THE UNIVERSITY OF HULL

Scalable Allocation of Safety Integrity Levels in Automotive Systems

being a Thesis submitted for the Degree of

Doctor of Philosophy

in the University of Hull

by

Luís Pedro da Silva Azevedo BSc., MSc.

December 2015

For Sofia and my parents…

Abstract

The allocation of safety integrity requirements is an important problem in modern safety engineering. It is necessary to find an allocation that meets system level safety integrity targets and that is simultaneously cost-effective. As safety-critical systems grow in size and complexity, the problem becomes too difficult to be solved in the context of a manual process. Although this thesis addresses the generic problem of safety integrity requirements allocation, the automotive industry is taken as an application example.

Recently, the problem has been partially addressed with the use of model-based safety analysis techniques and exact optimisation methods. However, usually, allocation cost impacts are either not directly taken into account or simple, linear cost models are considered; furthermore, given the combinatorial nature of the problem, applicability of the exact techniques to large problems is not a given. This thesis argues that it is possible to effectively and relatively efficiently solve the allocation problem using a mixture of model-based safety analysis and metaheuristic optimisation techniques. Since suitable model-based safety analysis techniques were already known at the start of this project (e.g. HiP-HOPS), the research focuses on the optimisation task.

The thesis reviews the process of safety integrity requirements allocation and presents relevant related work. Then, the state-of-the-art of metaheuristic optimisation is analysed and a series of techniques, based on Genetic Algorithms, the Particle Swarm Optimiser and Tabu Search are developed. These techniques are applied to a set of problems based on complex engineering systems considering the use of different cost functions. The most promising method is selected for investigation of performance improvements and usability enhancements. Overall, the results show the feasibility of the approach and suggest good scalability whilst also pointing towards areas for improvement.

Acknowledgements

Completing this thesis was a personally enriching and stretching experience. However, as I write these lines, I know I could not have done it on my own.

First and foremost, I would like to acknowledge my supervisor, Dr David Parker. The road was long but I could count on his support and patience all along. His experience and analytical thinking provided me with invaluable guidance to conclude this PhD. Learning from him I developed as a researcher and a professional and I will never be able to thank him enough.

Back in 2012, I travelled to Hull for my Master's final project. Unfortunately, on my arrival, my supervisor had to undergo surgery. With no obligation to do so, and at the cost of increasing his workload, Professor Yiannis Papadopoulos offered to supervise me and proposed me a new project. Looking back, this was a life changing moment. I had my first contact with Safety Engineering and, four years later, here I am writing a PhD thesis on this discipline. I will always be grateful to Yiannis for the restless help and advice and for believing in my capabilities in the first place.

I would like to express my gratitude to Dr Rui Esteves Araújo. After supervising me in my studies in Portugal, he encouraged me to pursue this PhD and has been an exceptional source of guidance and help ever since.

I am thankful to my colleagues and friends at the Computer Science department. From our discussions, I was able to improve my work. They were also an escape from the world where ASILs rule: Dr Sohag Kabir, Ioannis Sorokos, Dr Ernest Edifor, Dr Zhibao

Mian, Sahar Arshi, Ali Sa, Youcef Gheraibia, Francisco Inácio, Dr Joxe Unanue, Dr Lamis Al-Qora'n, John Dixon, John Stamford and Qian Wang.

I am deeply grateful to Dr Kreshnik Hoxha and Dr Flore-Anne Poujade, who have been a true family to me during my time in Hull. Our PhDs overlapped and we have gone through countless experiences together. I will always think about them when looking back at this period.

Thank you to my friends in Portugal, who have supported me from a distance and who have given me the most amazing times whenever I went back home: Vânia Barbosa, João Mario, Ricardo André Moreira, Carlos Correia, Ricardo Silva, Tiago Sá, Ana Moreira, Nuno Leitão and Pedro Amaro.

I am extremely thankful to my brother Martinho and his wife Sónia for always being there for me and for bringing to the world my god-daughter, Ana. Even though she does not understand it yet, Ana and her random emoticon messages were of amazing help in my least favourite days of writing. When she grows up, I will thank her properly. For now, I hope that spending time with her and a new airplane toy will do.

I will always be indebted to my parents, Basilio and Georgina, for their incredible support and encouragement all throughout my academic path. Thank you for teaching me the importance of education and hard work.

Finally, I thank my girlfriend Sofia for all the love, support and patience. She has been by my side throughout the entire journey and has been the most amazing and sweetest source of inspiration and motivation.

Contents

Figures

Tables

Glossary

AAC	ASIL Allocation Constraint
ACO	Ant Colony Optimisation
ADL	Architecture Description Language
ARP	Aerospace Recommended Practice
ASIL	Automotive Safety Integrity Level
CENELEC	European Committee for Electrotechnical Standardization
CFT	Component Fault Trees
DAL	Development Assurance Level
ECSS	European Cooperation for Space Standardization
E/E	Electrical/Electronic
EMB	Electromechanical Brake
FMEA	Failure Mode & Effects Analysis
FPTN	Failure Propagation and Transformation Notation
FSAP-NuSMV	Formal Safety Analysis Platform - New Symbolic Model Verifier
FSR	Functional Safety Requirements
FTA	Fault Tree Analysis
GA	Genetic Algorithm
HBS	Hybrid Braking System

HBSM	Hybrid Braking System Model
HiP-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies
HS	Harmony Search
HSR	Hardware Safety Requirements
IBM	International Business Machines
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IIE	Institute of Industrial Engineers
IFAC	International Federation for Automatic Control
ILP	Integer Linear Programming
IWM	In-Wheel Motor
ISO	International Organization for Standardization
LS	Local Search
MAENAD	Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles
MBSA	Model-Based Safety Analysis
MCS	Minimal Cut Set
NFT	Near Feasibility Threshold
OpenMP	Open Multi-Processing

PANDORA	Time of Priority AND gates
PSO	Particle Swarm Optimisation
QM	Quality Management
SA	Simulated Annealing
SAMD	Steepest Ascent Mildest Descent
SAE	Society of Automotive Engineers
SDMA	Steepest Descent Mildest Ascent
SG	Safety Goal
SIL	Safety Integrity Level
SMT	Satisfiability Modulus Theory
SSR	Software Safety Requirements
TDI	Total Degree of Infeasibility
TS	Tabu Search
TSDC	Tabu Search Divide and Conquer
TSR	Technical Safety Requirements
UAAP	Ultimate ASIL Allocation Problem
WNC	Wheel Node Controller

1. Introduction

1.1 Motivation

1.1.1 Developing safe systems

Our world is one that is quickly changing and rapidly adopts new technologies. There is, however, one inescapable question every corporate entity needs to ask itself before releasing a new product: “is it safe?”. The answer to this question is key to certification and market adoption of the product and is to be answered within the bounds of Safety Engineering.

Despite its vital importance, Safety Engineering has only matured as a discipline over the last 50 years (Bozzano & Villafiorita, 2010). It was initially used as an external body responsible for analysing causes of system failure after an accident, but has evolved immensely and it is now integrated with system development itself. New safety practices and safety analysis techniques have emerged, and have, throughout the years, become widespread across different industry sectors.

Granted the significant evolution, modern Safety Engineering faces important challenges. Safety-critical systems have been moving from well understood, mostly mechanical-based designs, towards complex computer-reliant architectures. The latter can perform more sophisticated control functions; their applications are immense and wide-ranging and they are a critical part of our daily lives. The vehicles we drive, for example, contain multiple programmable components which take part in safety-critical tasks such as engine control and battery management. Furthermore, computer-based systems propel advanced safety features in the vehicle. Electronic stability control systems, for instance, detect loss of steering control and automatically brake individual wheels to correct the vehicle’s trajectory.

While the merits of computer-based systems are undeniable, their inherent complexity is an obstacle for Safety Engineering. Complex systems are more difficult to design, and therefore, more likely to contain errors (Storey, 1996). Hardware architectures are composed of numerous small parts that can fail in multiple ways, giving rise to a huge number of failure scenarios to be considered. In software, the number of execution paths can easily grow into a number that is infeasible to test exhaustively.

Currently, there is a steady trend towards increasing system complexity (Bozzano & Villafiorita, 2010). The amount of time and effort that needs to be put into safety aspects opposes the companies’ competitive needs of reducing costs and of continuously introducing innovative solutions into the market. On the other hand, releasing an unsafe product can lead to accidents that endanger people’s lives, the adoption of the product and the prosperity of the company behind it.

1.1.2 Safety as part of a standardised process

International safety standards embody the state-of-the-art of Safety Engineering practices, collecting knowledge both from academia and industry. They represent common frameworks to guide practitioners when developing safety-critical systems. Compliance with safety standards serves as means to achieve designs with a low associated risk, and implicitly works to avoid liability issues. In meeting the specific characteristics of different industry sectors, different guiding documents have been released over the years. For example, ISO 26262 (Int’l Organization for Standardization,

2011) and ARP4754-A (Society of Automotive Engineers, 2010), apply to the automotive and the aerospace industry, respectively.

Currently, there is a common agreement that safety should be managed from the early stages of system design. In fact, if left to deal with at the end of development, the risk exists that safety targets will not be achieved. Also, opportunities for improvement would then entail major changes with deep impacts in development efforts and costs. The necessity of managing safety from the start of development is recognised by modern safety standards whose guidelines accompany system design right from the initial system concept specification.

Safety standards provide an important guiding framework, but practitioners struggle to fully implement them. The root causes for such difficulties are again, in great part, linked with the complexity of contemporary safety-critical systems. Given a limited availability of time and resources, effective implementation of the processes prescribed by the standards is often infeasible. One key process for which this has become more evident is the one of efficiently allocating safety integrity requirements across a system architecture.

1.1.3 The problem of allocating safety integrity requirements

IEC 61508 (Int'l Electrotechnical Commission, 1998) is a generic safety standard that served as basis for ISO 26262 and other industry-specific documents. It defines *safety integrity* as the probability of a safety-critical system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time. In contemporary safety standards, the process of elicitation and allocation of safety integrity requirements is accomplished through a concept known as Safety Integrity

Level, or SIL. Multiple SILs are considered, each implying requirements of different stringencies. Early into the development stages, investigation takes place to identify system function failures (or hazards). The risk posed by these hazards is quantified through consideration of different levels of severity. Other factors are sometimes taken into account, such as probability of occurrence. Then, each hazard is mapped with a SIL; this is to derive appropriate integrity requirements to reduce risk to an acceptable level. If a hazard is very severe, for example, it will be allocated with a SIL posing very stringent requirements. In turn, these requirements will have to be met by the system's architecture. Safety integrity requirements allocation to architectural elements is achieved on the basis of the dependability characteristics of the system. If the failure of an architectural element causes a given hazard, that element will inherit the SIL of the hazard. However, when redundancy is available, architectural elements can usually be developed according to lower SILs. SIL reduction (or *decomposition*) rules, vary from standard to standard. In the aerospace industry, for example, it is possible to allocate one component with the SIL of the hazard and the others with a SIL two times lower - see ARP4754-A (Society of Automotive Engineers, 2010). Once a SIL is allocated to a component, it will impose a series of safety activities for its development and validation.

Consider the scenario where three architectural elements (components A, B and C) must fail together to cause a system hazard. In this illustrative example three SILs are defined, relating to *low integrity*, *medium integrity* and *high integrity*. It is assumed that the system hazard can lead to a severe consequence and as a result it is associated with the *high integrity* SIL. This is a rather simple scenario, and in total there are only 27 possible allocations (three SILs for each of the three components). The allocations that follow the rule introduced in the previous paragraph for the aerospace industry are shown in Table 1.

Table 1 - Three allocations compliant with an aerospace SIL reduction rule.

Options	Component A	Component B	Component C
#1	High Integrity	Low Integrity	Low Integrity
#2	Low Integrity	High Integrity	Low Integrity
#3	Low Integrity	Low Integrity	High Integrity

Note that a valid solution would have still been found, if any of the components assigned with *low integrity* were allocated with SILs representing more stringent requirements.

SIL decomposition is not mandatory in safety standards. In principle, system developers are free to allocate all the elements of an entire architecture with the highest integrity requirements. This approach ensures that system level targets are met (or even exceeded). However, companies' budgets to deal with safety activities are not unlimited. The application of SIL decomposition is desirable, as components can be developed to less stringent requirements, leading to lower development times and efforts and in the end lower costs. The drive to minimise development costs has further implications during the allocation process. When employing the SIL decomposition rules, multiple options may be available, and here cost is usually the criterion used to select an allocation. Standards naturally focus on safety, and describe the problem of safety integrity requirements allocation as one where the goal is to find a solution which meets system level targets. However, from the practitioner point of view, meeting system integrity targets is a design constraint and the minimisation of development cost becomes a compelling objective.

Though the application of SIL allocation in small, simple architectures may be easy, modern safety-critical systems are often composed of large, networked, multi-function delivering architectures. The complexity of these systems is an obstacle. Firstly, it can be hard to understand how components' failures lead to system hazards and, as a result, how system level SILs can be allocated and decomposed across the architecture. It may not be clear, for example, that two components which are not situated within the same area of the system design, and which do not obviously interact to provide a system function, actually need to fail together to cause a system hazard. Decomposition opportunities will be missed if these relationships are not identified. Also, complex designs can hide failures that undermine redundancy assumptions, leading designers to use decomposition where it is not valid. Safety analysis techniques can be of help here, as they offer a systematic way to study the failure behaviour of a system. However, system size and complexity makes their manual application a long, difficult and error prone process.

Another concern is raised by the vast number of allocation possibilities that is usually available. In a relatively small system with only 10 components, if the three different SILs from the previous example are considered, there are 59,049 possible allocation solutions (3^{10}, three possible SIL allocations per component). Conceivably, the system designer would not enumerate all of them. However, the fast growth of solutions decreases the likelihood of finding a feasible, cost-optimal allocation. The system designer's tasks include deriving and evaluating allocations and confirming that they meet the system level requirements. Given the networked nature of modern system architectures, it is often the case that the same component is involved in assuring multiple functions and contributes to different hazards. All of these contributions need to be taken into account when validating an allocation. Performing all the tasks associated with SIL allocation quickly grows out of the realms of what is feasible to be accomplished in the context of a manual process.

The difficulty of applying SIL allocation manually is exacerbated by the iterative nature of this process. It can take multiple iterations to meet both the designer's and the suppliers' interests, for example. Also, design changes are often frequent and require SIL allocation to be re-executed. All of this raises the question: how can these processes be supported effectively and within a framework that allows for their iterative nature?

1.1.4 Automation can help

SIL allocation support needs have been presented in the previous section and can be summarised into the two following areas: 1) the study of how low-level component failures lead to system hazards – this allows rationalising how to allocate and decompose system SILs across an architecture; 2) the exploration of the huge number of allocation possibilities, to find a solution that meets system level SILs while minimising development costs.

Safety analysis techniques, namely Fault Tree Analysis (FTA), can provide an insight on how component-level failures propagate across an architecture and combine to create system hazards. They are natural candidates to support 1), especially in the context of an automated process. Tools and methods such as HiP-HOPS and FSAP-NuSMV can perform automated safety analysis. Using architectural and failure information obtained from system models, they can produce results in a matter of hours, minutes or even less.

In dealing with 2), automated allocation algorithms can help explore the vast number of solutions available. HiP-HOPS includes a smart algorithm that provides means to exhaustively explore the solution space. However, due to the rapid growth of the number of solutions available (a phenomenon known as combinatorial explosion) the technique does not scale well. Other automated allocation approaches have been presented in the literature, but they either do not directly consider the matter of costs in the allocation process, or contemplate simple models where development costs grow linearly with an increase to the SIL level to follow. A method is required which is capable of being applied to large, complex systems and that can accommodate user-defined functions to describe ASIL-imposed costs.

1.1.5 Exploring automatic optimisation algorithms

In dealing with large combinatorial problems, exhaustive allocation algorithms often require prohibitive processing effort. Automatic optimisation algorithms are employed instead. Many optimisation algorithms are available and they can be divided into two groups: exact methods and metaheuristics. Exact methods include techniques such as Integer Programming and discrete Lagrangian methods. Their main advantage is that where solutions are found, they are optimal. However, for large-scale systems the computation burden can be excessive; moreover, these techniques are often associated with mathematical properties of the problem.

Metaheuristics, on the other hand, cannot guarantee finding optimal solutions. Nonetheless, they are known to be efficient methods for retrieving good solutions in complex, high-dimensional problems. Furthermore, they present robustness to be applied to problems of different characteristics, which can be useful in dealing with different ASIL-dependent cost functions defined by the user. Metaheuristics are often inspired by natural phenomena, and include techniques such as Genetic Algorithms, which mimic the processes of natural evolution, and the Particle Swarm Optimiser that intends to reproduce the movement of flocks of birds in their search for food.

This thesis argues that SIL allocation can be supported effectively and efficiently in complex system architectures using a mixture of model-based safety analysis and metaheuristics optimisation algorithms.

1.1.6 Focusing on the automotive industry

Road safety is a major concern of modern society and some important targets have been defined internationally in regards to mortality and disability due to car crashes. In Europe, for example, the European Commission has targeted a reduction of 50% by the year of 2020 (versus 2010) (European Commission, 2015). Automotive companies help compliance with the targets by continuously integrating new safety functionalities in their vehicles and what many of these new features have in common is the integration of electronics and software.

Increasing system complexity is a trend across industry sectors and especially on the automotive domain. Six years ago, Charette (2009) published an article with a catchy name: “This Car Runs on Code”. Illustrating the issue of relative complexity, the article reveals that a premium car back then would have probably included an astonishing number of 100 million lines of code (versus, for example, the 6.5 million lines of code in Boeing’s 787 that was being delivered to customers in 2010). The same article also mentions a study that predicted that cars would include 200 to 300 million lines of code in the near future!

Whilst this thesis addresses the generic problem of "safety integrity requirements allocation", the automotive industry will be taken as an application example. In particular, the SIL decomposition algebra defined by ISO 26262 will be followed and automotive case studies will be analysed. In this industry sector, SILs take the name of Automotive SILs (or ASILs).

1.2 Research hypothesis

The allocation of Automotive Safety Integrity Levels in large and/or complex architectures can be efficiently supported through automated model-based safety analysis and the optimisation of development costs with metaheuristics.

1.3 Research goal and objectives

ASIL allocation impacts deeply on the development of safety-critical systems. It is important to find allocations that meet safety integrity requirements and that simultaneously minimise development costs. Given the complexity of modern safety-critical systems, finding these allocations clearly demands automated and efficient support. In the state-of-the-art of model-based safety analysis, methods and tools have been identified which can be applied to large architectures in very acceptable times. They provide the system failure information that allows rationalising which ASIL allocations are valid. Since suitable techniques are already available at the start of this project, research will not focus on the development or enhancement of model-based safety analysis. The goal of the thesis is the development of an efficient metaheuristic optimisation method that can find valid, cost-effective ASIL allocations. The technique, merged with state-of-the-art model-based safety analysis tools, shall provide an overall automated approach capable of supporting ASIL allocation in the context of complex systems.

Attaining the thesis goal requires meeting several objectives:

1. **Investigate, amongst the state-of-the-art in metaheuristic optimisation, a set of techniques that can be applied to solve the novel problem of scalable allocation of ASILs.**

ASIL allocation poses a complex combinatorial problem capable of generating solution spaces that are well beyond the realms of exhaustive evaluation. Metaheuristics have been successfully applied to countless combinatorial problems, and are known to be able to provide (near) optimal solutions within efficient time spans. There is, however, a wide range of metaheuristic optimisation techniques available, and there is no one technique that outperforms all others for every problem. Furthermore, metaheuristics are generic optimisation strategies, which need to be adapted to the problem at hand. Popular metaheuristics, and the state-of-the-art of their application, will be investigated to derive effective techniques to tackle the ASIL allocation problem. Although multiple techniques will be created, it would be important to point system designers to the most promising one. Consequently, the second objective is:

2. **Establish a set of tests and metrics that allow comparison of the performance of the metaheuristic techniques being investigated.**

The problem of finding cost-efficient ASIL allocations is a new one. It is important to define tests and metrics that allow evaluation and comparison of techniques developed to tackle it. Fulfilling this objective will enable choosing, amongst the multiple techniques investigated in the context of objective 1, the most promising to carry the optimisation task. Once a technique is chosen, it would be desirable to study it in more detail to try and enhance further its performance and usability. The third objective of this thesis is therefore to:

3. **Select the best performing metaheuristic and examine the possibility of improving it further and enhance its usability.**

Any improvement that can be made to the optimisation technique will be important, both as means to achieve better cost-effective solutions, and/or to improve scalability to larger and more complex problems. Examining opportunities for faster performance is also very relevant in the light of the iterative nature of ASIL allocation.

A set of guidelines for the usage of the algorithm are needed. These would minimise the effort spent by the system designer when using the optimisation technique, and as a result he would be able to spend more time analysing the allocation results.

4. **Apply the developed approach to case studies in order to validate its applicability and usefulness to the development of complex engineering systems.**

In comparing the different optimisation techniques, it is important to evaluate their performance in the context of problems from complex engineering systems. A benchmark of problems generated from automotive safety-critical systems will be created in the context of objective 2. The optimisation algorithms developed will be applied to this benchmark. Furthermore, the case study of a hybrid braking system for electrical vehicles will be used to illustrate the usefulness of the approach.

5. Explore the impacts of different cost heuristics in the allocation of ASILs.

Safety integrity requirements allocation has been analysed in the literature as a problem where the sum of SILs allocated across a system architecture are to be minimised. These approaches are implicitly assuming that development costs grow linearly with the SIL allocated to a component. However, in industry, a variety of SIL-dependent cost functions have been advanced. The impacts of using different cost functions in defining what solutions are seen as optimal will be analysed.

1.4 Thesis structure

An outlook on the remaining chapters that compose the thesis is provided now.

2. Background

Chapter 2 frames the research by presenting foundation concepts and processes, as well as a review of relevant literature. The chapter initially provides an overview of how safety integrity requirements allocation is generically guided across industries, then providing further detail on the process prescribed by the ISO 26262 standard for the passenger car sector. A review of methods and tools providing support for the allocation of safety integrity requirements is presented towards the end of the chapter.

3. Discovering cost efficient ASIL allocation strategies

Chapter 3 starts by formulating the problem of finding valid ASIL allocation solutions that minimise system development costs. A discussion on optimisation research areas is subsequently introduced and the chapter then converges to describing a series of sophisticated metaheuristic techniques to carry out ASIL allocation (objective

1). The techniques are based on the popular metaheuristics: Genetic Algorithms, the Particle Swarm Optimiser and Tabu Search.

4. Case studies and evaluation metrics

Chapter 4 presents a set of optimisation problems and evaluation metrics to enable comparison of different optimisation techniques to conduct ASIL allocation (objective 2). A case study of a hybrid braking system is used to evaluate the applicability and usefulness of the automated approach for ASIL allocation (objective 4). The same case study is then utilised to explore the impacts of different cost heuristics in defining the optimal solution for a problem (objective 5).

5. Experimentation and evaluation

Chapter 5 addresses objective 3 of the thesis. Initially, the metaheuristic optimisation techniques developed in Chapter 3 are compared. The best performing is then taken, and a series of strategies to improve its performance further are evaluated. Investigation towards deriving guidelines for the parameterisation of the algorithm is also shown. Finally, the impacts of considering different cost functions for ASIL allocation are further explored (objective 5).

6. Conclusions

Chapter 6 is the conclusion chapter of the thesis. It enumerates the contributions made in the context of the objectives defined and it describes the impact on external research.

7. Limitations and suggestions for future work

Chapter 7 is the final chapter and it discusses limitations, areas for improvements and relevant paths for future work that are left to pursue at the end of this thesis.

