PLoS ONE*, 10(4). Available online: <https://doi.org/10.1371/journal.pone.0121002> [Accessed: 20/03/2019].

Walker, M. & Papadopoulos, Y. (2006) PANDORA: The Time of Priority-AND Gates. *12th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 2006)*, 39(3), 237-242.

Walker, M. D. (2009) *Pandora: A Logic for the Qualitative Analysis of Temporal Fault Trees*. PhD thesis. The University of Hull.

- Washington A., Clothier, R., Neogi, N., Silva, J., Hayhurst, K. & Williams, B. (2019) Adoption of a Bayesian Belief Network for the System Safety Assessment of Remotely Piloted Aircraft Systems. *Safety Science*, 118, 654-673.
- Wilkins, D. J. (2002) *The Bathtub Curve and Product Failure Behavior. Part One - The Bathtub Curve, Infant Mortality and Burn-in*. Available online: <https://www.weibull.com/hotwire/issue21/hottopics21.htm> [Accessed: 05/03/2020]
- Wallace, M. (2005) Modular Architectural Representation and Analysis of Fault Propagation and Transformation. *Electronic Notes in Theoretical Computer Science*, 141(3), 53–71.
- Wei, R., Kelly, T. P., Dai, X., Zhao, S. & Hawkins, R. (2019) Model Based System Assurance Using the Structured Assurance Case Metamodel. *Journal of Systems and Software*, 154, 211-233.
- Wei, R, Kelly, T. P., Hawkins, R. & Armengaud, E. (2018) DEIS: Dependability Engineering Innovation for Cyber Physical Systems. *STAF 2017 Collocated Workshops*. Marburg, Germany, 17-21 July 2017, 409-416.
- Xiao, N., Huang, H.-Z., Li, Y., He, L. & Jin T. (2011) Multiple Failure Modes Analysis and Weighted Risk Priority Number Evaluation in FMEA. *Engineering Failure Analysis*, 18(4), 1162-1170.
- Yamamoto, S. & Matsuno, Y. (2013) An Evaluation of Argument Patterns to Reduce Pitfalls of Applying Assurance Case. *2013 1<sup>st</sup> International Workshop on Assurance Cases for Software-Intensive Systems, ASSURE*. San Fransisco, USA, 19 May 2013.
- Zu, Y., Liu, C., Dai, R. & Sharma, A. (2018) Hybrid Optimal Control for Time-Efficient Highway Traffic Management. *2018 Annual American Control Conference (ACC)*. Milwaukee, WI, USA, 27-29 June 2018.



Finally, Fig. 3 presents the part of the resulting argument similarly to what was demonstrated in Fig. 4.9. Note that even though the argument editor provides the basic shapes for the manual creation of argument elements if the user desires, the argument shown here was generated based on the algorithm described in Section 5.5.