1.5 Summary of contributions

This was the first work to investigate the impacts of user-defined cost functions when deciding on an ASIL allocation for a system architecture. The use of different cost functions have determined different optimal solutions for an automotive hybrid braking system. This investigation points for the need of work in industry to derive most accurate cost information, so that decision support techniques can provide the system designer with solutions which are closer to the "true" optimum. To enable support for ASIL allocation in complex architectures and across user defined cost functions, the research successfully combined, for the first time, an automated safety analysis technique with metaheuristic optimisation methods. Furthermore, in order to allow the evaluation and comparison of different optimisation methods, an ASIL allocation benchmark was created. Three metaheuristic-based techniques were developed, which can successfully solve the entire benchmark to optimality. Tabu Search was the fastest technique across the benchmark and its performance was subsequently improved further, namely through parallelisation. The technique's usability was also enhanced.

1.6 Publications

Part of the work presented in this thesis has been published in the context of multiple conferences and in one edition of the IEEE Software magazine. Azevedo *et al.* (2014b) describes an approach for the automatic allocation of Automotive Safety Integrity Levels based on HiP-HOPS automated Fault Tree Analysis capabilities and it shows its application to a hybrid braking system for electric vehicle integration. Azevedo *et al*. (2014a) explores the impacts of using different cost heuristics in defining the optimal allocation solutions in the hybrid braking system; the same article formulates ASIL allocation as a constrained cost optimisation problem. Parker *et al*. (2013) and Azevedo *et al*. (2013) show two metaheuristic-based techniques to find ASIL allocations that minimise system development costs. The first presents a Genetic Algorithm approach and the second a Tabu Search method.

2. Background

This chapter introduces the reader to the context of the research project. The concepts of functional safety and safety integrity levels are introduced and the standardised automotive process for top-down safety integrity requirements allocation is analysed. The current support for SIL allocation is examined, setting the scene for the work developed in this thesis.

2.1 Functional Safety and Safety Integrity Levels

When there is no unreasonable risk caused by a malfunctioning behaviour of Electrical/Electronic (E/E) systems, it is said that functional safety has been achieved (Int'l Organization for Standardization, 2011). Functional safety standards provide guidance for the development of safety-critical systems that include E/E components. IEC 61508, entitled "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems", was born within the process control industry and has set the grounds for modern functional safety standards. There have been two editions of IEC 61508 (Int'l Electrotechnical Commission, 1998; Int'l Electrotechnical Comission, 2010). Baufreton *et al*. (2010) provide an overview of safety standards history; the authors state that IEC 61508 was created with the intent of becoming a generic safety standard from which industry-targeted documents would be derived. The reality is, however, that the standard is applied as a general guiding document across different industries. Nevertheless, sector-targeted documents have also been introduced to cover needs for more specific guidance. A few examples of functional safety standards and the industry sector to which they apply are provided next.

- IEC 61511: Applies to the process industry

  (Int'l Electrotechnical Commission, 2003)
- IEC 61513: Applies to nuclear power plants

  (Int'l Electrotechnical Commission, 2011)
- EN 50126, EN 50126 and EN 50129: Apply to the railway industry

  (European Committee for Eletrotechnical Standardization, 1997, 2001, 2003)
- ARP4754-A: Applies to the aerospace industry

  (Society of Automotive Engineers, 2010)
- ISO 26262: Applies to the automotive industry

  (Int'l Organization for Standardization, 2011)

As introduced in Chapter 1, *safety integrity* refers to the probability of a safety-critical system satisfactorily performing the required safety functions. When dealing with random hardware failures, which are due to physical factors such as corrosion and wear out, safety standards prescribe safety integrity requirements in the form of targets for the maximum probability of failure. Baufreton *et al*. (2010) have studied safety standards from six industry sectors; they report that in dealing with systematic failures (i.e., failures introduced by errors in specification, design or implementation) none of the documents "gives credit to probabilistic assessment". Instead, safety standards use a process based on prescribing a range of activities to be undertaken during the development and validation of architectural elements. The process is enabled through a concept generally known as the Safety Integrity Level or SIL.

In the early stages of development, hazards, i.e. potential sources of physical injury caused by system malfunctions, are identified and their associated risk is estimated on the basis of some parameters (e.g. hazard’s severity of potential harm). Risk is characterised with a SIL. Usually there are five SILs (e.g. SIL 0 to SIL 4, in IEC 61508); the different levels serve to impose sufficient requirements to minimise the risk of a given hazard, reducing it to an acceptable, residual level. In a risk assessment procedure where severity is the only parameter to take into account, a life threatening hazard will be assigned with a high SIL and, as a consequence, strict requirements will apply. The SIL of the hazard is allocated top-down to architectural elements that can cause that system hazard. A high SIL imposes stringent development and validation requirements for the components it is allocated to.

The SIL was first introduced by the UK Health & Safety Executive Guidelines, and subsequently adopted by functional safety standards (Papadopoulos *et al.*, 2010). SILs take different names depending on the sector of application:

- IEC 61508 and EN 5012x: Safety Integrity Levels (SILs)
- ARP4754-A: Development Assurance Levels (DALs)
- ISO 26262: Automotive Safety Integrity Levels (ASILs)

The research project focuses on the automotive industry, which is governed by the processes from ISO 26262 “Road Vehicles – Functional Safety”. In the next section, an overview of this standard will initially be provided; then, the approach it prescribes for integrity requirements distribution will be presented.

2.2 Distributing safety integrity requirements in automotive systems

2.2.1 Overviewing ISO 26262

The development of ISO 26262's first edition was a 4 year-long project, which started in July of 2007 and was concluded in November of 2011. Before ISO 26262, the application of IEC 61508 was sometimes found, but practitioners often experienced difficulties in applying a standard that was not tailored to the automotive sector (Baufreton *et al.*, 2010). ISO 26262 specifically applies to E/E systems of series production cars with a maximum gross weight of 3500 kg. The standard includes 10 parts:

1. *Vocabulary*

   Defines all the terms used throughout the standard.

2. *Management of functional safety*

   Specifies project-independent safety management requirements for organisations, including requirements for safety culture implementation and competence management. It also defines project-specific requirements for the management of safety activities to be undertaken during each stage of the safety lifecycle.

3. *Concept phase*

   Specifies the requirements for item definition, initiation of the safety lifecycle, hazard analysis and risk assessment, and the functional safety concept.

4. *Product development at the system level*

   Specifies requirements for development at the system level, including safety requirements specification, design, verification, integration, testing and release for production.

5. *Product development at the hardware level*

   Specifies requirements for development at the hardware level, including safety requirements specification, design, verification, integration and testing.

6. *Product development at the software level*

   Specifies requirements for development at the software level including specification and verification of safety requirements, testing and integration.

7. *Production and operation*

   Specifies requirements for production, operation, service and decommissioning of the item.

8. *Supporting processes*

   Specifies requirements for supporting processes such as interfaces with distributed developments, change management and documentation.

9. *ASIL-oriented and safety-oriented analysis*

   Specifies requirements for the application of ASIL decomposition and for the execution of safety analysis.

10. *Guideline on ISO 26262*

    Provides an overview of the standard and examples of key processes for better understanding.

ISO 26262 establishes a complete automotive safety lifecycle, covering all stages of the vehicle's lifetime: from item definition, to development and all the way to decommissioning. In the context of ISO 26262, *item* stands for a system or array of systems to which the standard's process is going to be applied. An overview of ISO 26262's safety lifecycle is presented in Figure 1. It can be divided in to three main stages.

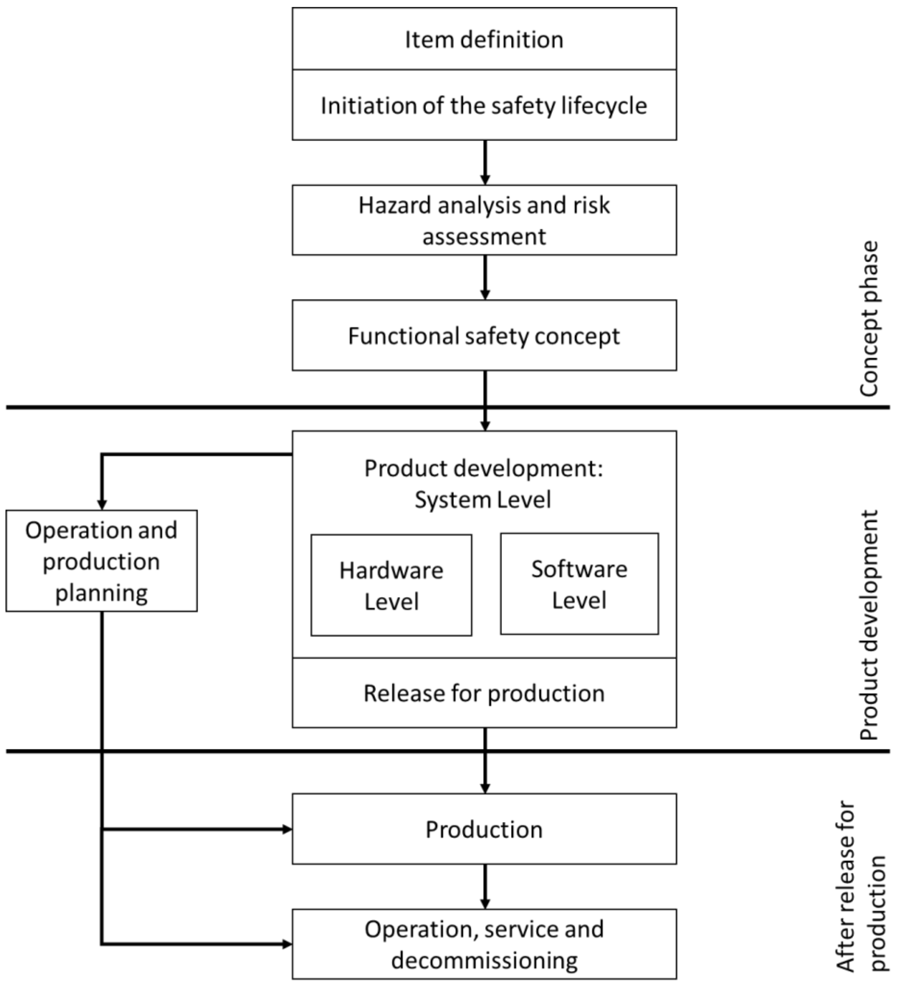
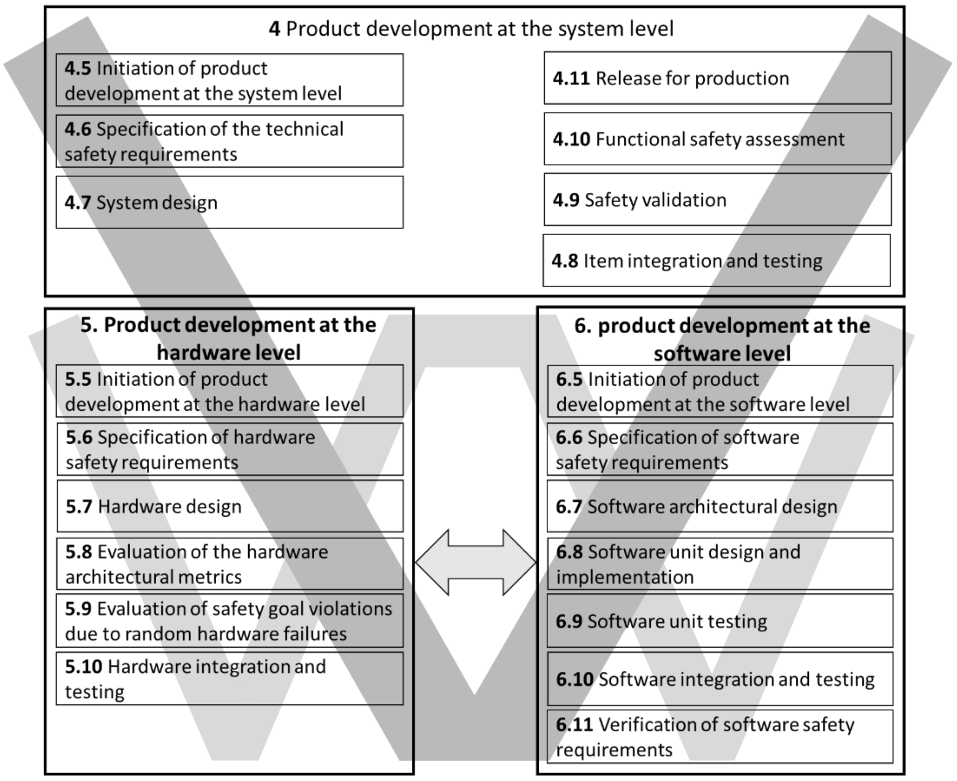

Item definition
Initiation of the safety lifecycle
Hazard analysis and risk assessment
Functional safety concept
Concept phase
Product development: System Level
Hardware Level
Software Level
Release for production
Operation and production planning
Product development
Production
Operation, service and decommissioning
After release for production

Figure 1 - Simplified safety lifecycle of ISO 26262.

At the *concept phase*, the item is to be defined and for the initiation of the safety lifecycle it should be clarified if the item is a new development or a modification to a pre-existing system. The risk associated with the item’s hazards is to be calculated and ASILs are to be determined to impose appropriate integrity requirements. A functional safety concept shall be specified which includes a set of safety requirements and a preliminary system architecture to address the identified hazards.

The various parts of the *product development* phase (system, software and hardware development) follow a V-model as a reference process model (see Figure 2). Requirements specification, design and implementation activities refer to the left-hand branches, whereas integration, testing, verification and validation activities belong to the right hand side branches.



4 Product development at the system level
4.5 Initiation of product development at the system level
4.6 Specification of the technical safety requirements
4.7 System design
4.11 Release for production
4.10 Functional safety assessment
4.9 Safety validation
4.8 Item integration and testing
5. Product development at the hardware level
5.5 Initiation of product development at the hardware level
5.6 Specification of hardware safety requirements
5.7 Hardware design
5.8 Evaluation of the hardware architectural metrics
5.9 Evaluation of safety goal violations due to random hardware failures
5.10 Hardware integration and testing
6. product development at the software level
6.5 Initiation of product development at the software level
6.6 Specification of software safety requirements
6.7 Software architectural design
6.8 Software unit design and implementation
6.9 Software unit testing
6.10 Software integration and testing
6.11 Verification of software safety requirements

Figure 2 – ISO 26262's V-model development process for system, hardware and software. The numbers showing in an "m-n" format point to the nth clause of the mth standard part (Int'l Organization for Standardization, 2011).

The last stage of the product development phase is the release for production step, which confirms that the item complies with the requirements for functional safety at the vehicle level and that it is ready for series-production and operation.

After release for production, the standard addresses production planning and control. These activities shall consider, for example, safety-related special characteristics of the item that have been determined during the development phases. A temperature range might have to be respected during the production process, for instance. Finally, requirements are specified for the development and management of the user manual as well as instructions for maintenance and repair, and lastly disassembly.

2.2.2 ASILs: From Hazard Analysis to allocation and decomposition

The standard uses a risk-based approach, specific to the automotive domain, to assign ASILs to hazardous events and through those ASILs it specifies requirements for avoiding unreasonable risk. This process will be reviewed next.

At the concept phase, hazards are identified and they are linked with the operational situations in which they may occur (e.g. weather and road layout). The combination of a hazard and an operational situation gives a *hazardous event*. A risk assessment process then takes place through the estimation of the *severity*, *exposure* and *controllability* of each hazardous event.

Severity of potential harm (S) refers to the degree of physical damage caused to each endangered individual, including the driver, the passengers and other traffic participants. Table 2 shows the classes of severity described in the standard.

Table 2 - Severity of potential harm levels (Int'l Organization for Standardization, 2011:Part 3).

S0	S1	S2	S3
No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-Threatening (survival uncertain), fatal injuries.

Probability of exposure (E) refers to how frequently and for how long individuals are exposed to the hazardous event. Table 3 shows the different classes of probability of exposure.

Table 3 - Probability of exposure levels (Int'l Organization for Standardization, 2011:Part 3).

E0	E1	E2	E3	E4
Incredible	Very low probability	Low probability	Medium probability	High Probability

Controllability (C) refers to the ability of the driver or other traffic participants to gain control of the hazardous situation in order to avoid the harm. There are four classes of controllability, as seen in Table 4.

Table 4 - Controllability levels (Int'l Organization for Standardization, 2011:Part 3).

C0	C1	C2	C3
Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable

The combination of the three factors maps an ASIL to each hazardous event (see Table 5). Each ASIL implies integrity requirements of different stringencies. The standard defines five ASILs. They range from A (least stringent requirements) to D (most stringent requirements). Furthermore, when no special safety-related requirements are to be applied, the level QM is assigned; QM redirects development to Quality Management.

Table 5 - ASIL to hazardous event mapping as a function of severity, exposure and controllability (Int'l Organization for Standardization, 2011:Part 3).

Severity class	Probability class	Controllability 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

A life threatening hazardous event, which is highly likely to occur and that is difficult to control is assigned with the most stringent level (ASIL D).

After characterising a hazardous event with an ASIL, the next step is to define Safety Goals (SGs). They are very high level safety requirements related to the prevention or mitigation of hazardous events. The standard uses the example of an airbag system to illustrate the notion of SG. In the example it provides, the concept of a *hazardous event* is bypassed, and only a hazard associated with the system is presented. Possibly, this has been done to simplify the example. One potential hazard associated with the airbag system is then: “unintended deployment”. The following SG can be formulated: “ensure that the airbag does not deploy, unless a crash occurs that requires deployment” (Int’l Organization for Standardization, 2011:Part 10). SGs inherit the ASILs of the hazardous events they are meant to avert.

The safety integrity requirements for random hardware failures are defined at the SG level. The standard defines two metrics to evaluate the robustness of the hardware architecture involved in achieving an SG to 1) single-point[1] and residual[2] faults, and 2) to latent faults[3]. Furthermore, two alternative methods are established to assess if the residual risk of SG violation due to random hardware failures is sufficiently low. One is based on a global probabilistic evaluation of the hardware involved in assuring the SG and the other independently examines the impact of each fault identified as a possible cause of SG violation. For all four evaluation methods that have been enumerated,

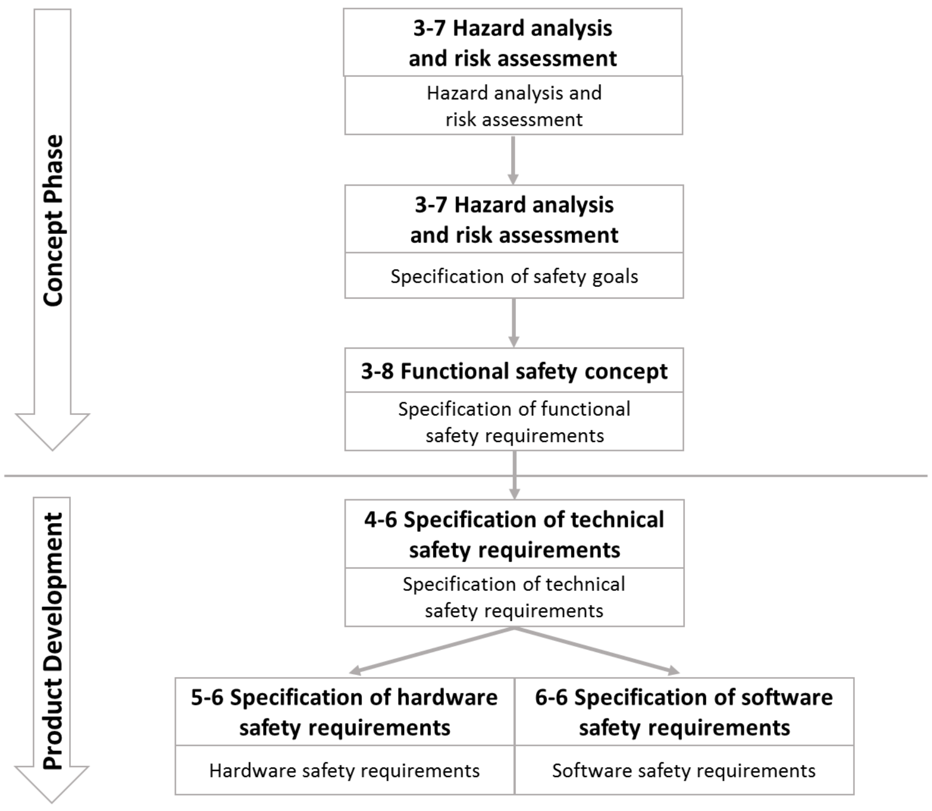
[1] This type of fault directly violates a safety goal; no mechanism is implemented to control faults in the respective hardware element.

[2] This type of fault directly violates a safety goal; at least one mechanism is implemented to control faults in the respective hardware element.

[3] This type of fault violates a safety goal only in conjunction with another independent fault; this fault is not detected any mechanism or perceived by the driver.

targets are defined as a function of the SG's ASIL – generally, the higher the ASIL, the more stringent targets that must be met.

ISO 26262 defines a top-down process for safety requirements specification, which starts from the SG concept (see Figure 3).



Concept Phase
3-7 Hazard analysis and risk assessment
Hazard analysis and risk assessment
3-7 Hazard analysis and risk assessment
Specification of safety goals
3-8 Functional safety concept
Specification of functional safety requirements
Product Development
4-6 Specification of technical safety requirements
Specification of technical safety requirements
5-6 Specification of hardware safety requirements
Hardware safety requirements
6-6 Specification of software safety requirements
Software safety requirements

Figure 3 - Structure of safety requirements. The numbers showing in an "m-n" format point to the nth clause of the mth standard part (Int'l Organization for Standardization, 2011:Part 3).

Functional Safety Requirements (FSR) are to be defined from SGs. FSRs should specify implementation-independent safety behaviour to achieve an SG. Revisiting the airbag example introduced earlier, at this stage it can be specified that there must be a redundant function to detect if a collision has occurred. Next, Technical Safety Requirements (TSRs) are formulated from FSRs. TSRs detail the implementation of the FSRs at the system level. Finally, Hardware Safety Requirements (HSRs) and Software Safety Requirements (SSRs) are to be derived from the TSRs.

As safety requirements are being formulated top-down, they inherit the ASIL of the upper level safety requirement they are derived from. Safety requirements and their respective ASILs are allocated to architectural elements of the current design iteration - the process follows the hierarchical definition of the system, from preliminary architecture, down to software and hardware. The process is illustrated in Figure 4.

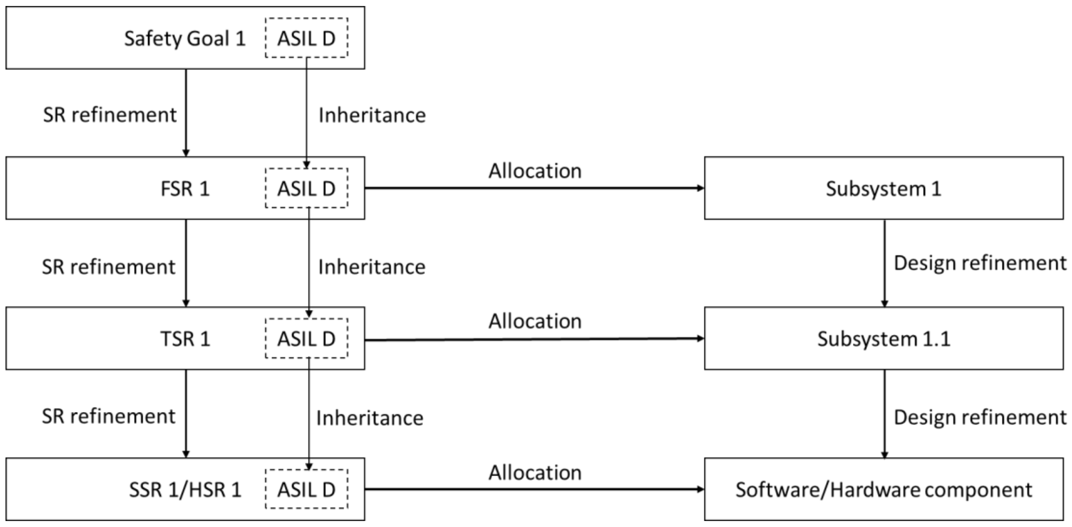


Figure 4 - Safety Requirements (SR) hierarchical inheritance of ASILs and allocation of SRs and respective ASILs to the elements of the system architecture. Only one SR and one architectural element are shown at each refinement step, but multiple may exist.

An architectural element that when failing leads to the violation of an ASIL B SG, will itself be allocated with ASIL B, as a result of the hierarchical safety requirement ASIL inheritance. The ASIL allocated to an architectural element defines the safety integrity requirements to deal with systematic failures that need to be fulfilled by that element. These include specific development and verification activities to be undertaken. Higher ASILs impose more stringent activities.

When architectural decisions lead to the existence of sufficiently independent architectural elements, safety requirements can be implemented redundantly by the independent elements. In these scenarios, the standard allows the allocation of lower

ASILs to the decomposed safety requirements. In turn, this means that the redundant architectural elements can be developed according to lower ASILs than the one of the SG they are addressing. This process is coined in the standard as *ASIL decomposition*. The decomposition schemes allowed can be observed in Figure 5. ASIL decomposition can be applied to the functional, technical, hardware or software requirements.

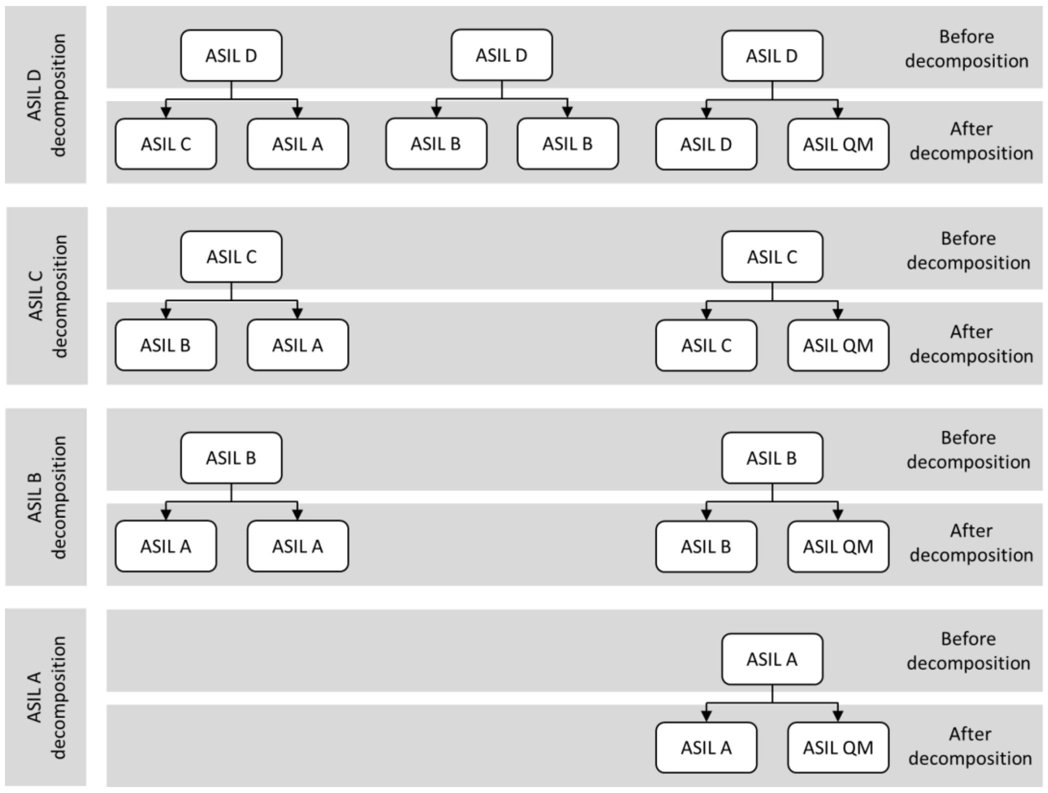


Figure 5 - ASIL decomposition schemes (Int'l Organization for Standardization, 2011:Part 9).

ASIL decomposition can be performed more than once to the same safety requirement according to the patterns of Figure 5. The standard says that, "in general, ASIL decomposition allows the apportioning of the ASIL of a safety requirement between several elements (…)" (Int'l Organization for Standardization, 2011:Part 9). The process can be formalised according to an *ASIL algebra* (Papadopoulos *et al.*, 2010; Mader *et al.*, 2012), which is based on assigning the different ASILs with integer numbers: ASIL QM -

0; ASIL A - 1; ASIL B - 2; ASIL C - 3; ASIL D - 4. If only the failure of i elements jointly violate an SG, Equation 1 applies.

$$\sum_{j=1}^{i} ASIL_{element_j} \geq ASIL_{SG} \qquad \textbf{Equation 1}$$

ASIL decomposition is desirable for developers of systems as it allows the meeting of a potentially high ASIL of an SG, without having to develop all components of the system architecture according to that high ASIL.

ASIL decomposition concerns systematic failures only, and therefore requirements imposed to random hardware failures are not affected by the technique.

2.2.3 ASIL decomposition example

To better demonstrate ASIL decomposition, ISO 26262 includes an illustrative example (Int'l Organization for Standardization, 2011:Part 10). That example is visited in this section.

A system has an actuator that is triggered by the driver's action on a dashboard switch. The actuator provides a comfort function, which is assumed to be dangerous if activated above 15 km/h. A Hazardous Event 1 (HE1) is then established as: "the activation of the actuator while driving at a speed above 15 km/h, with or without a driver request". Through risk assessment HE1 is assigned with ASIL C. A Safety Goal 1 (SG1) is then defined: "the actuator shall not be activated while the vehicle speed is above 15 km/h". SG1 inherits the ASIL of HE1 (ASIL C). A preliminary architecture of the system is created (see Figure 6).

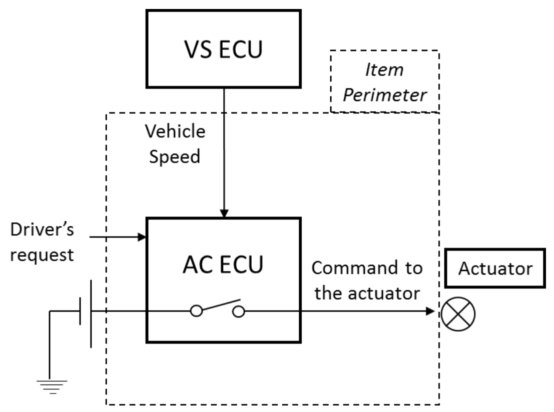


Figure 6 - Preliminary architecture of illustrative item for ASIL decomposition (Int'l Organization for Standardization, 2011:Part 10).

The driver's request as well as the vehicle speed are inputted to the Actuator ECU (AC ECU); the latter can activate or deactivate the actuator. The Vehicle Speed (VS) measurement is made in the VS ECU. Three safety requirements are defined for the system and inherit the ASIL of SG1.

- **SR1**: The VS ECU shall send the accurate vehicle speed information to the AC ECU
- **SR2**: The AC ECU shall power the actuator only if the vehicle speed is below 15 km/h
- **SR3**: The actuator shall only be activated when r by the AC ECU

The developers chose to introduce a redundant element to deal with powering the actuator. The evolved architecture can be observed in Figure 7.

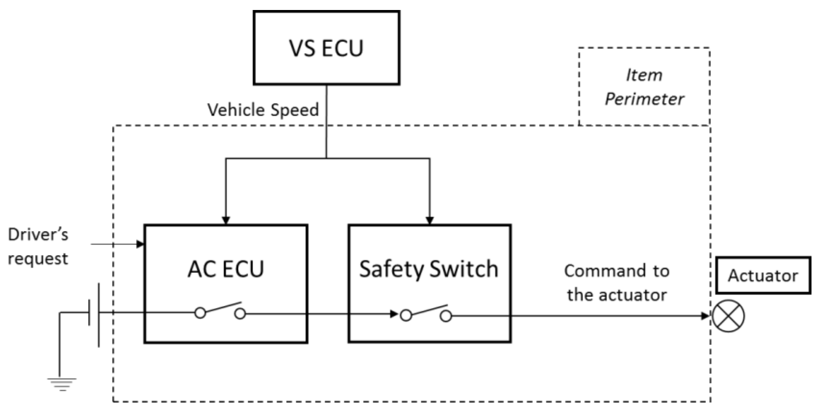


Figure 7 - Second design iteration of illustrative item for ASIL decomposition (Int'l Organization for Standardization, 2011:Part 10).

A safety switch has been introduced, so that in case of AC ECU failure the actuator can be disabled at speeds above 15 km/h. SR2 and a new SR4 address the powering of the actuator:

- **SR4:** The switch shall be in an open state if the vehicle speed is above 15 km/h.

SR2 and SR4 redundantly implement SG1. Only the failures of both the AC ECU and the safety switch would cause the violation of the SG and therefore ASIL decomposition can be applied. The decomposition possibilities are presented in Table 6.

Table 6 - Possible ASIL allocations for SR2 and SR4 (and respective components).

Allocation Possibilities	Requirement SR2 (AC ECU)	Requirement SR4 (Safety Switch)	SG1
#1	C (3)	QM (0)	ASIL C (3)
#2	B (2)	A (1)	
#3	A (1)	B (2)	
#4	QM (0)	C (3)	

Failure to meet either SR1 or SR3 directly results in the violation of the safety goal. These two requirements (and their respective architectural elements) receive the ASIL of SG1.

2.3 Decomposing SILs in the aerospace industry

Several functional safety standards share similar concepts of SILs, but the majority does not provide guidance for SIL reduction when in the presence of functional or architectural redundancy (Blanquart *et al.*, 2012). However, some of the latest guiding documents have introduced specific rules to allow such downgrading. This is the case of ISO 26262 (released in 2011), as shown already in this thesis, and of ARP4754-A for the aerospace domain (released in 2010).

Development Assurance Levels (DALs) are the SILs for the aerospace industry. As with the automotive case, five levels are identified: DALs range from A to E, but run opposite to ASILs with A implying the most stringent requirements. In regards to how DALs are distributed across a system architecture, the process starts with DALs being assigned to system failures. This assignment is based on the risk associated with the system failure. In the ARP4754-A, system failures are termed *failure conditions* and risk is estimated solely on the basis of severity (see Table 7).

Table 7 - ARP4754-A: Failure condition classes and DALs.

Failure Condition Severity	**DAL**
Catastrophic	A
Hazardous	B
Major	C
Minor	D
No Safety Effect	E

The process then continues with DALs being allocated to functions and afterwards inherited by the subsystems and components that implement them. In regards to DALs decomposition, ARP4754-A gives two options:

- **Option 1:** Allocate one element with the original DAL and downgrade the remaining elements two levels at the most
- **Option 2:** Downgrade by one level the DAL of a pair of elements and reduce the remaining elements by two levels at the most

Figure 8 gives the DAL decomposition schemes for three components which must fail together to cause a failure condition. The alternatives shown take maximum permissiveness of the DAL reduction options, i.e. “the remaining elements” are allocated with two lower levels. The latter only does not hold true for DAL D decompositions, since there is only one lower level (DAL E).

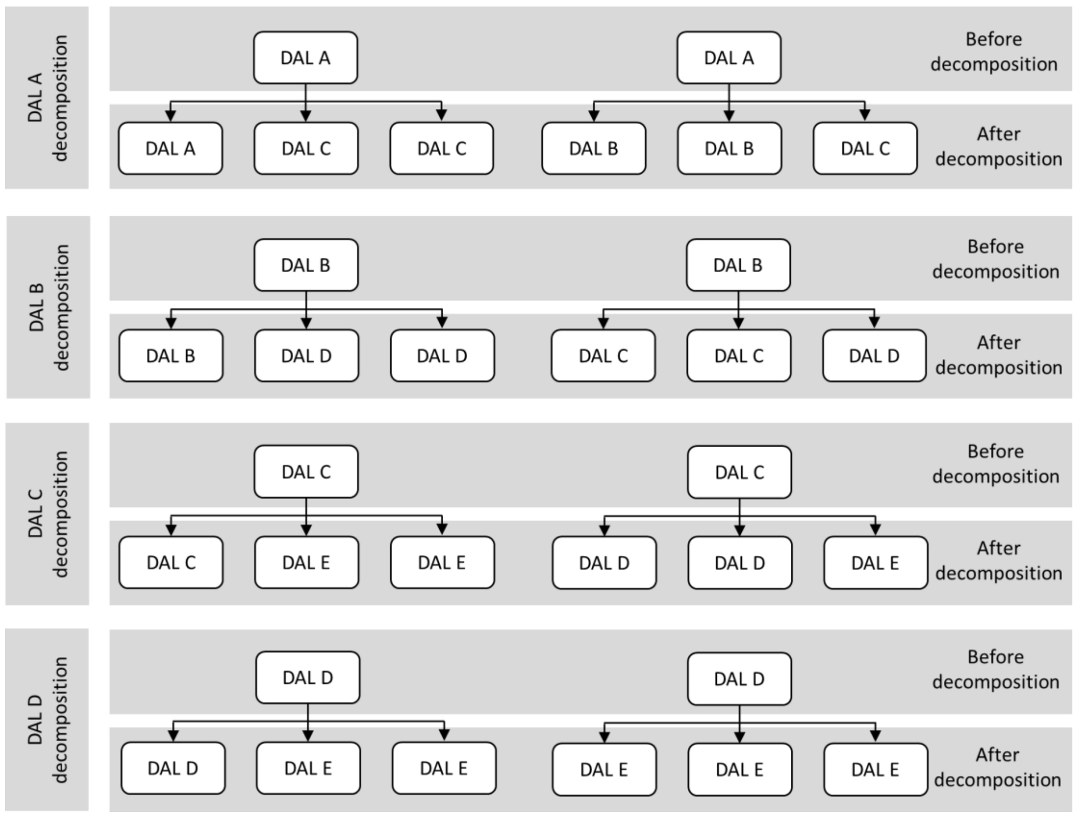


Figure 8 - DAL decomposition schemes for three components.

2.4 SIL allocation - costs matter

SIL allocation is presented in ISO 26262, and other in other safety standards, as a problem where the goal is to fulfil a set of constraints, ensuring that system level requirements are being met. Naturally, standards are focused on safety, and cost considerations are not really taken into account. As argued in Azevedo *et al*. (2014a), there are unexplored opportunities if the problem is approached in this way. Complying with a given ASIL means that a set of activities needs to be performed during development and validation. They are reflected in the end as costs, and vary with the specific ASIL prescribed to a component. Many allocation possibilities may potentially exist, and in order to choose the most advantageous solutions, one needs to contemplate their different cost implications.

SIL decomposition is in the first instance a technique that allows decreasing development costs, as where it is applied, components do not individually need to entirely fulfil the SIL of the SG they are assuring. However, the cost differences between the different SILs can reveal decomposition strategies that are more promising than others. At time of design, exact development costs can be hard to obtain. However, deliberations can still be made on the basis of some heuristic, which indicates the cost jumps between complying with the various ASILs. For example, if considering a logarithmic cost increase between integrity levels (ASIL QM = 0; ASIL A = 10; ASIL B = 100; ASIL C = 1000; ASIL D = 10000), when decomposing an ASIL B amongst two components, C1 and C2, one single optimal solution is revealed (see in bold).

- C1 (ASIL QM) + C2 (ASIL B): 0 + 100 = 100 cost units
- **C1 (ASIL A) + C2 (ASIL A): 10 + 10 = 20 cost units**
- C1 (ASIL B) + C2 (ASIL QM): 100 + 0 = 100 cost units

The example above should not mislead the reader into trusting that this is a trivial problem where each decomposition opportunity can be examined independently. Components are often participating in assuring multiple safety goals and numerous chains of conflicting constraints must be examined. This is clearly a complex combinatorial problem where the paradigm is constrained optimisation of costs, rather than satisfaction of safety integrity constraints.

2.5 Supporting ASIL allocation

When performing ASIL allocation, the complexity associated with modern safety-critical architectures is a concern. Systems are often composed of a large number of components. Furthermore, they deliver multiple functions, which can be interdependent and share elements. As identified in chapter 1, there are two main challenges in allocating ASILs in these types of architectures: 1) studying the complex failure behaviour of the system to rationalise what allocations are valid and 2) efficiently explore the huge number of ASIL allocation options available and determine a cost-effective solution that meets allocation constraints. This section provides an overview of current work towards meeting these challenges and establishes the basis for the approach developed in the thesis.

2.5.1 Safety analysis to the rescue!

Safety analyses are systematic techniques that can help with understanding the failure behaviour of a system, and how an architecture can be improved given the identified safety flaws, or shortcomings. Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) are two of the most popular safety analysis techniques. They have been widely used across industries (e.g. nuclear, automotive, aerospace) contributing significantly to making modern systems safer.

FMEA was first introduced in the 1940s (U.S. Department of Defense, 1949). When performing an FMEA, ways in which a component can fail, i.e. failure modes, are identified and their impacts on the rest of the architecture are evaluated up to the system level. FMEA works bottom up, falling into the category of inductive safety analysis. Risks associated with the effects of each of the failure modes are calculated on the basis of a quantification of their severity, occurrence and detection. Risk estimation allows the prioritisation of the failure modes that need correction (McDermott *et al.*, 1996). The results are presented in tables, often of large sizes.

FTA dates back to the 1960s (Watson, 1961) and is a deductive safety analysis: starting from system hazards (*top events*), analysis is performed in a top-down fashion in order to find the causes of system failure in the form of component failure modes – the *basic events*. At the end of the process, *basic* and *top events* are connected through a series of logical gates (*OR, AND*…) in a tree shaped graphical representation. Qualitative and quantitative analysis can be performed on the constructed fault tree. In qualitative analysis, logical operations and simplifications are applied in order to obtain the *Minimal Cut Sets (MCSs)*. MCSs are the smallest conjunctions of basic events that can cause a top event. This gives a better perception of the causes of a system failure than the results of classical FMEA application, where the interactions between components' failures are neglected. MCSs can be characterised by the number of basic events they include – the *MCS order.* If a MCS is composed of two component failure modes, it is said to be a MCS of order 2, for example. As for quantitative analysis, the probability of a top event happening is calculated on the basis of the failure probabilities of the basic events.

Qualitative FTA is particularly useful for supporting (A)SIL allocation and is the general basis for related work in this area (see Papadopoulos *et al*. (2010), Bieber *et al*. (2011), Mader *et al*. (2012) and Dhouibi *et al*. (2014)). The resulting products of its application – the MCSs – can be used to formulate the constraints that must be met when allocating SILs across the architectural elements of a system design. Consider an illustrative System 1 (S1), which provides a safety-critical function. The system integrates four components and the safety-critical function is delivered at the output of a component C4. One SG (SG1) is formulated in dealing with the failure of the safety-critical function and is assigned with ASIL C. The fault tree in Figure 9 has been created to understand the failure behavior of S1, namely what causes the violation of SG1.

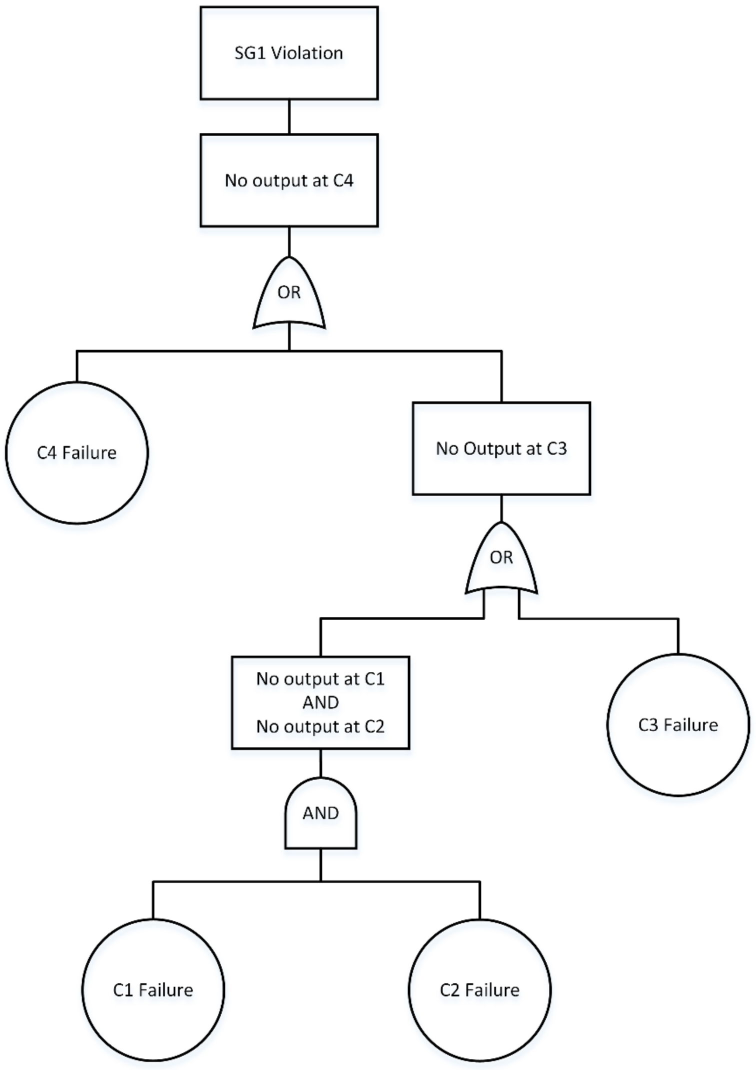


Figure 9 - Illustrative fault tree to investigate causes of SG1 violation in S1.

The fault tree of Figure 9 is a simple one, and the MCSs can be easily determined. Starting with the component failure closest to the top of the fault tree, “C4 failure” is connected to an OR gate which links to SG1 violation. This means that the failure of C4 directly causes SG1, and therefore a first cut set has been found for the fault tree:

MCS1: Violation SG1 = C4 Failure

“C4 failure” is in an OR gate with “No output at C3”; the latter can be caused in two ways. One is “C3 failure”. “C3 failure” is linked with the violation of SG1 through two OR gates, which means that it directly causes this top event. MCS2 can be formulated:

MCS2: Violation SG1 = C3 Failure

Finally, the conjoint failure of C1 and C2 also cause the “No output at C3” event, and in turn the violation of SG1. Note that it is necessary that the two components fail to cause the top event. An MCS3 is composed by the failure of the two components.

MCS3: Violation SG1 = C1 Failure AND C2 Failure

Three constraints can then be easily formulated from the three identified MCSs:

Constraint 1*:* From *MCS1:* $ASIL_{C4} \geq ASIL_{SG1}$ (ASIL C)

Constraint 2*:* From *MCS2:* $ASIL_{C3} \geq ASIL_{SG1}$ (ASIL C)

Constraint 3: From *MCS3*: $ASIL_{C1} + ASIL_{C2} \geq ASIL_{SG1}$ (ASIL C)

Understanding the failure behavior of a system is crucial for the application of ASIL allocation. FTA is an effective, systematic way of studying how a system can fail. It allows a user to determine the causes of system hazards in the form of minimum combinations of components’ failures. This information can in turn be used to identify where decomposition can be applied. The illustrative fault tree presented in this section represents a very simple scenario. In real-world designs, systems can be composed of a vast number of components, resulting in very large fault trees. Moreover, numerous AND gates often appear, and complex failure dependencies need to be analysed. When dealing with this type of system, fault tree synthesis and analysis clearly demands automated support.