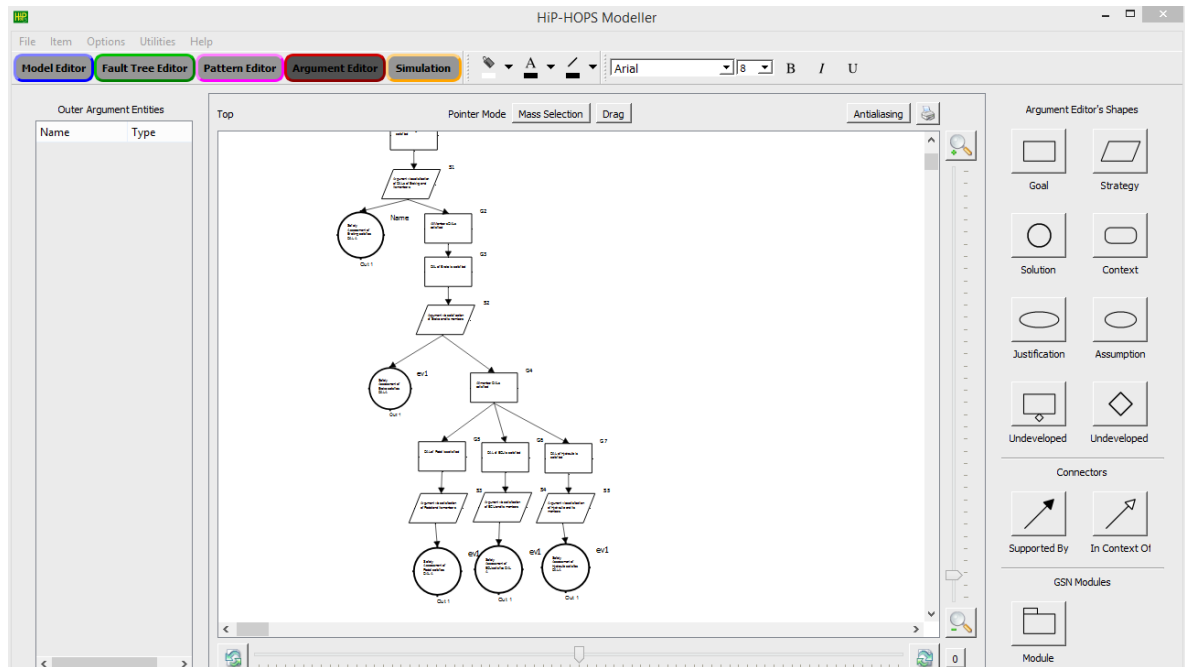


Figure 3: Argument structure from example in Chapter 4

## Appendix II: Metrics Implementation

---

This section briefly presents the means for producing the execution metrics, related to the framework, provided in Table 6.3 (Chapter 6). The code utilised a Windows API for collecting time stamps, known as Query Performance Counter (QPC). This allowed high-precision measuring (Microsoft, 2020) of the execution time for major processes within our tool such as:

- the analysis of the model by the HiP-HOPS engine.
- DAL optimisation algorithm and allocation to the architectural elements by the program.
- pattern instantiation and argument generation (including the creation of tree nodes and their placement on the scene by the ArgumentEditor class).

To achieve this, a few lines of code were placed just before the start of each of the above processes and then a few more lines exactly at the end. Below, an example of the use of the QPC (API) is shown for demonstrative purposes. It is a mix of pseudocode and C++ code.

*// Variable Initialisation Step*

`LARGE_INTEGER Start, End, ElapsedSeconds, Frequency;`

*// Requesting Total Number of Ticks and CPU Frequency*

`QueryPerformanceFrequency(&Frequency); // Assign to Frequency the value of CPU frequency`

`QueryPerformanceCounter(&Start); //Assign to Start the total amount of ticks (resets with reboot)`

-Actions to be performed- This can be any of the processes from the bullet points above.

`QueryPerformanceCounter(&End); //Assign to End the total amount of ticks`

`ElapsedSeconds = (End - Start); // Assign ElapsedSeconds the difference between the values`

`ElapsedSeconds = ElapsedSeconds * 1000; //Adjust ElapsedSeconds to indicate milliseconds`

`ElapsedSeconds = ElapsedSeconds/Frequency; //Divide and Assign the time interval to the variable`

In brief, the `QueryPerformanceFrequency` function requests from a hardware counter (within the CPU) the total amount of ticks per second. For example, in a 4GHz CPU, the register would count about 3,900,000 to 4,000,000 ticks. This means, that for every second (period), the CPU completes that many cycles. The latter refers to the basic operations, such as accessing a memory location or writing, a CPU can perform. So, following the formula that relates period to frequency, it is calculated that the CPU in question takes about 250 nanoseconds to complete a simple operation. This number typically is measured during the initialisation (boot process) and remains unchanged until the processor is shut down. The function saves that value to the input parameter.

The `QueryPerformanceCounter`, on the other hand, collects the total amount of ticks at the time of the function call, which are registered by the Windows operating system. As a result, it is called initially to pick the total value at that point, and then it is called again to get an update. By subtracting the value of `Start` from the `End` variable, we calculate the amount of ticks added during the execution process we examine. Finally, the `ElapsedSeconds` variable is multiplied by 1000 to receive the time in milliseconds, and then divide it by the `Frequency` to get the number of milliseconds our CPU would need to perform all those ticks.

Note that the QPC is independent of external time references and does not require specialised synchronisation steps. Therefore, instead of using absolute clocks, most benchmarking approaches opt towards solutions similar to the QPC. Although there are some precision-related problems with this category of solutions, this topic is omitted as it is beyond the scope of this thesis.