2.5.2 Model-based Safety Analysis

Classical application of safety analysis relies on the skills and experience of the safety engineer. Although this approach may work for small, simple systems, as the architectures under analysis become larger and more complex, the probability of human introduced error increases dramatically. Furthermore, such manual application means that an important amount of time needs to be expended. This contributes to safety analysis being performed only a few times during system design, which is unfortunate as they are especially useful as part of an iterative design method where there is a continuing process for identifying flaws and highlighting possible chances for improvement.

Over the years, automated support has risen in the form of Model-based Safety Analysis techniques. They usually fall into two main categories. One centres on compositional safety analysis approaches, where formal or semi-formal languages are developed to specify system failure behaviour and allow their analysis. Compositional Safety Analysis approaches include techniques such as the pioneering Failure Propagation and Transformation Notation (FPTN) (Fenelon & McDermid, 1993), or Component Fault Trees (CFT) (Kaiser *et al.*, 2003; Grunske & Kaiser, 2005) and Hierarchically Performed Hazard Origin & Propagation Studies (HiP-HOPS)

(Papadopoulos & McDermid, 1999). The second category is based on more rigorous modelling to allow model-checking and simulation techniques to identify the effects of component failures on system safety goals – for example, Altarica (Bieber *et al.*, 2004) and FSAP-NuSMV (Bozzano & Villafiorita, 2003).

Compositional safety analyses have the main drawback of not being a fully automated methodology, as the failure data needs to be inputted into the model manually. However, they usually work in a deductive manner to quickly generate Fault Trees. For Simulation-based techniques, on the other hand, component information can be extracted from domain libraries with failure data. Fault trees can be built and analysed using these techniques, but their usual inductive nature makes the process more vulnerable to combinatorial explosion; as a result, it can be difficult to obtain cut sets of high orders (Papadopoulos *et al.*, 2011).

This thesis seeks to investigate an approach that allows the support of ASIL allocation in complex engineering systems. HiP-HOPS was the model-based safety analysis tool chosen to produce the MCSs required to formulate ASIL allocation constraints. It is a deductive technique, with fast and well tested fault tree synthesis and analysis algorithms. HiP-HOPS has further advantages from a practical point of view. It uses the architectural model to capture failure information about components, instead of requiring a separate failure model as the basis for its safety analysis. Also, it does not rely on any external tools for analysis; fault tree synthesis and analysis is completely performed within the tool.

HiP-HOPS contributions to the field of dependability go beyond automated FTA and FMEA:

- analysis of time dependent failure behaviour through PANDORA
  (Walker & Papadopoulos, 2008)
- consideration of preventive maintenance for dependability evaluation
  (Nggada *et al.*, 2010)
- multi-objective optimisation of safety-critical architectures
  (Adachi *et al.*, 2011)
- representation and reuse of component failure patterns
  (Wolforth *et al.*, 2010)

In addition to the above, HiP-HOPS already includes an algorithm for ASIL allocation. The algorithm has been applied successfully to a small case study, but subsequent tests have shown scalability issues. More details about the algorithm will be provided in section 2.5.4.

2.5.3 Fault Tree Analysis with HiP-HOPS

This section provides some insight into the HiP-HOPS process for fault tree analysis. HiP-HOPS line of action is divided into three main phases:

1. Modelling phase
2. Synthesis phase
3. Analysis phase

Modelling phase

The modelling phase is the only non-automated stage of the three. At this point, i) an architectural model must be provided and ii) components must be annotated with expressions that explain their failure behaviour.

Currently, HiP-HOPS interfaces with popular modelling tools, such as Matlab Simulink (Mathworks, Inc, 2015) and Simulation X (ITI Gmbh, 2015). It must be pointed out that these modelling frameworks allow for other analyses to take place upon the same model (e.g. nominal performance analysis). The architectures can be designed hierarchically to deal with complexity. This is also important in meeting current safety standards philosophy of top-down design and analysis, from system level functional architecture down to detailed software and hardware.

In HiP-HOPS, the failure behaviour of a component is to be expressed in the form of logical expressions that relate how a failure at a given output can be originated from failures on its inputs and/or internal failure modes. Input/Output (I/O) failures can be of various classes, and are user defined. The most common are:

- *omission* - the absence of an I/O when there should be one
- *commission* - an I/O exists when it should not
- *value* - an I/O exists with a value different that the one expected (e.g. higher or lower)

Consider the example of a component with one input and one output. The omission of the component's output can be because of an omission of its input, or an internal failure mode. This behaviour can be annotated in HiP-HOPS with the following expression:

Omission-Output1 = Omission-In1 OR FailureMode1

At this point, system level hazards are to be identified as well. They can be defined as logical combinations of component output failures. An example is provided now with an illustrative Hazard 1 (H1) associated with an automatic car lighting system. H1 is defined as: "Loss of front lighting". It can be linked with the light omission of the vehicle's front lamps as follows:

H1 = Omission-LampLeft.light AND Omission-LampRight.light

Synthesis phase

The components' logical failure annotations are, in their essence, small fault trees that explain the failure behaviour of a component. At the synthesis stage, HiP-HOPS links input failures of components to matching classes of output failures from other components. This is enabled via the connections in the architectural model. The tool travels backwards from system hazards to errors in the system inputs. The result is a fault tree per system hazard, with each fault tree being composed of multiple component fault trees glued together (see Figure 10).

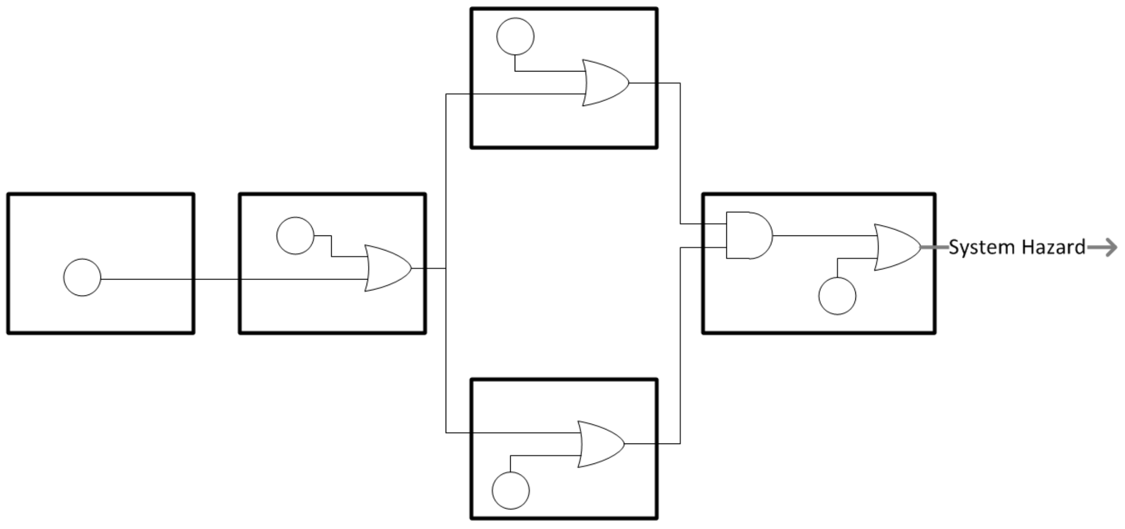


Figure 10 - HiP-HOPS - Synthesizing system hazard fault trees from components fault trees.

HiP-HOPS can output the fault trees in the input format of the popular fault tree analysis software Fault Tree+ (Isograph Ltd, 2015) to improve their visualization.

Analysis phase

In the last stage of its safety analysis ecosystem, HiP-HOPS simplifies the failure logic captured in the synthesised fault trees. Intermediate paths are eliminated and the results consist solely of the relationships between top and basic events. HiP-HOPS works bottom up through the fault tree, applying a modified MICSUP (Minimal Cut Sets Upwards) algorithm (Pande *et al.*, 1975) to obtain the Minimal Cut Sets (MCSs). The MCSs can then be quantitatively analysed to give the probability of top events.

2.5.4 ASIL allocation with HiP-HOPS

The HiP-HOPS algorithm for ASIL allocation provides a framework to find all *non-dominated solutions*. A non-dominated solution includes at least one element with a lower ASIL than in any other solution. Consider an example of a problem where an ASIL D (4) is to be decomposed amongst two redundant elements. A possible non-dominated solution, Solution 1, corresponds to allocating both elements with ASIL B (2). Solution 1 dominates, for example, a Solution 2 where one element is allocated with ASIL B (2) and the other with ASIL C (3). Solution 1 includes one element with a lower ASIL than in Solution 2, but no element with a higher integrity level. In fact, no valid solution exists where one element is allocated with a lower ASIL than in Solution 1, without having the remaining element with a higher ASIL.

HiP-HOPS allocation algorithm is of an exhaustive nature, but it does not enumerate all solutions from the search space. It goes through each cut set and calculates the ASIL algebra compliant solutions. During the process, if an ASIL is assigned to a component, the solutions considered for the following cut sets that include that same element take the previous allocation into account. This limits the search space to be covered as it is illustrated in the following simple example:

- Cut Set 3 (CS3) is constituted by elements Y and Z that jointly violate a safety goal of ASIL D (i.e., $ASIL_Y + ASIL_Z \geq 4$)
- Assume that in cut sets CS1 and CS2, analysed previously, HiP-HOPS assigned Y and Z with ASIL B and A, respectively
- In this way, from the initial 15 feasible solutions that could be considered for CS3, only 11 are left to analyse (those marked “Yes" in the rightmost column of Table 8)

Table 8 - ASIL allocation example for Cut Set 3 - CS3.

	Y	**Z**	**Dominance**	**Total ASIL**	**Compliant w/ previous allocation**
#1	0	4	Non-dominated	4	No
#2	1	3	Non-dominated	4	No
#3	1	4	Dominated	5	No
#4	2	2	Non-dominated	4	Yes
#5	2	3	Dominated	5	Yes
#6	2	4	Dominated	6	Yes
#7	3	1	Non-dominated	4	Yes
#8	3	2	Dominated	5	Yes
#9	3	3	Dominated	6	Yes
#10	3	4	Dominated	7	Yes
#11	4	0	Non-dominated	4	No
#12	4	1	Dominated	5	Yes
#13	4	2	Dominated	6	Yes
#14	4	3	Dominated	7	Yes
#15	4	4	Dominated	8	Yes

HiP-HOPS does not record every valid solution, but only the non-dominated ones. This means that from the 11 feasible solutions marked with “Yes” in Table 8, only allocations #4 and #7 would be presented to the system designer.

Finally, one interesting feature of the tool is that it does not assign ASILs to components, but rather directly to their failure modes. This is to allow a better refinement of requirements when a component can fail in multiple ways. Sub-components that can only contribute to the failure that causes the most severe outcome at the system outputs can be assigned with the stricter requirements, whereas the remaining sub-components can receive lower ASILs. Nevertheless, if the designer requires, HiP-HOPS can revert to a component allocation mode, where the ASIL constraints associated with the different component failures must all be respected by the component allocation.

As mentioned before, HiP-HOPS allocation algorithm suffers from scalability issues. It was applied to a large case study and after two days of processing time (and of generating more than 80 million dominated solutions) it had not finished the problem.

2.5.5 Other support

Two other techniques for automated SIL allocation have been identified: one was developed for the automotive domain and introduced by Mader *et al*. (2012), as a plugin to Papyrus (The Eclipse Foundation, 2015) and the other is called DALculator (Bieber *et al.*, 2011) and targets the aerospace industry.

As with HiP-HOPS, the Mader *et al*. technique is based on annotating a system model with component failure expressions and from them building fault trees that when analysed yield the relationships between component-level failures and SGs. This technique builds an integer linear programming problem where the total number of ASILs is minimised subject to two sets of constraints: ASIL allocation rules and the designer's preferences - he can decide beforehand a specific ASIL for a component based on previous experience with an architectural element. The optimisation task is performed with a constraint solver embedded in the plug-in; Mader *et al*. claim to present the designer with only one solution, the optimal allocation. Scalability is not addressed.

In regards to the DALculator, the approach is described from the point where minimal cut sets are already available. The tool uses pseudo-Boolean logic to formulate the DAL downgrading optimisation problem. The goal is to minimise the sum of DALs across the system while satisfying the reduction rules. Tests were carried out with two solvers, Sat4j (Artois Univeristy & CNRS, 2015) and WBO (Bieber *et al.*, 2011), and results were very satisfactory, even for real-world systems.

Over the course of this thesis project, another work towards automation of SIL allocation has been proposed by Dhouibi *et al*. (2014) for the automotive industry. The automation process proposed is based on the results of FTA, however, no automated FTA engine is mentioned. Furthermore, similarly to the work of Mader *et al*., the safety engineer can input his preferences. The authors argue that to their knowledge there is no efficient cost model to describe the impact of ASILs on development costs, and therefore the system designer should be provided with all non-dominated solutions. They propose an allocation method based on interpreting the problem as a system of linear equations, and deem their approach feasible for small and medium sized architectures.

2.5.6 Discussion and optimisation approach definition

For an architectural element to comply with a higher SIL, development costs necessarily grow. By searching for the solution that reduces the sum of the SILs allocated across the system, the techniques from Mader *et al*. (2012) and Bieber *et al*. (2011) implicitly act to minimise SIL-dependent expenses. Conceptually, they are using a cost function which grows linearly with the SIL increase (e.g. ASIL QM = cost 0; ASIL A = cost 10; ASIL B = cost 20; ASIL C = cost 30; ASIL D = cost 40). This represents a fairly simplistic cost model and instead, multiple heuristics can be formulated to evaluate SIL cost.

The approach from Dhouibi *et al*. (2014) does not seek to find solutions which evaluate as optimal against some SIL cost function, and instead returns all non-dominated solutions. However, a vast space of non-dominated allocations can be available. While currently a definitive ASIL cost heuristic is not available, valid assumptions can still be made that improve the support for the decision making process from the system designer.

The work from this thesis targets the development of an optimisation method which minimises ASIL-imposed development costs and that scales up to complex engineering systems. The optimisation problem is necessarily different from the one tackled by Bieber *et al*. (2011), since ASIL decomposition rules are distinct (more permissive) than the ones from DAL downgrading. Furthermore, dissimilarities are stretched because a user-defined ASIL cost function is allowed in this work. This last difference is also valid in comparison to the work of Mader *et al*. (2012).

As already mentioned, HiP-HOPS will be the tool used to generate the MCSs that allow formulating constraints for ASIL allocation. The HiP-HOPS FTA engine can simultaneously study multiple fault trees from the same system model. This feature enables capturing the causes of violation of the multiple SGs defined for a system. If an element contributes to the violation of more than one SG, it will be possible to formulate constraints which ensure that the element receives an allocation that meets the ASILs of each SG.

Finally, the strategy used by the HiP-HOPS SIL allocation algorithm, where integrity requirements are allocated to failure modes instead of components, is to be adopted in the optimisation approach developed here. This allows for a more recursive methodology to be applied during the hierarchical definition of the system: a subsystem which is refined into complex networks of components can be seen as a "system" that has multiple SGs and SIL requirements.

2.6 Summary

Functional safety standards address the development of safety-critical systems based on E/E technologies. Over the years different standards have been developed to target specific industry sectors (e.g. ARP4754-A and ISO 26262, for the aerospace and automotive domain). They prescribe a top-down approach to allocate integrity requirements across a system architecture, which is based on the concept of SILs. Once allocated to an architectural element, a SIL encapsulates a series of development and validation activities that need to be undertaken. When architectural redundancy is present, modern standards allow for the burden of achieving an original SIL to be divided between the redundant elements, a technique commonly known as SIL decomposition. In the automotive industry, SILs are called Automotive SILs (or ASILs) and guidelines define that the sum of the decomposed ASILs must meet the original level before decomposition.

Whilst the application of ASIL allocation can be effectively achieved in the context of small, simple systems, as system complexity increases their application is difficult.

First, it requires the clear understanding of the complex system architecture failure behaviour to identify which are the valid allocations. Here, safety analysis methods such as FMEA and particularly FTA can help, especially when automation is available. HiP-HOPS and FSAP-NuSMV are two techniques allowing for model-based safety analysis. A second difficulty is related to exploring the large number of allocation possibilities available and finding a valid, cost-efficient allocation. While some support has been proposed in the literature, current approaches either do not directly consider SIL-imposed development costs in determining the allocations to present to the system designer, or consider a single, linear cost model.

This thesis investigates an optimisation approach capable of finding cost-effective ASIL allocations in large, complex automotive architectures, which works across user-defined cost functions. The approach will explore HiP-HOPS's ability to analyse multiple fault trees simultaneously to find efficient allocations that comply with the ASIL requirements defined for all the SGs of a system. Furthermore, it makes use of an allocation method where different ASILs can be assigned to the different failures of a subsystem, as to allow for a better refinement of requirements during the subsystem's hierarchical design.

3. Discovering cost efficient ASIL allocation strategies

When performing ASIL allocation, solutions need to be found which a) comply with the ASIL algebra requirements and b) minimise development costs. Contemporary systems have the potential to generate very large solutions spaces. Efficient optimisation techniques need to be investigated which can provide solutions meeting the two criteria above within time spans allowing for the iterative nature of modern safety systems design.

This chapter addresses objective 1 of the thesis. Initially, the problem of finding ASIL allocations that minimise development costs is mathematically formulated and this is followed by a discussion on possible optimisation research areas. The chapter then converges to presenting the optimisation techniques developed to conduct ASIL allocation.

3.1 The ASIL allocation optimisation problem

Optimisation is generally defined as the process of finding a solution that maximises or minimises a given function and that can be subject to a set of constraints. In ASIL allocation optimisation, the goal is to minimise the sum of the costs associated with the ASILs allocated to the *n* Failure Modes (FMs) of the system. Cost is represented by a function *C*. See Equation 2.

$$\text{objective: } \min \sum_{i=1}^{n} C(ASIL_i)$$

$$ASIL_i \in [0, 1, \dots 4], \qquad n \in \mathbb{N}, \qquad C: \mathbb{Z} \to \mathbb{R}_0^+$$

Equation 2

The solution that minimises ASIL-imposed costs must respect a set of ASIL Allocation Constraints (AACs). AACs define that the sum of the ASILs allocated to the FMs of a Minimal Cut Set (MCS) must be at least the ASIL of the safety goal the MCS violates. Let: *l* be the number of MCSs of a system; m_k the size of the k_{th} MCS; and k_{ASIL} the minimum ASIL requirement for the k_{th} MCS. AACs are formalised in Equation 3.

$$\text{AACs:}\ \left(\sum_{j=1}^{m_k} ASIL_j\right) \geq k_{ASIL}, \qquad 1 \leq k \leq l$$

Equation 3

$$ASIL_j, k_{ASIL} \in [0, 1, \ldots 4], \qquad m_k, l \in \mathbb{N}$$

ASIL allocation is a combinatorial problem. These are characterised by having only discrete decision variables and a finite search space. There is no restriction in regards to the form of the objective function and constraints (for example they can be nonlinear, nonanalytic, black-box) (Papadimitriou & Steiglitz, 1982; Talbi, 2009).

The variables in ASIL allocation are clearly discrete: one of five ASILs is to be allocated to each FM in a system. The solution space is finite and its size can be easily calculated. It reflects the number of combinations of the five ASILs across the *n* failure modes of a system (see Equation 4).

$$\text{Search space size} = 5^n$$

Equation 4

AACs are linear inequalities: the sum of the ASILs allocated to a set of FMs needs to be greater or equal than the ASIL requirement of an AAC. In regards to the objective function, the system designer can input any ASIL-dependent cost heuristic and it does not have to follow any analytical expression. It is only assumed that such function is strictly increasing, i.e., there is always a positive cost jump in implementing an element from a lower ASIL to a higher one.

It is worth noting that depending on the ASIL allocation problem instance, multiple optimal solutions may be available. The optimisation goal pursued here is to find one of them.

3.2 Discussion on optimisation research areas

ASIL allocation optimisation is a complex combinatorial problem with the potential of generating large solution spaces with multiple local optima. When tackling problems with such characteristics the need to investigate *global optimisers* arises. Many global optimisers exist and they can be divided in two major groups: exact and metaheuristics methods. Exact methods include algorithms such as Branch & Bound, the Cutting Plane method, Lagrangian relaxation-based methods and dynamic programming (Wolsey & Nemhauser, 1988; Puchinger & Raidl, 2005). In turn, examples of metaheuristics are Simulated Annealing (Kirkpatrick *et al.*, 1983), the Particle Swarm Optimiser (Kennedy & Eberhart, 1995) and Genetic Algorithms (Holland, 1975). Metaheuristic tactics to explore the search space of a problem are often inspired by natural phenomena, such as the movements of birds in their search for food.

The advantage of exact methods is that when they can complete a problem, they are able to provide its global optimal solution. Exact methods can solve relatively small size problems efficiently; however, in dealing with large scale instances the associated computational burden often becomes excessive (Woon, 2009; Affenzeller *et al.*, 2009; Lin *et al.*, 2012). Furthermore, exact methods often rely on rigorous mathematical characteristics of a problem (e.g. properties of the objective function and/or constraints);

finding a model for a problem that can be solved by these techniques can require great effort. Metaheuristics, on the other hand, cannot guarantee finding optimal solutions, but are means to find good results in high dimensional problems within reasonable time spans. Furthermore, they do not heavily depend on mathematical properties of the problem and are more robust when tackling problems with different characteristics.

Given that this research targets the development of a scalable technique, and taking into account that adaptability across different, user-defined cost functions is also a requirement, metaheuristics are chosen for further investigation. Although the finding of the global optimal solution is not guaranteed, this research will try to discover methods that can provide good results within satisfactory time spans, thus contributing to a more effective design of dependable systems. The flexibility of metaheuristics can also be advantageous not just in dealing with different cost functions, but also, in handling extra constraints that may arise from user preferences, for example.

3.3 Metaheuristics – An overview

The term *metaheuristic* was first introduced by Glover in 1986 (Glover, 1986). The word has Greek origins: *heuristic* comes from the verb *heuriskein (euriskein)* which means “to find”, and *meta* translates to “beyond, in an upper level”. Metaheuristics are high level strategies meant to guide a search process that can be applied to a wide range of optimisation problems. When using metaheuristics, solution manipulation operators need to be tailored to the problem at hand; however, the optimisation strategy itself is problem-independent (Affenzeller *et al.*, 2009).

Metaheuristics are usually non-deterministic and often include a means to avoid getting trapped in local optima. By employing sophisticated methods to explore only a small fraction of the entire search space, they give a reasonable trade-off between solution quality and processing time (Blum & Roli, 2003). There is a common agreement that one of the most defining aspects for the successful application of a metaheuristic is the balance between *exploration* (or *diversification*) and *exploitation* (or *intensification*) (Blum & Roli, 2003; Gendreau & Potvin, 2005a; Yang, 2010; Boussaïd *et al.*, 2013). Exploration relates to the identification of areas with good solutions, whereas exploitation refers to the process of intensifying the search in the promising areas. It is important to quickly identify areas of good solutions, and not waste time in search areas that have no good solutions or that have been searched previously (Blum & Roli, 2003).

Metaheuristics have been applied successfully to numerous combinatorial problems, namely in routing, scheduling and production planning. It is extremely difficult to keep track of all the work with metaheuristics in combinatorial optimisation; the last attempt found in the literature for an exhaustive account dates to the mid-1990s, by Osman and Laporte (1996).

A wide range of different metaheuristics is available. There are multiple ways of classifying them, such as distinguishing if they use search history to guide the optimisation process (e.g. Tabu Search (Glover, 1989, 1990)), or for instance, if the algorithms are nature inspired (e.g. Particle Swarm Optimiser and Ant Colony Optimisation (Dorigo, 1992)). Classifying metaheuristics into single-point and population-based methods is, however, one of the most fundamental distinctions in the literature (Boussaïd *et al.*, 2013). Single-point techniques, also known as trajectory methods, include algorithms such as Tabu Search and Simulated Annealing: they start from a single solution that through neighbouring changes describes a trajectory in the search space. Population-based methods, on the other hand, deal with a group of solutions in every iteration. In the process, characteristics of the population as a whole, are often used to guide their search. Genetic Algorithms, the Ant Colony Optimiser and Harmony Search (Geem *et al.*, 2001), for example, are all population-based metaheuristics.

To the knowledge of the author there has not been any previous work with metaheuristics in optimising ASIL allocation. The reason for the availability of so many metaheuristics in the literature is that they present varying performances depending on the problem they are applied to. In fact, as the No Free Lunch Theorem by Wolpert and Macready (1997) postulates, no technique is versatile enough to overcome all others for every problem. This thesis will investigate suitable optimisation approaches for ASIL allocation based on three of the most popular metaheuristics: Genetic Algorithms, Particle Swarm and Tabu Search. They represent disparate areas of metaheuristics search algorithms. For example, GA and PSO are population-based, whereas TS is a trajectory-based metaheuristic. They are also different in terms of the doctrine underpinning their search process. For instance, PSO stems from an artificial intelligence disciple called *Swarm Intelligence*, which is inspired by the social behaviours of insects and animals (for example ant colonies and schools of fish) (Blum & Li, 2008). The differences between the approaches will be discussed in more detail across the next sections.

3.4 Metaheuristics for ASIL allocation optimisation

GA, PSO and TS have been selected to tackle the ASIL allocation problem due to their popularity and because they represent disparate fields of metaheuristic optimisation algorithms. Where metaheuristics are to be employed, they must be adapted to the specifics of the problem at hand. In reality, their performance in much depends on the suitability of this adaptation. To arrive at ASIL allocation versions of the selected metaheuristics, algorithm designs created for other reliability engineering problems were important sources of inspiration - similarly to ASIL allocation, most of them are discrete and constrained (Levitin *et al.*, 2007). The investigation culminated in three particularly promising algorithms and this section presents them in detail.

In their search procedures, the techniques created share a common representation for ASIL allocation solutions. This representation will be presented before entering the description of each optimisation algorithm.

3.4.1 Solution representation

For the implementation of the algorithms presented in this thesis, a fixed-length, integer-number encoding is used which stores the ASIL value for each Failure Mode (FM) in the system. The number in each slot of the solution vector varies from 0 to 4 and relates to the numbers assigned to ASILs by the ASIL algebra (ASIL QM = 0, ASIL A = 1, ASIL B = 2, ASIL C = 3 and ASIL D = 4). An example is provided in Figure 11.

FM1	FM2	FM3	FM4	FM5
1	3	4	2	0
ASIL A	ASIL C	ASIL D	ASIL B	ASIL QM

Figure 11 - ASIL allocation solution representation (FM = Failure Mode).

The total cost of a solution visited by an algorithm is given by the sum of the costs associated with the ASILs allocated across the different FMs. For the illustrative solution in Figure 11, and assuming a logarithmic cost heuristic (ASIL QM – cost 0; ASIL A – cost 10; ASIL B – cost 100; ASIL C – cost 1000, ASIL D – cost 10000), the solution cost amounts to 11110 units (10 + 1000 + 10000 + 100 + 0).

3.4.2 Genetic Algorithm

Genetic Algorithms, or GAs, are metaheuristics that mimic processes of natural evolution, such as *selection*, *crossover* and *mutation*. GAs are one of the oldest and most well established metaheuristics, and are particularly effective for solving large, complex combinatorial problems (Coit & Smith, 1996b; Kuo, 2001). A typical GA approach starts with a random *population* of solutions, known as *candidates* or *individuals*, who are likely to be infeasible or present a poor *fitness*. In ASIL allocation, a candidate with a poor fitness is a solution imposing high development costs. The idea then is to evolve the initial population, through the use of a set of *genetic operators* and for a given number of *generations*, into feasible, high fitness candidates. The next pages detail the GA approach designed to tackle ASIL allocation.

Selection

Candidates are *selected* from the population to *breed* the next generation. Following the *survival of the fittest* principle, the *selection* mechanism has a bias towards the fitter members of the population. In a first step, candidates are ranked by fitness. Candidates with a better fitness rank higher. Then, a random number, *rand*, is chosen from [1 to $\surd N_I$], where N_I is the number of individuals in the population in each iteration. The candidate which ranks closest to $rand^2$ is selected for breeding. It might not be immediate how this method biases the selection process but the effectiveness of the procedure is illustrated now with an example where $N_I = 100$ and consequently $\sqrt{N_I} = 10$. Choosing a random number between 1 and 10 from a uniform distribution, there is a 70% chance that the selected number is between 1 and 7; therefore there is a 70% probability that the candidate selected ranks between 1 and 49 (1^2 and 7^2, respectively). The 51 lowest ranked candidates (more than half of the entire population) are only left with a selection likelihood of 30%.

The fitness bias of the selection operator can be altered by varying the power of the root (Tate & Smith, 1995).

Crossover

The *crossover* operator relates to biological sexual reproduction and it constructs new candidates (the *children*) using parts of pre-existent solutions (the *parents*). As the selection operator is biased towards the fittest individuals, crossover combines parts of already good solutions in the hope that better ones can be found. Thus, crossover contributes to the algorithm's convergence, focusing the search in areas where good solutions have been found.

In the GA approach presented here, uniform crossover is utilised. Syswerda (1989) has shown that for combinatorial problems, uniform crossover is superior to traditional one and two-point crossover. In uniform crossover, the algorithm iterates through the different allocation variables; for each, it selects the ASIL of one of two parents according to a fixed *uniform crossover probability* and assigns it to the child solution. In the scenario where this parameter is fixed to 40%, a child solution will have approximately

40% of the genes from one parent and 60% from the other. The result of a reproduction example with a *uniform crossover probability* set to 40% is illustrated in Figure 12.

	FM1	FM2	FM3	FM4	FM5
Parent 1	1	3	4	2	0
Parent 2	2	2	2	1	4
Child	2	3	2	1	0

Figure 12 - Crossover operator with *uniform crossover probability* set to 40%.

If one-point crossover were to be used, one FM would be selected as the *crossover point*. The ASILs allocated to that FM and all FMs after it would be copied from one parent to the child. In turn, the ASILs allocated to the FMs that exist before the crossover point would be copied from the second parent. In two-point crossover, two FMs would be selected as crossover points. The ASILs allocated to these FMs and all the ASILs allocated to the FMs between the crossover points would be copied from one parent to the child. The ASILs allocated to the remaining FMs would be copied from the second parent.

Mutation

The *mutation* operator randomly changes ASILs of *children* solutions, introducing diversity in the search process and also contributing to the search not becoming stuck in local optima. In the operator adopted, the algorithm goes through the various allocations of a given children solution and selects the ones to change according to a fixed *mutation rate*. The ASIL of selected FMs are changed to a random number between 0 (ASIL QM) and 4 (ASIL D). Figure 13 shows an example where the *mutation rate* is set to 20% (see next page).

	FM1	FM2	FM3	FM4	FM5
Child	1	3	4	2	0
Mutated Child	4	3	4	2	0

Figure 13 - Mutation example with *mutation rate* set to 20%.

Only part of the children bred, following a fixed *mutation probability*, are selected for mutation.

GA algorithm overview

1. Randomly initialise a population of individuals;
2. Select parents to breed;
3. Generate children via crossover;
4. Mutate children;
5. Add next generation individuals to population;
6. Rank population by fitness;
7. Remove worst individuals until population limit is reached;
8. Return to step 2 if stopping criterion not reached.

3.4.3 Particle Swarm Optimiser

The Particle Swarm Optimiser (PSO) aims to mimic the swarming behaviours from flocks of birds or schools of fish. In PSO, a group of solutions, or a *swarm* of *particles* travel throughout the search space, conceptually looking for food. Each particle learns from its own past experience and other members' knowledge. PSO is often quicker than GA on benchmark functions by at least one order of magnitude (Kennedy *et al.*, 2001; Wang & Li, 2012). The technique was initially presented in 1995 (Kennedy & Erberhart (1995)); in 2008, Poli highlighted "the amazing rate of growth of PSO applications" (Poli, 2008); Poli's report was based on the IEEE Xplore digital library (IEEE, 2015), which showed that between 1995 and 2006 the number of publications on PSO applications grew almost exponentially. PSO continues to be fairly popular since then; over the last 10 years, the number of publications on IEEE Xplore about PSO is greater than for many older metaheuristics, namely, Ant Colony Optimisation, Simulated Annealing and Tabu Search. PSO's popularity is often linked to its simplicity and easy adaption to different problems.

In PSO, each *particle* in a *swarm* has information about its position and velocity; the *position* of a particle represents a solution and the *velocity* vector determines to what position a particle will travel to in the next iteration. The way the velocity vectors are generated depends on the PSO model implemented - there are two main representations of the PSO: the *gbest* and *lbest* models. For both, the velocity vector is influenced by the particle's personal best known position, pbest; however the models differ on their cooperative behaviour - in the gbest model (global best) - the particles make use of the best known position of the entire *swarm*, while in the lbest model (local best) they take into account the best known position from a set of neighbouring particles. It is recognised that the first has a faster convergence speed, but is more susceptible to be attracted to local optima (Kennedy *et al.*, 2001; Poli *et al.*, 2007; Li *et al.*, 2008). The gbest model, also known as the *conventional* PSO, is used in this work; a variety of techniques have been proposed to make this widespread version less vulnerable to becoming stuck in local optima while keeping its fast convergence properties.

It is important to note that PSO was originally developed to deal with problems defined in continuous domains. However, it can be an effective tool in dealing with discrete optimisation problems if properly adapted (Levitin *et al.*, 2007).

The different stages of the PSO algorithm developed are presented now.

Position update

Following the gbest model, in the search for the optimal solution, the particles move according to the velocity and position update expressions depicted in Equation 5 and Equation 6, respectively.

$$v_{zi}^{t+1} = wv_{zi}^{t} + c_1 r_1^t (pbest_{zi}^{t} - x_{zi}^{t}) + c_2 r_2^t (gbest_{i}^{t} - x_{zi}^{t})$$ **Equation 5**

$$x_{zi}^{t+1} = x_{zi}^{t} + v_{zi}^{t+1}$$ **Equation 6**

x_{zi}^{t+1} and v_{zi}^{t+1} are the position and velocity of the *i*th FM to be allocated with an ASIL, in the *z*th particle of the swarm and at the *(t+1)*th algorithm iteration; r_1 and r_2 are random numbers between 0 and 1 from a uniform distribution; *w* is the inertia of the particle which is responsible for determining how the current velocity impacts the one in the next iteration; c_1 and c_2 are weights that determine how much a particle is influenced by its own best position (*pbest*), or the swarm's best (*gbest*), respectively, and are known as *individuality* and *sociability* factors. The velocity expression can be divided into three parts, relating to the influence of current velocity, and personal and global best known positions. Nakano *et al*. (2007) gives a spatial representation on how a particle's position update is influenced by these three factors (see Figure 14).

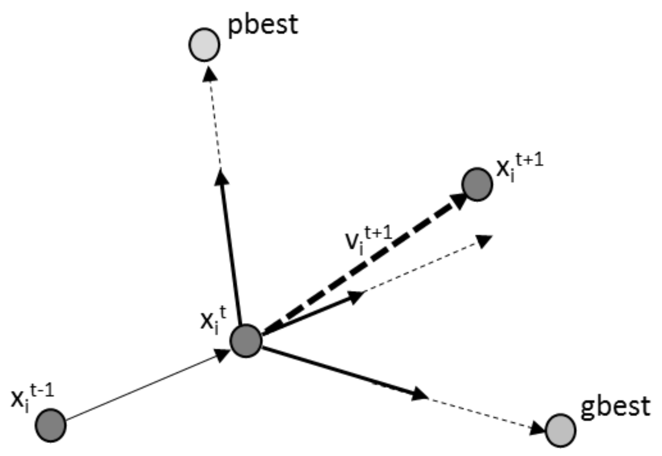


Figure 14 - Spatial representation of PSO's position update.

An example of how an ASIL allocation solution is generated from a particle's position and velocity vectors is provided in Figure 15.

	FM1	FM2	FM3	FM4	FM5
x^t	0	3	1	1	2
v^{t+1}	1	0	1	1	0
x^{t+1}	1	3	2	2	2

Figure 15 - Example of PSO position update.

Discretisation

The ASIL allocation decision variables are of an integer type; PSO cannot be directly applied since the particle's movement equations use real numbers. In the algorithm presented here, the values of the velocity vector are rounded to the closest integer. Velocity is limited to a maximum absolute value of two units to avoid abrupt jumps in the search space.

In order to compensate for errors from discretisation, a Local Search (LS) approach is implemented to explore the area around a particle's position. Wang and Li (2012) have also implemented a LS approach when applying PSO to a redundancy allocation problem.

In order to not increase the computation effort too much, LS is only performed with a fixed *LS period*.

The implemented LS procedure is based on checking the unrounded velocity that dictates the update to the particle's position. For each component of the unrounded velocity vector, the algorithm stores a move that contradicts the rounding performed. For example, if a component of the unrounded velocity is 1.6, its corresponding rounded value is 2; the rounding has resulted in an increment to the particle's position and the algorithm stores a decrement to the corresponding solution component in a list. The elements of that list are then randomly selected. If the change implied by them means an improvement to the current solution, the change is maintained; if no improvement is found, the change is discarded. This process is followed until the entire list has been completely analysed. In situations where the absolute value of the unrounded velocity is greater than two, no change is added to the list. The rounding of this velocity would give two or three units. Changes to the position of a particle are always capped at two units to avoid abrupt jumps in the search space, and therefore the alternative of moving three units is discarded.

The example in Figures 16 and 17 illustrates the LS procedure. A logarithmic cost function is considered (ASIL QM – cost 0; ASIL A – cost 10; ASIL B – cost 100; ASIL C – cost 1000, ASIL D – cost 10000). For the purpose of demonstration, no constraints are formulated and therefore any solution in the two figures is considered to be feasible.

	FM1	FM2	FM3	FM4	FM5	Cost
x^t	0	2	2	1	2	310
v^{t+1} (unrounded)	3.1	-0.6	-2.4	1.8	-2.3	
v^{t+1} (rounded)	2	-1	-2	2	-2	
LS list	-	+1	-	-1	-	

Figure 16 - Example of PSO LS.

FMs 2 and 4 match the criterion for LS, as their unrounded velocities are smaller than 2 units. Moves that contradict the velocity rounding adopted for these FMs are stored in a *LS list*. In this example, the ASIL of FM4 is randomly chosen to be changed first (see Figure 17). The change is kept as it represents an improvement to the cost of the solution. The ASIL of FM2 is modified in a second step; the alteration is discarded as it makes the solution worse.

	FM1	FM2	FM3	FM4	FM5	Cost
x^{t+1} (rounded)	2	1	0	3	0	1010
x^{t+1} (LS – FM4)	2	1	0	2	0	210
x^{t+1} (LS – FM2)	2	2	0	2	0	300
x^{t+1} (final)	2	1	0	2	0	210

Figure 17 - Example of PSO LS (continued).

Mutation

As noted before, the gbest model of PSO is susceptible to getting stuck in local optima. Because of this, a variety of work has considered the integration of the mutation operator from GA into PSO with good results (see for example the works of Li *et al*. (2008) and Stacey *et al*. (2003)). The PSO algorithm shown here also considers mutation. The mutation approach used is the same as the one adopted by the GA algorithm described in section 3.4.2.

Other approaches were experimented with in order to overcome the fast convergence to local optima of PSO's gbest model. The methodologies of Nakano *et al*. (2007, 2010) seek to transport memory mechanisms from Tabu Search (metaheuristic presented in section 3.4.5) into PSO. The approach in Nakano *et al*. (2010), for example, makes a particle forget its best known position for a given number of iterations. In the meantime, it considers other good solutions previously visited in an effort to explore other regions. Early trials with these approaches did not reveal improvements over the mutation operator, and entailed a much harder tuning experience.

PSO algorithm overview

1. Initialise particles with random positions and velocities;
2. Update *pbest* for every particle;
3. Update *gbest*;
4. Update particles' velocities and positions using Equation 5 and Equation 6;
5. Perform LS with every *LS period* iterations;
6. Mutate particles;
7. Return to step 2 if stopping criterion not reached.

3.4.4 GA and PSO - Dealing with infeasibility

GA and PSO "evolution" operators are likely to produce infeasible allocations throughout the search process. In dealing with infeasibility, perhaps the easiest approach is to use the so called *death penalty*, i.e., solutions that do not meet the constraints are simply rejected. The death penalty is appealing for its simplicity, but also for being an efficient approach. As soon as a solution is verified to violate a first constraint, it can immediately be discarded. There are, however, important disadvantages to the use of the death penalty. Firstly, it can lead to search "stagnation" (Coello Coello, 2002): if at a given iteration all solutions are infeasible, the search will not progress. Another drawback is that the approach does not take advantage of infeasibility information to guide the search. Coit and Smith (1996a) have shown that for a redundancy allocation problem, which similarly to ASIL allocation is heavily constrained, GA with death penalty is outperformed by other GA approaches where infeasibility exploration is allowed and infeasibility information is used to direct the search process.

Based on this understanding, the GA and PSO algorithms presented here allow for infeasible solutions to be part of the optimisation procedure. These solutions receive a penalty that adds to their cost, which is based on the concept of degree of infeasibility: if an AAC has an ASIL D requirement, but at a given iteration only ASIL B is achieved, that gives a degree of infeasibility of 2 units (4 (ASIL D) – 2 (ASIL B) = 2); a solution's penalisation depends on the *Total Degree of Infeasibility* (*TDI*) across all the AACs formulated for a system architecture. The magnitude of the penalty to apply is determined by a hybrid approach combining concepts from the works of Coit and Smith (1996a) and of Wang and Li (2012); they have successfully applied Genetic Algorithms and Particle Swarm, respectively, to reliability optimisation. The penalty approach is composed of two stages, referred to as *Pen-I* and *Pen-II* henceforth. Pen-I is a dynamic method that starts by mildly penalising infeasible solutions; the penalty becomes harsher with the algorithm's iteration count. The idea behind the strategy is that by allowing the exploration of infeasible space in the early iterations, efficient routes to (near) optimal solutions can be found (Coit & Smith, 1996a). The dynamic behaviour is realised through the use of a *Near Feasibility Threshold* (*NFT*) that decreases with the iteration count. The *NFT* is updated according to Equation 7.

$$NFT = \frac{NFT_{initial}}{e^{\alpha \times iteration}}$$ **Equation 7**

Parameter α defines how quick *NFT* decreases, which in turn, determines at what rate the optimisation process is pressured into feasible search space; the penalty to apply to an infeasible solution is calculated from the expression in Equation 8.

$$Penalty = \left(\frac{TDI}{NFT}\right)^{\beta} \qquad \textbf{Equation 8}$$

The exponent β is a pre-set severity parameter.

The work of Wang and Li (2012) suggests benefits in allowing for a continuous exploration of the border between feasible and infeasible search space. To take advantage of this, when the search is finally pressed into feasibility by Pen-I, Pen-II is enabled. The feasibility of the best allocation present within the set of solutions in each iteration is continuously examined; from that, the proportion of infeasible solutions over the past δ iterations, $\gamma(t+\delta)$, is calculated (see Equation 9).

$$\gamma(t+\delta) = \frac{inf_{best}(t+\delta)}{\delta} \qquad \textbf{Equation 9}$$

$inf_{best}(t+\delta)$ represents the number of infeasible best solutions in the last δ iterations. $\gamma(t+\delta)$ is used for a new way of updating the *NFT* (see Equation 10). The penalty to apply to a solution is still obtained from Equation 8.

$$NFT(t+\delta) = \begin{cases} NFT(t) \times \frac{1}{\varepsilon} \times \left(1.5 - \gamma(t+\delta)\right), & \gamma(t+\delta) > 0.5 \\ NFT(t), & \gamma(t+\delta) = 0.5 \\ NFT(t) \times \varepsilon \times \left(1.5 - \gamma(t+\delta)\right), & \gamma(t+\delta) < 0.5 \end{cases} \qquad \textbf{Equation 10}$$

ε is to be set to a value equal or greater to 1.

If $\gamma(t+\delta) > 0.5$, there are more best infeasible solutions than feasible and therefore the penalty is increased, through the *NFT*, to direct the search towards feasibility (smaller values for *NFT* give a greater penalisation – see Equation 8); on the other hand, if *γ(t+δ)* < 0.5, it means that in the last *δ* iterations there were more feasible bests and the penalty is decreased to encourage exploration of infeasible search space. Changing *ε* manipulates the magnitude of *NFT* change in directing the search towards feasible or infeasible search space. The *NFT* update, as given by Equation 10, takes into account the last *δ* algorithm iterations and it occurs with a period of *δ* iterations.

3.4.5 Tabu Search

Kulturel-Konak *et al*. (2003) claim that the computational effort posed by population-based methods, where multiple solutions are evaluated over the course of many iterations, can be significant for some complex problems. The Tabu Search (TS) metaheuristic is a simpler algorithm based on a single solution being changed iteratively through neighbouring moves. TS implementations usually combine a local search strategy with a set of memory mechanisms designed to avoid the trap of local optimality. TS has proven to behave competitively in various complex problems, outperforming, for instance GA, both in processing time and quality of solutions within some applications - e.g. timetable scheduling (Chu & Fang, 1999) and redundancy allocation (Kulturel-Konak *et al.*, 2003).

The Steepest Ascent Mildest Descent (SAMD) method (Hansen & Jaumard, 1990) is a member of the TS family; the approach presented in this section is based on the work of Hansen and Li (1996) who applied SAMD to system reliability. ASIL allocation optimisation is a minimisation problem and SAMD was originally conceived for maximisation; therefore the method was appropriately modified to a Steepest Descent Mildest Ascent (SDMA) variant. SDMA follows the steepest descent direction until a local minimum is reached, and then makes use of the mildest ascent route to escape from it. The method is described in detail over the following pages.

Neighbourhood exploration

The algorithm always starts from a feasible solution. Every new allocation visited results from evaluating a set of possible changes to the current solution and making a decision. Changes that produce an infeasible allocation are immediately removed from the decision process.

The optimisation begins with the pursuit of the steepest descent direction. As defined in the beginning of the current chapter, every ASIL cost heuristic is considered to be strictly increasing, meaning that reducing the ASIL of a FM will always result in a decrease to the system's total ASIL-dependent cost. The steepest descent direction is followed by decrementing the ASIL of the FM for which this results in the highest system cost reduction. Consider the illustrative Tabu Search solution, for a given iteration *t*, presented in Figure 18.

	FM1	FM2	FM3	FM4	FM5
Solution^t	2	4	3	2	1

Figure 18 - Illustrative TS solution at iteration *t*.

Assuming a logarithmic ASIL cost function (ASIL QM – cost 0; ASIL A – cost 10; ASIL B – cost 100; ASIL C – cost 1000, ASIL D – cost 10000), decrementing the ASIL of FM2 represents taking the steepest descent direction:

- FM1 and FM4 (ASIL B to ASIL A) – cost reduction: 100 – 10 = 90
- **FM2 (ASIL D to ASIL C) – cost reduction: 10000 – 1000 = 9000**
- FM5 (ASIL A to ASIL QM) – cost reduction: 10 – 0 = 10

The solution that would be obtained for iteration *t+1* is presented in Figure 19.

	FM1	FM2	FM3	FM4	FM5
Solution^{t+1}	2	3	3	2	1

Figure 19 - Illustrative TS solution at iteration *t+1*.

When analysing descent moves, some can present the same cost variations and it is possible that there are multiple which evaluate as the steepest descent. When this is the case, all the steepest descent moves are taken and the algorithm randomly chooses one to be followed in the next iteration.

Consider now that the solution at iteration *t+1* is a (local) minimum, i.e. any further descent moves would result in the violation of one AAC or more. In escaping from the potential local optimum, the algorithm takes the mildest ascent route, that is, it increments the ASIL of the FM representing the lowest cost growth. For the solution of iteration *t+1*, incrementing the ASIL of FM5 is the mildest ascent move.

- FM1 and FM4 (ASIL B to ASIL C) – cost increase: 1000 – 100 = 900
- FM2 and FM3 (ASIL C to ASIL D) – cost increase: 10000 – 1000 = 9000
- **FM5 (ASIL A to ASIL B) – cost increase: 100 – 10 = 90**

The result of taking the mildest ascent move in iteration *t+2* is shown in Figure 20.

	FM1	FM2	FM3	FM4	FM5
Solution^{t+2}	2	3	3	2	2

Figure 20 - Illustrative TS solution at iteration *t+2*.

In scenarios where multiple moves imply the same minimum cost growth, one of them is randomly chosen to be followed in the next iteration.

The memory mechanism

While escaping from local optima through the pursuit of the mildest ascent route, reverse moves are forbidden for *p* iterations. Assuming that *p* is greater than zero, this gives that in iteration *t+3* the ASIL of FM5 cannot be reduced. The number of iterations to forbid decrementing the ASIL of a given FM *i* is stored in a variable f_i. To promote diversity further, the algorithm also forbids incrementing a FM's ASIL during *q* iterations after a descent move. For a FM *i*, ascent move restrictions are stored in f'_i.

The values of *p* and *q* are dynamically changed in order to reduce the algorithm's sensitivity to their selection; they are incremented with periods $updatePeriod_p$ and $updatePeriod_q$, respectively. $limit_p$ and $limit_q$ are the maximum values for *p* and *q*; when *p* and *q* reach theirs limits, they are reset to zero.

The Hansen and Lih (1996) approach has been extended to allow the overruling of a tabu move restriction if this means 1) obtaining a better solution than those found previously or 2) reaching a solution with the current best cost but that has not been visited before. Conditions allowing for tabu constructs to be disregarded are known as *aspiration criteria* (Gendreau & Potvin, 2005b).

Initial solution generation

The Tabu Search technique does not allow for infeasible search space exploration; consequently, it requires a method to generate an initial feasible solution. An approach has been devised where each FM starts by being assigned with a random ASIL from 0 to 4. The resulting solution is, at this point, likely infeasible. Next, a FM that has not been assigned with ASIL 4, and which integrates at least one AAC not being satisfied, is randomly selected and gets its ASIL incremented. This step is repeated until the solution becomes feasible.

Is the optimum solution always within reach?

ASIL allocation is a constrained optimisation problem and, as such, it produces search spaces with both feasible and infeasible solutions. One possible concern regarding the Tabu Search technique is how feasible ASIL allocations get distributed across the solution space, namely if multiple isolated regions of feasible solutions are generated. If that scenario were to be confirmed, TS would not be able to solve a problem to optimality if it is initialised in a feasible region different from the one where the optimal allocation is located. In fact, the neighbourhood exploration strategy integrated in the TS technique, where only a single increment or decrement to one of the allocation variables is permitted, does not allow for jumps in the solution space. Furthermore, the TS technique does not accept infeasible solutions in its search process; in this way, travelling through infeasibility to the region where the optimal solution is located would not be possible.

The scenario where multiple feasible regions are sparsely distributed in the solution space is not, however, an issue for the ASIL allocation problem. In fact, between any two feasible allocations, there is always a feasible path through iterative increments and decrements to the ASILs assigned across a problem's FMs. Consider a simple illustrative ASIL allocation problem for which there are only two variables (FM1 and FM2); a logarithmic cost function is assumed again to evaluate solutions for this problem. Only one AAC is defined:

AAC1: $ASIL_{FM1} + ASIL_{FM2} \geq$ ASIL C (ASIL 3)

For this simple example, a global optimal solution can be easily identified. With the logarithmic cost function (ASIL QM – cost 0; ASIL A – cost 10; ASIL B – cost 100; ASIL C – cost 1000, ASIL D – cost 10000), the cheapest way to decompose an ASIL C through two FMs is ASIL A + ASIL B (cost: 100 + 10 = 110). One can arrive at a solution like this from anywhere within the feasible search space without violating AAC1 in the process. Consider the scenario in Figure 21 where a *local optimal* solution is found in iteration 1; making that solution worse by incrementing one of the variables (but still remaining within feasible space) makes it possible to travel to the *global optimum*.

- **Iteration 1:** AAC1 is strictly fulfilled, i.e., a decrement to the ASIL of FM1 or FM2 would result in an infeasible solution.
- **Iteration 2:** The ASIL of FM1 is incremented, resulting in a cost increase. The requirement of AAC1 is now surpassed by one unit, which means that the ASIL of one of the FMs can be decremented.
- **Iteration 3:** The ASIL of FM2 is decremented, resulting in a cost decrease and a new best cost is identified. AAC1 is strictly fulfilled.

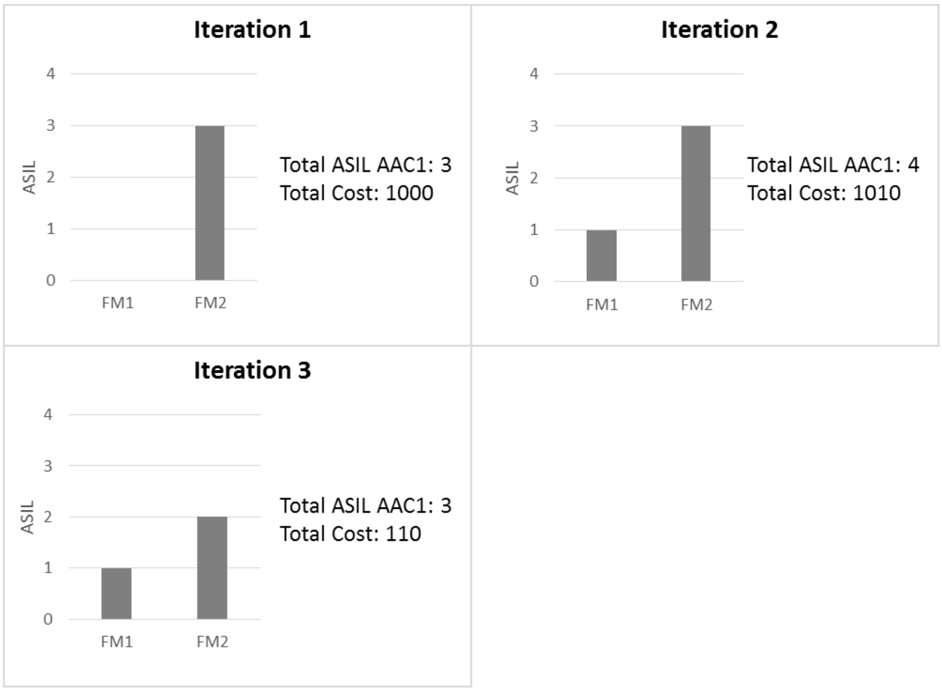


Figure 21 - Travelling from local to global minimum through feasible search space.

To travel between any two feasible solutions, in a worst case scenario, all ASILs could be incremented until reaching ASIL D, and from such solution, any feasible allocation can be reached through successive decrements to the ASILs allocated to the different FMs (See Figure 22).

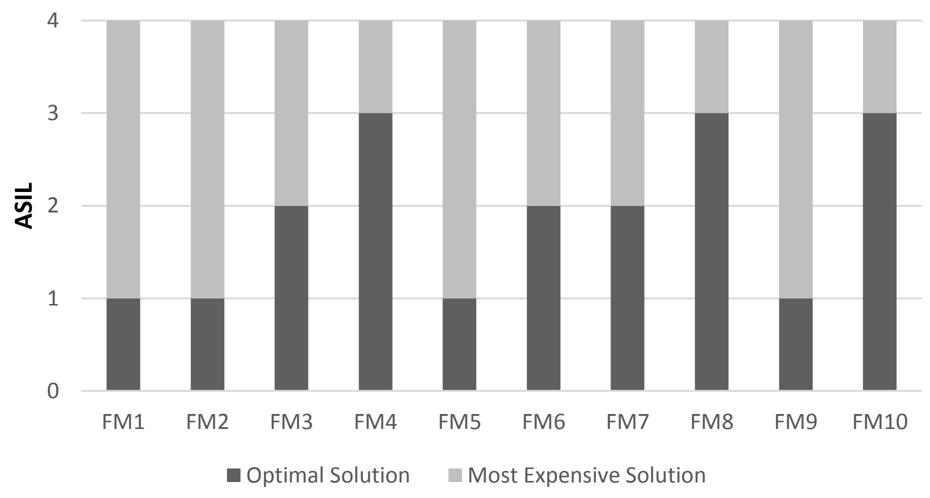
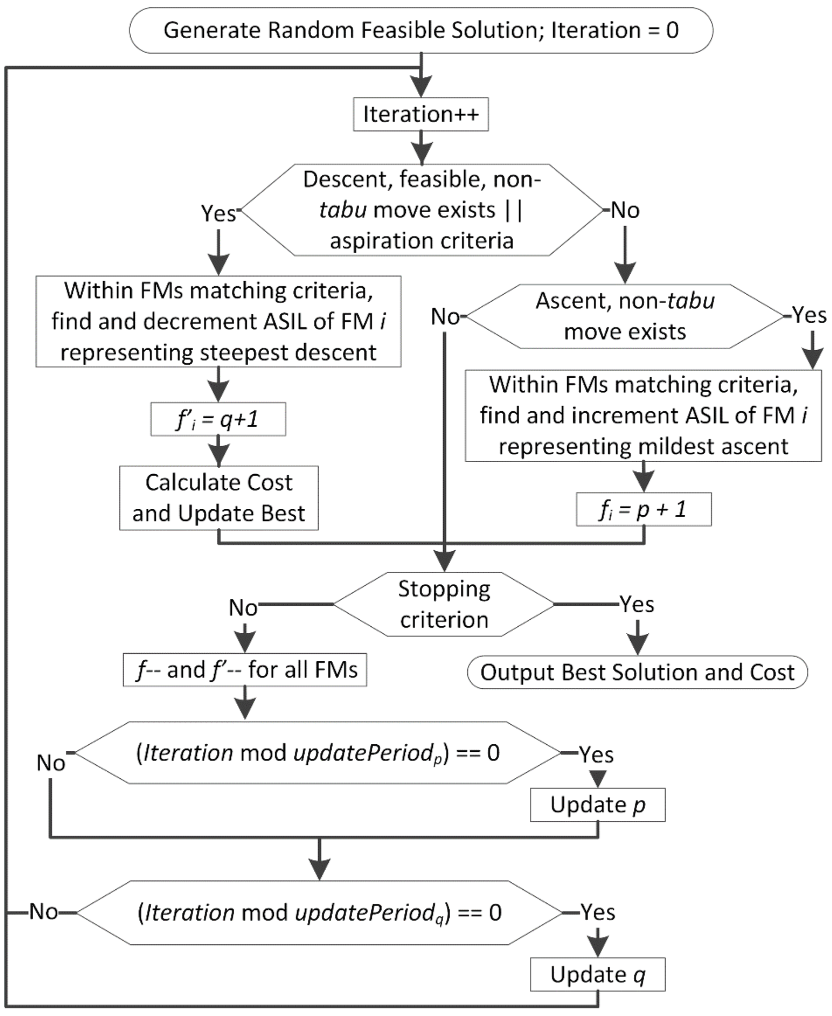


Figure 22 - Travelling from the most expensive solution (light grey) to the optimum (dark grey).

TS's means to explore the solution space considers, precisely, successive increments and decrements to the different allocation variables. Conceptually, the technique can reach the optimum solution for any ASIL allocation instance, independently of where its search process starts in the solution space.

TS overview

For better understanding of the TS approach, the *flowchart* of Figure 23 was created.



Generate Random Feasible Solution; Iteration = 0
Iteration++
Descent, feasible, non-*tabu* move exists || aspiration criteria
Yes
No
Within FMs matching criteria, find and decrement ASIL of FM *i* representing steepest descent
$f'_i = q+1$
Calculate Cost and Update Best
No
Ascent, non-*tabu* move exists
Yes
Within FMs matching criteria, find and increment ASIL of FM *i* representing mildest ascent
$f_i = p + 1$
Stopping criterion
No
Yes
f-- and *f'*-- for all FMs
Output Best Solution and Cost
No
(*Iteration* mod *updatePeriod$_p$*) == 0
Yes
Update *p*
No
(*Iteration* mod *updatePeriod$_q$*) == 0
Yes
Update *q*

Figure 23 - Flowchart describing the Tabu Search algorithm. "mod" is the modulus operator, which gives the remainder of a division between two numbers.

3.4.6 Other (less) successful approaches

Arriving at the three optimisation techniques presented up to this point in the chapter was an evolutionary process. ASIL allocation is a new problem, and multiple approaches were tried to solve it. This section provides a brief overview of two optimisation methods that gave satisfactory results, but that fall short of the ones given by the three techniques already identified in this chapter.

Kulturel-Konak (2003) proposed a Tabu Search technique to solve a redundancy allocation problem where the exploration of infeasible search space is permitted and a dynamic penalty approach is implemented. The use of TS with dynamic penalty to solve ASIL allocation resulted in a large computational effort increase per iteration. When infeasibility exploration is used, evaluating the outcome of a move requires analysing every AAC in which the ASIL to be changed is included; the prospective cost variation depends on the *TDI* obtained after the move is taken. In the feasibility-only version of TS, when a move is under analysis, the cost variation it implies can be easily calculated; it is simply the difference between the cost associated with the current allocated ASIL and the cost of the one resulting from the move being analysed. There is still the need for AACs evaluation; however, as soon as the approach finds a first AAC that is not fulfilled, it is known that the move under analysis leads to an infeasible solution, and therefore it can be immediately discarded.

The increase in processing effort per iteration was not accompanied with an improvement to the quality of the solutions found; in fact, in some cases, the solutions were worse than the ones found by the feasibility-only version of TS. As shown in the previous section, TS does not necessarily need to visit infeasible search space to access any of the solutions of the feasible region. The dynamic penalisation approach helps the population-based algorithms to move throughout the solution space; their search operators can easily generate infeasible solutions and if these are not allowed in the search process, the search can stagnate. In the case of TS, a set of candidate neighbour solutions are evaluated and the best, according to the search criteria at that point, is selected. Since this *a priori* evaluation process exists, infeasible solutions are not visited unless the algorithm implementation decides to do so. For the ASIL allocation problem, ensuring that TS visits feasible search space only seemed to provide for a better solution space exploration strategy. The number of parameters for the TS technique also grows if the dynamic penalisation approach is considered; this necessarily makes the tuning experience harder and could have impacted the performance of the TS version that permits infeasible search space exploration.

PSO-TS hybrid

In recent years, a vast number of optimisation approaches have been proposed that combine concepts from different optimisation techniques. These *hybrid* methods are expected to benefit from complementary characteristics of the different techniques they integrate (Blum *et al.*, 2011). A hybrid approach combining the PSO and TS algorithms presented in the previous sections of this chapter was created. PSO was used for global exploration, and thus responsible for identifying areas of prospective good solutions, and TS acted as a powerful local search method. A similar approach has been considered by Li *et al*. (2007) to solve an energy transmission optimisation problem.

The non-penalty based TS was used; the technique is initialised every time a particle from PSO finds a new feasible *pbest*. The main issue with the PSO-TS hybrid seemed to be related with fact that once the Tabu Search stage is activated for the first time, the powerful local search technique reveals solutions that are substantially better than the ones found to that point; as a result, the entire swarm gets attracted to that particular area. To counter this behaviour, PSO's c_1 and c_2 parameters were manipulated to favour further global exploration, but no substantial improvements were found. Parameterisation is another of the issues with this approach. In PSO-TS, the number of parameters to tune is the sum of the ones from the two original independent algorithms - PSO and TS.

3.5 Summary

The chapter has provided a formulation for the optimisation problem of finding requirements-compliant and cost-effective ASIL allocations. Exact and metaheuristic techniques were considered to solve this combinatorial problem. Exact methods can find optimal solutions for small size problems, but for large scale instances they often require excessive computational effort. Metaheuristics were chosen for investigation; they cannot guarantee finding optimal solutions, but they are known for finding good solutions efficiently in large search spaces and to be robust when tackling problems of different characteristics.

Many metaheuristics have been proposed in the literature over the years. The rationale for the co-existence of this variety of approaches is that their performance varies from application to application and there is not one that overcomes all others in every problem. Three of the most popular metaheuristics were selected - GA, PSO and TS - and from them, problem specific algorithms have been developed.

The three most promising techniques described in this chapter are a result of an in-depth literature examination, as well as of a study of many nuances of the algorithms operators and strategies for feasible and infeasible search space exploration. Experimentation was also conducted with hybrid approaches. The three most promising algorithms will be compared in chapter 5.

4. Case studies and evaluation metrics

ASIL allocation is a new optimisation problem and no benchmark has been defined in the literature to evaluate the performance of the optimisation techniques when tackling it. Part of this chapter is to meet objective 2 defined for the thesis: a group of problems and a set of evaluation metrics are established in order to compare the ASIL allocation optimisation methods developed in chapter 3.

Four different case studies were created. Three are variants of a Hybrid Braking System (HBS) and the fourth is a system of systems comprising braking and steering functionalities (it will be referred to as Brake and Steering System, or BSS, from now onwards). The BSS has been provided by an industrial partner of HiP-HOPS and for confidentiality reasons it cannot be described in the same depth as the HBS. In this way, the HBS will serve to illustrate how ASIL allocation can be supported in a complex scenario by HiP-HOPS, together with an automated allocation algorithm (objective 4). The HBS will also be used to address objective 5 of the thesis: the impacts of using different cost heuristics will be analysed over the optimal ASIL allocations for this case study.

4.1 The hybrid braking system

In chapter 1 attention was brought to the increasing reliance of safety-critical systems on software and hardware. *Brake-by-wire* represents the pinnacle of this evolution for braking technologies. In brake-by-wire architectures, there is no mechanical or hydraulic link between the braking pedal and the actuators. Instead, sensors measure the movements of the pedal and computer-controlled braking actuators are activated. Brake-by-wire systems are attractive because they provide a faster and more accurate braking control; they also integrate more easily with assistance systems (such as antilock systems) and are lighter and less bulky than technologies reliant on hydraulic and mechanical parts (Xiang *et al.*, 2008). Since in “true” brake-by-wire systems there is no mechanical backup in case of failure, safety is a major concern. Consequently, brake-by-wire architectures usually include fault-tolerant mechanisms.

The HBS is a brake-by-wire technology designed for electrical vehicles. Specifically, the HBS was conceived for propulsion architectures that integrate one electric motor in each of the vehicle’s wheels (de Castro *et al.*, 2011). The system is *hybrid* because it combines the actions of two types of devices: the electrical In-Wheel Motors (IWMs) from the propulsion system and frictional Electromechanical Brakes (EMBs). When braking, IWMs work as generators and transform the vehicle’s kinetic energy into electrical energy that is provided to the Powertrain Battery. This is an important feature, since it works to increase the vehicle’s range. IWMs have, however, braking torque limitations at high wheel speed regimes or when the battery has a high state of charge, for example. The EMBs provide the braking torque required to meet the demands when the IWMs do not suffice. Another interesting feature of the system is that it ensures independent braking control for each wheel; this improves braking performance.

The HBS architecture is displayed in Figure 24. As it can be observed, there is no hydraulic link connecting the Braking Pedal and the braking actuators. Instead, a redundant electronic bus system is responsible for communication between an Electronic Pedal and Wheel Node Controllers (WNCs).

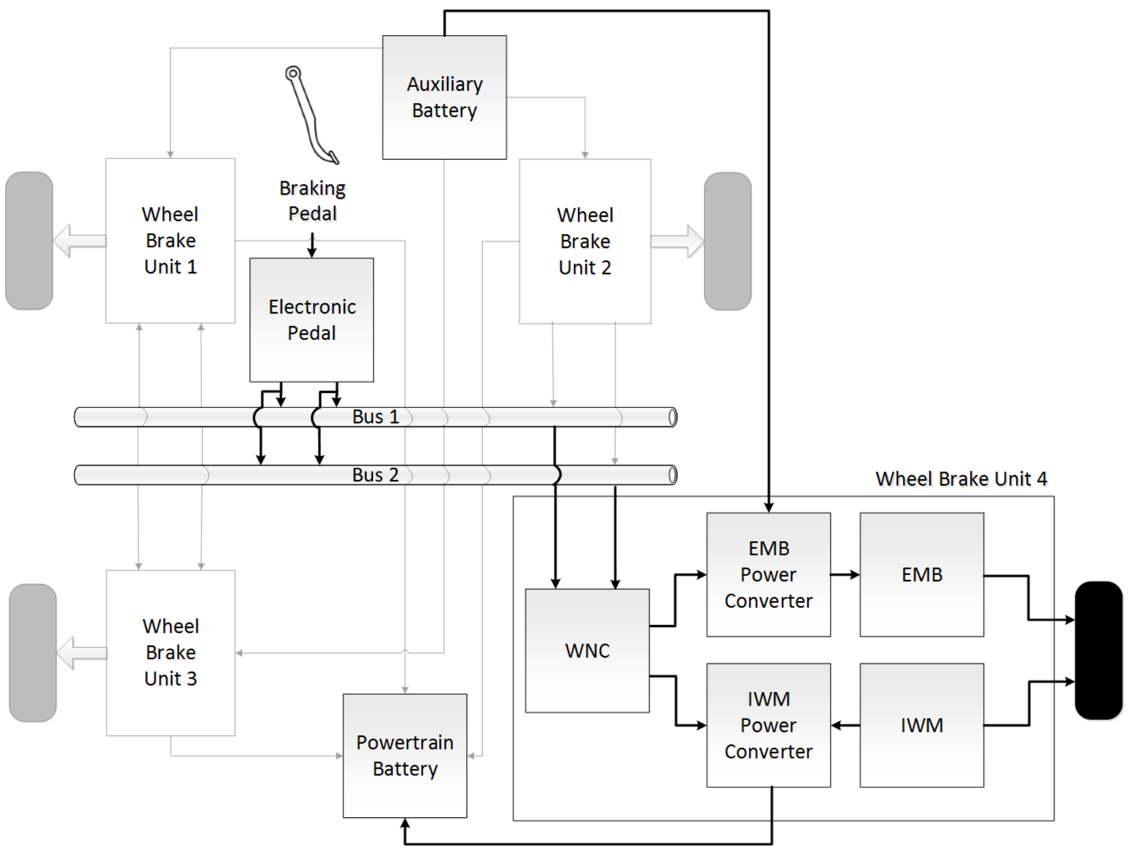


Figure 24 - Hybrid braking system architecture.

The Electronic Pedal measures the brake demands from the driver and determines the braking to be produced in each wheel. The WNCs receive the braking needs for the wheel they are associated with and subsequently control a pair of braking actuators. The system includes two batteries. The Auxiliary Battery is responsible for feeding the vehicle’s low voltage loads, including the EMBs; the Powertrain Battery, on the other hand, is used in the high-voltage energy transmissions with the IWMs.

An overview of the HBS was given. The system will now be used to demonstrate how HiP-HOPS, together with an automated optimisation technique, can provide effective and efficient support for ASIL allocation.

The Fault Tree Analysis (FTA) capabilities of HiP-HOPS determine how system hazards are caused by combinations of component level failures; in turn, this information enables rationalising how system level ASILs can be allocated and decomposed across the components of a system architecture. As described in chapter 2, the first stage in HiP-HOPS approach for FTA is called *modelling*; essentially, a model of the system needs to be provided and its architecture annotated with a series of expressions that describe its failure behaviour. This is the only non-automated step for the entire ASIL allocation approach proposed in this thesis.

Three different variants of the HBS were designed to evaluate the performance of the optimisation techniques. The simplest variant, HBS Model 1 (or HBSM1) is described for illustration. HBSM1 only considers the braking of one wheel (see Figure 25).

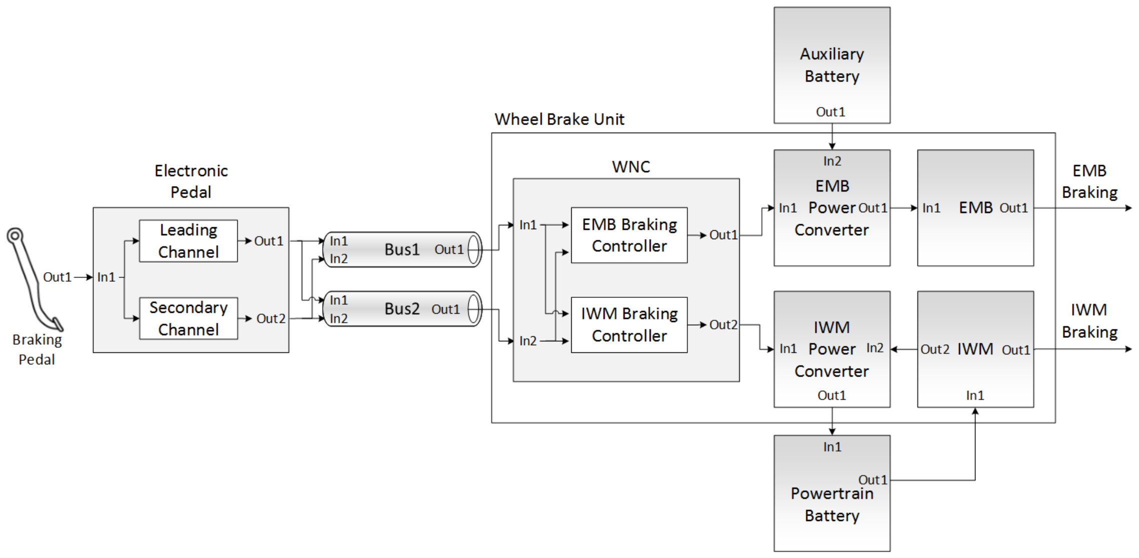


Figure 25 - HBSM1.

Figure 25 shows assumptions about the internal structures to be adopted for the Electronic Pedal and the WNC. These have been taken into account when describing the failure behaviour for the system. In the Electronic Pedal there are two independent channels that sense and process the braking demands from the driver. If the output of the Leading Channel gets omitted, the information provided by the Secondary Channel is used. For the WNC architecture, it is defined that EMB and IWM braking are controlled from independent processing units. If the unit responsible for EMB braking fails, the IWM output can still be controlled correctly. Finally, failures in the IWM Power Converter and in the Powertrain Battery can have effects on the IWM braking output. So that failures in these two components can be propagated in the model to the IWM, a virtual connection has been created between the Powertrain Battery and the IWM.

The functional behaviour for the HBSM1 is as follows: as the driver presses the braking pedal, his actions are sensed and processed in an Electronic Pedal unit; the braking force for the wheel is calculated and sent through the redundant communication bus to the WNC. There, braking torque demands are determined for each actuator; commands are then directed to the power converters that control the IWM and the EMB. When braking action is taking place, energy flows from the auxiliary battery to the EMB; as stated previously, regarding the IWM, energy flows towards the Powertrain Battery while braking.

Two hazards are considered for the HBSM1:

- No braking after command – Hazard 1 (H1)
- Braking with wrong value – Hazard 2 (H2)

ASIL D and A are assigned to H1 and H2, respectively. Conducting ISO 26262's risk assessment procedure to determine the ASIL of each hazard falls outside of the scope of the thesis. For simplicity, ASILs are assigned solely on the basis of the hazards' assumed severities.

So that HiP-HOPS can look for causes of the two hazards identified, the hazards need to be linked to problems at the outputs of components in the system model. It was considered that H1 occurs if both braking actuators omit their outputs; H2, on the other hand, is the result of an incorrect value output of at least one of the actuators. This is represented by the following expressions:

- H1 = Omission-EMB.out1 AND Omission-IWM.out1
- H2 = IncorrectValue-EMB.out1 OR IncorrectValue-IWM.out1

The next step is to annotate each component in the system architecture with failure expressions that explain how the components can produce deviated outputs in response to internal failures or problems at their inputs. Consider the example of Bus 1, which provides a communication link between Electronic Pedal and the WNC. Bus 1 will have its output omitted if there is a problem at both of its inputs, or if it has an internal problem (an Omission Failure, or OFailure). This failure behaviour can be formalised through the expression below:

- Omission-Out1 = OFailure OR Omission-ElectronicPedal.Out1 AND Omission-ElectronicPedal.Out2

The full list of failure annotations for the HBSM1 is included in Appendix I. More details about the failure behaviour of the system can be found in Azevedo (2012).

As discussed in chapter 2, the approach from this thesis follows HiP-HOPS previously introduced method of allocating ASILs to FMs instead of components. In this way, where components can fail in multiple ways, requirements can be better refined. When applying the approach hierarchically, sub-components that only contribute to the component FM which originates the most severe system hazard can be assigned with the stricter requirements, whereas the remaining sub-components can be allocated with less stringent ASILs.

By the end of the annotation process, 24 component FMs were identified for the HBSM1. This gives a total search space size of 5^{24} (5 ASILs and 24 FMs), which amounts to approximately 5.96×10^{16} solutions. This number is clearly out of reach of manual exhaustive evaluation. Conceivably, the system designer would not enumerate all the solutions to choose an allocation. But in the face of this large solution space, finding a feasible allocation that minimises development costs is a laborious, difficult task.

After the modelling stage, HiP-HOPS fault tree *synthesis* and *analysis* phases are automatically carried out. Results reveal 31 minimal cut sets for the HBSM1:

- 19 minimal cut sets for H1
    - 1 single point of failure[1] and 18 dual points of failure[2];
- 11 minimal cut sets for H2
    - 10 single points of failure and 2 dual points of failure.

Appendix I of the thesis shows the cut sets for HBSM1.

HiP-HOPS automatically converts the cut sets information into AACs. Consider the example of the cut set constituted by the internal Omission Failures (OFailures) of the EMB and the IWM, which together originate H1:

- Cut set: EMB.OFailure AND IWM.OFailure = H1
- AAC: $ASIL_{EMB.OFailure} + ASIL_{IWM.OFailure} \geq ASIL_{H1}$

[1] Single points of failure are cut sets composed of one component failure mode.
[2] Dual points of failure are cut sets composed of two component failure modes.

With this information, and with a cost function that expresses how development costs change with the different ASILs, the optimisation problem is completely formulated and can be directly tackled with any of the optimisation techniques presented in chapter 3. One possible cost function to evaluate ASIL allocations is presented in Table 9. It is based on a discussion from the LinkedIn forum on ISO 26262 (Allen, 2012) which indicated that there is a greater cost jump between ASILs B and C, than between any other two ASILs.

Table 9 - Experiential-I cost function.

Cost Function	ASIL QM	ASIL A	ASIL B	ASIL C	ASIL D
Experiential-I	0	10	20	40	50

The HiP-HOPS exhaustive algorithm for ASIL allocation does not scale well, but it is capable of finishing the one-wheel model of the HBS. The technique returns all non-dominated solutions for a problem. An allocation is said to be dominated if it presents one or more Failure Modes (FMs) with higher ASILs than other solutions, but no FM with lower ASILs. The optimal solution for a given ASIL allocation instance is logically non-dominated. For the HBSM1, the HiP-HOPS exhaustive technique found a total of 125 non-dominated solutions; evaluating each one of them with the Experiential-I cost heuristic singled out the optimal solution presented in Table 10 (see next page). In Table 10, and throughout the remainder of the thesis, FMs names are given in the following format: “Name of component”.”X”Failure”Y”. “X” is to be replaced by an “O” for an internal omission failure or a “V” for an internal failure causing a value deviated output; within a same component, and for a same type of internal failure, “Y” is replaced by an FM ID starting with number 1. For the Electronic Pedal component, XFailure1 refers to failure modes of the Leading Channel, whereas XFailure2 to FMs of the Secondary Channel. In the case of the WNC, XFailure1 relates to FMs of the processing unit controlling EMB braking; XFailure2 concerns the FMs of the processing unit responsible for controlling IWM braking.

Table 10 - HBSM1 optimal ASIL allocation for Experiential-I cost function.

	Exp-I Cost: 390
Components FMs	#1
Braking Pedal.OFailure1	4
Braking Pedal.VFailure1	1
Electronic Pedal.OFailure1	2
Electronic Pedal.OFailure2	2
Electronic Pedal.VFailure1	1
Electronic Pedal.VFailure2	0
Bus1.OFailure1	2
Bus2.OFailure1	2
WNC.OFailure1	2
WNC.OFailure2	2
WNC.VFailure1	1
WNC.VFailure2	1
Auxiliary Battery.OFailure1	2
Auxiliary Battery.VFailure1	1
Powertrain Battery.OFailure1	2
Powertrain Battery.VFailure1	1
EMB Power Converter.OFailure1	2
EMB Power Converter.VFailure1	1
IWM Power Converter.OFailure1	2
IWM Power Converter.VFailure1	1
EMB.OFailure1	2
EMB.VFailure1	1
IWM.OFailure1	2
IWM.VFailure1	1

The solution Table 10 is found using any of the three optimisation techniques presented in chapter 3. From the end of the *modelling* stage in HiP-HOPS up until the optimal solution is retrieved by any of the optimisation techniques developed, no more than a few seconds pass.

The solution given in Table 10 is not necessarily final. The decision on which allocation to follow can be impacted by suppliers, for example. Two FMs were considered for the EMB: one that resulted in the omission of its output and another that produced an incorrect value. They were allocated with ASILs B and A in the best solution. However, the supplier can indicate that it has a component immediately available that meets ASIL B for both types of failures. Making use of this component would speed up development and it would be likely cheaper than requiring the development of a new component that meets the allocation from Table 10. Accepting the use of the supplier's component would mean revisiting the ASIL allocation process, namely to evaluate if the increase to the allocation of EMB's incorrect value FM allows decreasing the ASIL of some other FM in the system. The entire process can take several iterations until reaching a definitive allocation. In this scenario, the use of an automated and efficient approach, such as the one presented here, makes the process much simpler and faster.

Besides HBSM1, two other case studies were created from the HBS. HBSM2 and HBSM3 both consider braking in four wheels. The HBSM2 is meant to be just a direct conversion of HBSM1 to a four wheel version. In this way, it was possible to directly infer the optimal solutions for this model. HBSM2 considers the same hazards of HBSM1:

1. No braking in 1 wheel – ASIL D
2. Braking with wrong value in 1 wheel – ASIL A

For the HBSM3, a total of six hazards were defined:

1. No braking in 4 wheels – ASIL D
2. No braking in 3 wheels – ASIL D
3. No braking front – ASIL D
4. No braking rear – ASIL C
5. No braking diagonal – ASILC C
6. Braking with wrong value in 4 wheels – ASIL D

4.2 Case studies: The facts

The HiP-HOPS fault tree synthesis and analysis technique was applied to the system models of the four case studies considered in the thesis. For the sake of results reproducibility, the fault trees obtained for each case study have been included in Appendix III.

This section presents details about the optimisation problems that the case studies generate. Table 11 shows information about dimensionality, number of AACs and total solution space.

Table 11 - Characteristics of the optimisation problems posed by each case study.

Case Study	Dimensionality	#AACs	Total Search Space Size
HBSM1	24	31	5.96×10^{16}
HBSM2	60	94	8.67×10^{41}
HBSM3	60	11573	8.67×10^{41}
BSS	185	8218	2.04×10^{219}

HBSM1 provides problems of smaller dimensionality, and so it generates the smaller search spaces. HBSM2 and HBSM3 are based on the same architectural model; consequently they produce problems of the same dimensionality and with the same search space size. Optimisation problems based on HBSM3 include, however, ≈123 times more AACs than the problems from HBSM2. Finally, BSS creates the problems of higher dimensionality and generates by far the largest search spaces. Problems based on BSS include the second highest number of constraints, after problems based on HBSM3. Optimisation problems created from HBSM3 do not just integrate the highest number of AACs. They also include the most complex to evaluate. Table 12 shows the size of the AACs associated with each case study.

Table 12 - Number of AACs associated with each case study, organised by size.

	Size of AACs						
Case Study	*1*	*2*	*3*	*4*	*5*	*6*	*8*
HBSM1	11	20	-	-	-	-	-
HBSM2	29	65	-	-	-	-	-
HBSM3	9	17	72	1836	162	2916	6561
BSS	154	720	5040	2304	-	-	-

The maximum AAC size found is eight. HBSM3 is the only case study that generates optimisation problems with AACs of size five or greater. Furthermore, problems based on the HBSM3 integrate more AACs of the maximum size than of all other sizes combined. HBSM1 and HBSM2 generate the problems with the simplest AACs; in both cases there are only size one and size two AACs.

Higher dimensionalities define large search spaces. Furthermore, feasibility evaluation has an impact on the processing effort spent per iteration. The BSS is the case study of highest dimensionality and it creates problems with the second highest number of AACs. On the other hand, HBSM3 ranks second in regards to dimensionality, and the problems based on this case study include the highest number of AACs; these AACs are also the most complex. The characteristics of BSS and HBSM3 hint that they can be the two case studies imposing higher processing efforts.

The feasible search of a problem can be divided into two groups: 1) non-dominated solutions and 2) the solutions that are dominated by the first group. For HBSM1, HiP-HOPS exhaustive ASIL allocation technique has shown that there are 125 non-dominated solutions; the number of dominated solutions is, however, left to determine.

Consider Solution 1 in Figure 26, which is a non-dominated allocation for a hypothetical optimisation problem (Problem 1). The only AAC defined for Problem 1 requires that the total ASIL allocated to FM1, FM2 and FM3 needs to be at least ASIL 4. Solution 1, being non-dominated, is just outside of infeasible search space.

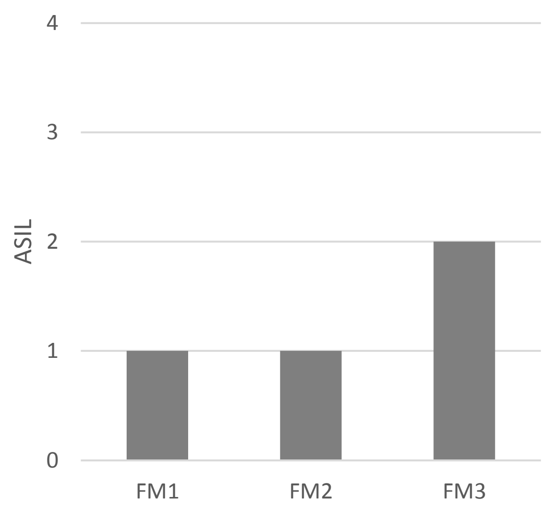


Figure 26 - Solution 1, a non-dominated solution for Problem 1.

Any solution with one or more FMs allocated with smaller ASILs than in Solution 1 is infeasible. On the other hand, all solutions that present one or more FMs allocated with higher ASILs than in Solution 1 are feasible; these are the solutions dominated by Solution 1. The number of allocations Solution 1 dominates can be easily calculated. It is given by the product of the ASIL allocation possibilities per FM, which are greater or equal to the ones of Solution 1; to the result of this calculation, one solution needs to be removed as it refers to Solution 1 itself: 4 × 4 × 3 - 1 = 47.

In determining the number of dominated solutions for a given problem, a couple of additional considerations need be introduced. Firstly, a dominated solution is not necessarily dominated by all non-dominated solutions. Observe the three solutions in Figure 27, which again relates to Problem 1. Solutions 2 and 3 are non-dominated - decrementing any of the ASILs allocated to their FMs would make them infeasible.

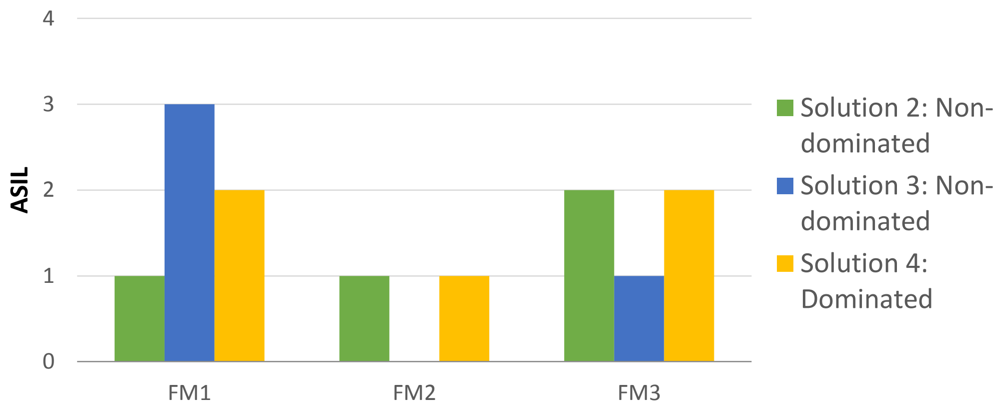


Figure 27 - A solution (solution 3) that is dominated by one non-dominated solution (solution 2), but that is not dominated by another non-dominated solution (solution 3).

Solution 4 is a dominated solution; however, it is dominated by Solution 2, but not by Solution 3. For FM2 and FM3, solutions 2 and 4 have the same allocations. FM1 is, however, allocated with a smaller ASIL in Solution 2 than in Solution 4. Solution 3, on the other hand, has two FMs allocated with lower ASILs than in Solution 4 (FM2 and FM3), but it has also one FM with a higher ASIL (FM1).

The previous example shows that it is not always the case, but different non-dominated solutions can dominate common allocations. Take the example of the three solutions for Problem 1 in Figure 28. Solutions 5 and 6 are non-dominated. It is easy to conclude that both of them dominate Solution 7 - they have lower ASILs across all FMs.

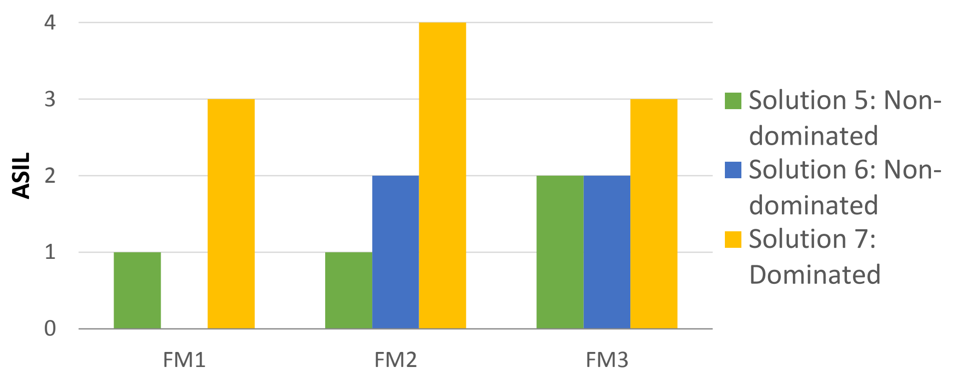


Figure 28 - Two non-dominated allocations (solutions 5 and 6) dominating a common allocation (Solution 7).

Finding the number of different solutions that are dominated by the group of non-dominated allocations of a problem is difficult. However, the sum of all the allocations that are dominated by each of the non-dominated solutions can be taken as a maximum bound. On the other hand, the maximum number of solutions dominated by one of the non-dominated allocations can be taken as the minimum bound. These bounds on the number of dominated solutions can be converted into bounds for the feasible region size of a problem - one only needs to add to them the number of non-dominated solutions.

For optimisation problems based on the HBSM1, the minimum bound for the feasible region is 2.78×10^{12}, whereas the maximum bound amounts to 9.98×10^{13}. Even if the maximum bound is considered, the search space is by far majorly infeasible; the feasible region represents less than 1% of the total search space (0.1675%). The HBSM1 generates optimisation problems with the smallest ratio of AACs count per failure mode and it is possible that the fraction of feasible space is even smaller for the remaining case studies.

4.3 Exploring the impacts of different cost heuristics in ASIL allocations

It is clear that the processes of developing one component according to any two different ASILs are distinct. There are different implications for implementation and validation, which impact, for example, number of lines of code or of safety analysis execution. Consequently, these have repercussions in development times and team sizes, which ultimately are translated into different development costs. One is necessarily looking for a requirements-compliant allocation which minimises these costs across a system architecture. This section explores the impacts of using different cost heuristics over the allocations seen as optimal for the HBSM1. Part of this work has been published in Azevedo *et al*. (2014a).

Two cost functions are utilised in this study. One is the Experiential-I heuristic presented in section 4.1; the second cost function, named Experiential-II, is presented in Table 13.

Table 13 - Experiential-II ASIL cost function.

Cost Function	ASIL QM	ASIL A	ASIL B	ASIL C	ASIL D
Experiential-II	0	5	30	35	50

All the non-dominated solutions for the HBSM1 are known and from them the optimal solution according to any cost function can be determined. Table 14 (see next page) shows the optimal solutions for the HBSM1 in the scenarios where the Experiential-I and Experiential-II functions are used to evaluate the allocations.

Table 14 - HBSM1 optimal ASIL allocations for Experiential-I and Experiential-II functions.

	Exp-I Cost: 390	**Exp-II** Cost: 340							
Components FMs	#1	#1	#2	#3	#4	#5	#6	#7	#8
Braking Pedal.OFailure1	4	4	4	4	4	4	4	4	4
Braking Pedal.VFailure1	1	1	1	1	1	1	1	1	1
Electronic Pedal.OFailure1	2	1	3	1	3	1	3	1	3
Electronic Pedal.OFailure2	2	3	1	3	1	3	1	3	1
Electronic Pedal.VFailure1	1	1	1	1	1	1	1	1	1
Electronic Pedal.VFailure2	0	0	0	0	0	0	0	0	0
Bus1.OFailure1	2	1	1	3	3	1	1	3	3
Bus2.OFailure1	2	3	3	1	1	3	3	1	1
WNC.OFailure1	2	1	1	1	1	3	3	3	3
WNC.OFailure2	2	3	3	3	3	1	1	1	1
WNC.VFailure1	1	1	1	1	1	1	1	1	1
WNC.VFailure2	1	1	1	1	1	1	1	1	1
Auxiliary Battery.OFailure1	2	1	1	1	1	3	3	3	3
Auxiliary Battery.VFailure1	1	1	1	1	1	1	1	1	1
Powertrain Battery.OFailure1	2	3	3	3	3	1	1	1	1
Powertrain Battery.VFailure1	1	1	1	1	1	1	1	1	1
EMB Power Converter.OFailure1	2	1	1	1	1	3	3	3	3
EMB Power Converter.VFailure1	1	1	1	1	1	1	1	1	1
IWM Power Converter.OFailure1	2	3	3	3	3	1	1	1	1
IWM Power Converter.VFailure1	1	1	1	1	1	1	1	1	1
EMB.OFailure1	2	1	1	1	1	3	3	3	3
EMB.VFailure1	1	1	1	1	1	1	1	1	1
IWM.OFailure1	2	3	3	3	3	1	1	1	1
IWM.VFailure1	1	1	1	1	1	1	1	1	1

The most obvious difference is that Experiential-I and Experiential-II reveal a different number of solutions with the same best cost: one and eight, respectively. A more detailed analysis reveals that none of the optimal solutions from Experiential-II includes the one from Experiential-I.

It could be the case that two ASIL cost heuristics can be defined that present the same best solution for the HBSM1 (or for some other case study, for that matter). However, with the example given in this section, it has been shown that this is not the general case. Work needs to be undertaken within the industry to define a plausible and widely accepted cost heuristic that unveils meaningful optimal ASIL allocations.

While an agreement is not achieved for a unified ASIL cost function, it is important that optimisation algorithms for the ASIL allocation problem are capable of dealing with any that the system designers finds more suitable. Moreover, ISO 26262 is still young (introduced in late 2011), and development is continuingly shifting from development from scratch according to a given ASIL, to use of Off-the-Shelf ASIL compliant parts; the most suitable cost function is therefore likely to change, and this further strengthens the need for a flexible optimisation technique. In an effort to test the capacity of the optimisation techniques developed to deal with different cost functions, the four case studies developed in the context of this thesis are combined with four different ASIL cost heuristics. The ASIL allocation benchmark is in this way composed of 16 problems in total. The cost heuristics considered are shown in Table 15.

Table 15 - ASIL cost heuristics.

	ASIL QM	ASIL A	ASIL B	ASIL C	ASIL D
Linear	0	10	20	30	40
Experiential-I	0	10	20	40	50
Experiential-II	0	5	30	35	50
Logarithmic	0	10	100	1000	10000

4.4 Methodology for optimisation techniques comparison

In comparing the metaheuristics it is important to define a stopping criterion, which allows for a similar processing effort to be spent across the different optimisation techniques. It is common to find approaches where the stopping criterion is based on iteration count. However, for the different algorithms presented in this thesis, the processing effort per iteration varies throughout the search process. For example, in

Tabu Search, depending on tabu moves restrictions, there is a varying number of variables being considered for ASIL change. Also, in PSO and GA, whenever a new solution is generated, only the AACs associated with the solution elements that have been changed are evaluated. Given these observations, it is not possible to know the processing effort per iteration for each of the algorithms and, in turn, it is not feasible to set an iteration limit for each technique that ensures a common processing effort. In this way, and taking into account the practical nature of the problem being addressed, execution time has been chosen as the stopping criterion. Algorithms are stopped after 5 minutes of execution time.

Due to the stochastic nature of the optimisation techniques, their performance is likely to vary between different trials. Thus, comparison is made on the basis of 50 algorithm runs. From the work of Murashkin *et al*. (2015) with exact solvers, the optimal solutions for the problems in the benchmark defined in this section are available. The optimal costs for each problem are presented in Table 16.

Table 16 - Optimal solution costs for the different problems in the ASIL allocation benchmark.

	ASIL Cost Heuristic			
Case study	*Linear*	*Experiential-I*	*Experiential-II*	*Logarithmic*
HBSM1	380	390	340	11300
HBSM2	920	930	790	13280
HBSM3	770	830	620	51150
BSS	4340	5370	5075	397040

Having access to the optimal allocations, instead of comparing the quality of the solutions obtained by the different techniques against each other's, all of them can be directly evaluated against the optimum. To compare the different algorithms, the percentage of runs returning an optimal solution will be examined. Another metric that will be used in the comparison is the average relative cost increase given by the best solutions found against the optimal allocations.

4.5 Summary

The chapter has presented a first benchmark for ASIL allocation optimisation. The benchmark is based on four case studies, which give problems with a variety of dimensionalities and of AACs characteristics. Different ASIL cost heuristics have impacts on the nature of the optimisation problem, and therefore four cost heuristics were considered together with each of the four cases studies; this gives a benchmark with a total of 16 problems.

The chapter has also demonstrated how the use of HiP-HOPS FTA capabilities together with an automated optimisation algorithm can be used to find cost efficient ASIL allocations on a complex scenario. The automated and efficient characteristics of the approach can support the iterative nature of ASIL allocation, namely in interactions with suppliers or in cases where system design changes are required.

In addition, with HBSM1 it has been possible to show that the use of different ASIL cost heuristics changes the considerations about what is the optimal ASIL allocation across the system. This conclusion could inspire work in the industry to arrive at a widely accepted cost heuristic. Such work would allow decision support approaches, such as the one presented in this thesis, to provide solutions which are closer to the “true” optimum solution.

Finally, the chapter has introduced a methodology for the comparison of the optimisation techniques presented in chapter 3. Since the algorithms include randomness in their search process, they are run 50 times for each problem.

Furthermore, given that the effort per iteration is not constant in any of the optimisation algorithms, the stopping criterion for an algorithm run is based on processing effort: 5 minutes have been arbitrarily defined as a maximum execution time. Two metrics will be used to compare the optimisation techniques: 1) the number of runs returning the optimal solution, and 2) the average relative cost increase against the optimum cost across the different problems.

5. Experimentation and evaluation

This chapter presents a series of experiments and their evaluation. The work targets, in its greatest part, objective 3 defined for the thesis: select the best metaheuristic optimisation technique from the ones developed in chapter 3, and evaluate the possibility of improving further its performance and enhance its usability. The chapter also provides further insights about the impacts of different cost functions over the most efficient ASIL allocations for a system architecture (objective 5).

5.1 The ASIL allocation lottery

ASIL allocation is a complex combinatorial optimisation problem, capable of generating very large solution spaces that cannot be explored in the context of exhaustive search. This section evaluates the possibility of finding good ASIL allocations by randomly generating solutions. Initially, the "ASIL allocation lottery" is compared to EuroMillions, a popular lottery game played across Europe.

In EuroMillions, players must choose five numbers from a range of 50 and two extra "lucky stars" from a set of 11. The number of different bets can be determined using the formula to calculate the number of available combinations when *s* elements are to be withdrawn from a set of size *n* (see Equation 11).

$$^{n}C_{s} = \frac{n!}{s!\,(n-s)!}$$ **Equation 11**

n! and *s!* stand for the factorial of *n* and s, respectively.

When choosing the five numbers, *n* is 50 and *s* is 5. From Equation 11, there are 2,118,760 combinations for the five numbers. On the other hand, when selecting two stars, *n* is 11 and *s* is 2; this gives 55 possible combinations of stars. The total number of betting possibilities is given by the product of the combinations for the numbers and the combinations for the stars. In total, 116,531,800 different betting possibilities are available. A player submitting a single bet would face chances of 1 in 116,531,800, or a probability of $\approx 8.58 \times 10^{-9}$ to win the EuroMillions. It would be interesting to compare these odds to the ones of winning the ASIL allocation lottery. For each of the case studies in the ASIL allocation benchmark, and using the linear cost heuristic, Table 17 shows the chances of hitting the optimal solution if one were to try to and "bet" randomly in a single allocation. They are calculated on the basis of the ratio between the number of optimal solutions for a case study and its total number of solutions.

Table 17 - Probabilities of single random bets finding optimal solutions for ASIL allocation problems.

ASIL Allocation Case Study	Number of optima	Total number of solutions	Probability	Probability Ratio EuroMillions / ASIL Allocation Case Study
HBSM1-Linear	100	5.96×10^{16}	1.68×10^{-15}	5.11×10^{6}
HBSM2-Linear	100	8.67×10^{41}	1.15×10^{-40}	7.44×10^{31}
HBSM3-Linear	9800	8.67×10^{41}	1.13×10^{-38}	7.60×10^{29}
BSS-Linear	3	2.04×10^{129}	1.47×10^{-129}	5.83×10^{120}

The last column in Table 17 shows the ratio between the probability of winning the EuroMillions and the probability of randomly finding an optimal ASIL allocation solution. One would have to be much luckier to win the ASIL allocation lottery than to become a millionaire with EuroMillions. The probability ratios regarding the HBSM2, HBSM3 and BSS case studies are even greater (much greater!) than the number of available solutions for EuroMillions.

Of course, for ASIL allocation there is no direct cost per bet; one is concerned with time consumption to generate a valid, low cost solution. Automation can speed up the process of generating and evaluating allocations, but exactly how good are the results if generated randomly? To investigate this, an algorithm has been created that produces a random ASIL for each FM in the system under analysis; by the time the entire solution is generated it is evaluated and its cost compared to the one of the optimal allocation(s). The algorithm was applied to each of the case studies shown in Table 17. For each one of them, the method was left to run for an entire day. Results are presented in Table 18.

Table 18 - Results of using automation to produce random ASIL allocations for the problems of Table 17. The quality of infeasible solutions is not compared; for cases where feasible solutions were not found, dashes are used for the columns referring to cost, iterations and time of best solution found.

Case Study	Optimal Cost	Feasible Found?	Cost Best	Iteration Best	Time Best (seconds)	Total Iteration
HBSM1-Linear	380	Yes	470	2.93×10^6	1.17	2.19×10^{11}
HBSM2-Linear	920	No	-	-	-	8.95×10^{10}
HBSM3-Linear	770	Yes	1370	2.25×10^8	355.06	5.87×10^{10}
BSS-Linear	4340	No	-	-	-	2.96×10^{10}

A vast number of solutions were produced for each of the experimental setups. However, the optimal solutions of the cases studies were not found. Furthermore, for HBSM2 and BSS case studies no feasible solution could be generated. For HBSM1, the simplest case study, a feasible solution was found, but there is still a cost increase of 90 cost units in comparison to the optimal solution (corresponding roughly to 24% of cost increase). For HBSM3, the best feasible solution found is far more expensive than the optimal (approximately 78%).

These results clearly demonstrate that ASIL allocation is not a trivial problem that can be solved purely resorting to luck; not even when luck gets help from a few gigahertz of automated processing power. Furthermore, the fact that in some cases the random solution generation approach did not even find feasible solutions is symptomatic of the severely constrained nature of ASIL allocation optimisation. This further supports the evidence presented in chapter 4 when it was determined that the feasible region size of problems based on HBSM1 amounted to less than 1% of the total search space.

5.2 Metaheuristics - The survival of the fittest

In chapter 3, a set of metaheuristic-based optimisation techniques have been developed to tackle ASIL allocation problems. The current section presents a comparison of their performance across the ASIL allocation benchmark defined in chapter 4.

5.2.1 Parameterisation of the species

Each of the optimisation algorithms includes different parameters that need to be set and that affect the performance of the algorithms. Experiments with values common in the literature were tried, but parameterisations were often changed to impose specific behaviours to the algorithms and also on the basis of trial and error. This section shows the best parameterisations found after a large number of experiments for each technique.

Genetic Algorithm and Particle Swarm

GA and PSO share a common approach for the dynamic penalisation of infeasible solutions. The penalisation approach consists of two sequential parts, *Pen-I* and *Pen-II*. In Pen-I, the penalty to infeasible solutions starts relatively small and it is increased with iteration count. During Pen-II, penalty to infeasible solutions is increased or reduced, depending on whether recent best solutions in the population are mostly infeasible or feasible. Control of how severely infeasible solutions are penalised is achieved through the *Near Feasibility Threshold* (*NFT*) parameter. If the *NFT* increases, the penalty to an infeasible solution is reduced; on the other hand if the *NFT* decreases, the penalty grows. For Pen-I a value needs to be set for the initial *NFT*. In the approach followed here, the initial *NFT* is calculated so that in the first iteration, the most infeasible solution (corresponding to all allocation variables being assigned with ASIL 0) gets penalised with a value equivalent to a percentage, τ, of the most expensive feasible solution cost (scenario where every variable is allocated with ASIL 4). This allows for the values of the penalty to hold meaning in the context of the cost heuristic being used and the size of the problem – together, these two factors determine the cost of the most expensive feasible allocation. Furthermore, this approach also enables a greater initial infeasibility exploration allowance in problems where solutions can present a higher *Total Degree of Infeasibility* (*TDI*). Consider two illustrative problems, Problem 1 and Problem 2, which are of the same dimensionality. If the same cost function is used to evaluate the allocations, the cost of the most expensive solution is the same in both problems. Following the approach described for initial *NFT* calculation, the two most infeasible solutions will be initially penalised with a value corresponding to a percentage, τ, of the most expensive feasible solution cost. Since the costs of the most expensive feasible solution is the same for both problems, the two most infeasible solutions present the same total cost as well (assuming a fixed τ). However, consider that the most infeasible solution for Problem 1 presents a higher *TDI* than the most infeasible solution for Problem 2; this means that the initial *NFT* for Problem 1 is higher than the one obtained for Problem 2. This reasoning can be better understood by recovering the expression for penalty calculation (see Equation 12).

$$Penalty = \left(\frac{TDI}{NFT}\right)^{\beta}$$ **Equation 12**

If the penalty to apply to the most infeasible solutions of Problem 1 and Problem 2 is the same, and the *TDI* of the most infeasible solution is higher for Problem 1, then the *NFT* for Problem 1 must be higher as well. Finally, if the initial *NFT* is higher for Problem 1, this means that for the first algorithm iteration, a solution with a given *TDI* for Problem 1 will be less penalised than a solution with the same *TDI* for Problem 2.

The parameterisations of the penalty approach for GA and PSO can be observed in Table 19 and Table 20, respectively. Except for τ, which has only been defined in the current section, the different penalisation parameters have been introduced in detail in section 3.4.4. In order to provide for a better understanding of the tables, they are briefly described now: β is a pre-defined severity parameter used both during Pen-I and Pen-II - the penalty to apply is the result of the division between a solution's *TDI* and the current *NFT*, raised to the power of β; α determines how fast the *NFT* decreases during Pen-I*;* δ is the period for *NFT* update during Pen-II; finally, ε is used during Pen-II to scale the *NFT*'s decrease or increase needs depicted by the proportion of infeasible solutions over the last δ iterations.

Table 19 - Penalty approach parameterisation for GA.

β (Pen-I & Pen-II)	α (Pen-I)	τ (Pen-I)	δ (Pen-II)	ε (Pen-II)
1.5	0.005	1%	400	100

Table 20 - Penalty approach parameterisation for PSO.

β (Pen-I & Pen-II)	α (Pen-I)	τ (Pen-I)	δ (Pen-II)	ε (Pen-II)
1.5	0.01	5%	100	100

The penalty settings of PSO define a more volatile change in what areas are favoured in the solution space (higher *NFT* decrease rate during Pen-I (α) and smaller period for Pen-II (δ)). PSO's Local Search (LS) tactic is likely to play an important role here. Whilst initially integrated in PSO with the purpose of dealing with discretisation errors, LS concurrently allows, to some extent, for an improvement driven exploration of the area around a particle. This can be important to make a particle travel more easily into the areas favoured by the penalty function and also for the algorithm to be more focused and converge faster. The GA technique, on the other hand, only makes use of stochastic operators - crossover and mutation - possibly making it harder to travel between penalty-favoured regions and weighting for a slower convergence. Another interesting aspect is that the penalty to apply to infeasible solutions is initially smaller in GA (the τ value is smaller). This need for a higher infeasibility exploration allowance can again be linked to the random nature of the operators in GA; in finding the most promising solution of a given area, GA's stochastic process of exploration may require going through a set of solutions that are more infeasible that in the case of PSO. Finally, setting ε=100 seemed to work well for both algorithms. The need for a higher infeasibility allowance in GA might be shadowed here by the adaptive nature of Pen-II.

Having introduced the best parameterisations for the penalty method, it is now time to present the parameterisations for the algorithms themselves. The GA parameterisation is shown in Table 21.

Table 21 - GA parameterisation.

	Population	***Children***	***Uniform crossover probability***	***Mutation probability***	***Mutation rate***
All HBSM	3	5	30%	60%	1%
BSS	15	50			

A bigger population and a higher number of children were required for the large BSS. For this case study, the use of populations as high as 50, and as low as 5 obtained solutions of the same quality, with minor execution time variations (keeping the offspring to 50). It must be said that manipulation of the penalty parameterisation, by choosing smaller *NFT* decrease rates during Pen-I and higher *NFT* update periods for Pen-II, allowed for smaller offspring sizes to be set for BSS. In these circumstances, intensification over a penalty favoured region lasted more generations, but with less solution evaluations per generation. However, experiments with such parameterisations did not reveal improvements over the results shown here. In regards to the different HBS variants, running GA with both parameter settings from Table 21 resulted in finding equally good solutions. Nonetheless, selecting the smaller sizes for the population and offspring obtained the solutions substantially faster, although, in general, after a higher number of generations.

In regards to PSO, the parameterisation adopted is depicted in Table 22.

Table 22 - PSO parameterisation.

Swarm size	c_1	c_2	w	***LS probability***	***LS period***	***Mutation probability***	***Mutation rate***
10	1.2	0.6	0.729	50%	5 iterations	50%	1%

Throughout experiments, mutation showed to be valuable in efficiently and effectively introducing diversity in the search. It plays an essential role in the current parameterisation as well; if set to zero, the number of algorithm runs retrieving optimum solutions is dramatically reduced.

Research on how to select values for c_1, c_2 and w has been conducted in the past by a variety of authors (Eberhart & Kennedy, 1995; Clerc, 1999; Clerc & Kennedy, 2002;

Trelea, 2003). Clerc (1999), for example, recommends: c_1=1.494, c_2=1.494 and w=0.729. The PSO algorithm presented here, however, extends the original technique with different operators, namely LS. The former seems to have a special influence in the selection of c_1 and c_2. LS around a particle's position can provide an extra contribution for the exploration of the area surrounding the particle's *pbest* and of the swarm's *gbest*, perhaps with different weights depending on parameterisation. Determining a relationship between LS and the selection of c_1 and c_2 still requires further investigation. A good performance across the benchmark has been found experimentally by setting c_1 and c_2 to 1.2 and 0.6, respectively, while performing LS to 50% of the swarm (5 particles) with every 5 iterations.

Tabu Search

The parameterisation utilised with TS is presented in Table 23.

Table 23 - TS parameterisation.

updatePeriod$_p$	***updatePeriod$_q$***	***limit$_p$***	***limit$_q$***
10	11	2n	2n

Imposing long tabu restrictions to certain solution elements naturally means that more changes can take place elsewhere. Conceptually, forbidding descent moves for a relatively long period, thus locking parts of the solution to higher ASILs for that duration, can be important in providing a greater degree of freedom for the remaining parts to change; as a result, the solution can travel to a different area in the search space. On the other hand, locking solution components to lower values for long, might allow for good parts of the solution to be conserved, even if not recognised as such in the current search state. In both cases, however, it is important for these moves restrictions not to be present for unnecessarily long periods, as they might prevent convergence whilst in areas of good solutions. The most promising results were obtained across the benchmark when descent and ascent moves were forbidden up to two times the problem dimensionality. Furthermore, although in the best parameterisation found, $updatePeriod_p$ and $updatePeriod_q$ are set to 10 and 11, respectively, some efficient performances were also obtained, for example, with the parameterisation suggested by Hansen and Lih (1996): ($updatePeriod_p$; $updatePeriod_q$) = (3; 4). Across experiments, this approach of setting $updatePeriod_q$ to a value one unit greater than $updatePeriod_p$ provided good results. Also, it seemed important not to choose values for these parameters that are either too big or too small. High values mean that the same memory state gets applied consecutively to a big number of solution components and too many solution parts could get locked for too long, for example. Low values could also lead to undesired effects, such as not locking enough parts of the solution for a period that allows for the rest to converge into a good solution in the area.

5.2.2 And the fittest species is…

This section presents a comparison of the three most promising metaheuristic optimisation techniques presented in chapter 3. In section 4.4, a comparison methodology had been defined, which was based on the quality of the solutions each technique finds. However, the final versions of the three algorithms discover the optimal solution for each problem in the benchmark, in 50 out of 50 runs. These results are obtained under the previously defined limit of 5 minutes. Model-based development facilitates a highly dynamic design process, where changes to a system architecture can be easily performed and immediately evaluated in terms of cost, safety, weight, etc. This type of development meets modern system development needs, namely evaluating multiple design solutions for a same system architecture or dealing with changes in requirements. Although the three optimisation techniques can complete the problems in the benchmark in reasonable times, it would be important to identify the most efficient. Using the fastest technique would favour further a dynamic design process - ASIL allocation information would be more readily available for the user to make a decision on the architecture it has conceived at a given design iteration. Highlighting the most efficient technique is also important in the light of dealing with future, more complex problems. In this way, this section compares the GA, PSO and TS optimisation techniques on the basis of the average time they take to complete the problems in the benchmark, over 50 runs. Experiments were conducted on a machine equipped with an Intel i5-3570K processor clocked at 3.40GHz and with 8GB of RAM.

The results from the experiments carried out are presented in Tables 24, 25, 26 and 27. Each table relates to one case study of the ASIL allocation benchmark.

Table 24 - Results for HBSM1 problems.

Algorithm	Cost Function	Min Time (s)	Avg Time (s)	Max Time (s)	Std Dev Time (s)
GA	Lin	0.006389	0.0124880	0.030777	0.0063719
	Exp-I	0.006965	0.0558921	0.219299	0.0492780
	Exp-II	0.006442	0.0227274	0.084308	0.0197152
	Log	0.003465	0.0137471	0.149952	0.0228430
PSO	Lin	0.005056	0.0057361	0.006407	0.000325
	Exp-I	0.005034	0.0059819	0.006675	0.0003897
	Exp-II	0.005066	0.0058188	0.006787	0.0003109
	Log	0.001433	0.0030475	0.005409	0.0010558
TS	Lin	0.000025	0.0000289	0.000038	0.0000024
	Exp-I	0.000028	0.0001227	0.000513	0.0001033
	Exp-II	0.000025	0.0000361	0.000061	0.0000104
	Log	0.000025	0.0000579	0.000201	0.0000323

Table 25 - Results for HBSM2 problems.

Algorithm	Cost Function	Min Time (s)	Avg Time (s)	Max Time (s)	Std Dev Time (s)
GA	Lin	0.036079	0.7235642	3.020296	0.7235532
	Exp-I	0.020858	0.2526737	1.358449	0.2863532
	Exp-II	0.015580	0.6781136	2.734967	0.6140482
	Log	0.010254	0.0290923	0.145465	0.0285950
PSO	Lin	0.012894	0.0437167	0.142182	0.0367445
	Exp-I	0.012247	0.0142772	0.016915	0.0009266
	Exp-II	0.012960	0.1510058	0.928491	0.1552747
	Log	0.004181	0.0097878	0.014618	0.0020801
TS	Lin	0.000083	0.0042940	0.025781	0.0051581
	Exp-I	0.000087	0.0011129	0.005599	0.0013986
	Exp-II	0.000083	0.0019743	0.009872	0.0025251
	Log	0.000074	0.0001870	0.001087	0.0001448

Table 26 - Results for HBSM3 problems.

Algorithm	Cost Function	Min Time (s)	Avg Time (s)	Max Time (s)	Std Dev Time (s)
GA	Lin	0.071032	0.4492811	1.578897	0.3992775
	Exp-I	0.073513	2.1811851	9.148364	2.2033160
	Exp-II	0.070480	0.5963541	3.985500	0.7792485
	Log	0.073740	0.8116967	27.011508	3.7773049
PSO	Lin	0.351748	0.6739416	1.480792	0.2762455
	Exp-I	0.397512	1.0775289	2.820434	0.7179697
	Exp-II	0.335210	0.5084100	1.967718	0.3163233
	Log	0.313197	0.3584900	0.466582	0.0253947
TS	Lin	0.001816	0.0166488	0.082580	0.0174064
	Exp-I	0.001927	0.0860322	0.357000	0.0875105
	Exp-II	0.001953	0.0258641	0.156816	0.0278785
	Log	0.001970	0.0058873	0.058060	0.0088219

Table 27 - Results for BSS problems.

Algorithm	Cost Function	Min Time (s)	Avg Time (s)	Max Time (s)	Std Dev Time (s)
GA	Lin	0.726641	0.7764428	0.873860	0.0370999
	Exp-I	0.752787	0.7941502	0.874269	0.0276522
	Exp-II	0.764339	0.8122183	0.896643	0.0342757
	Log	0.795020	0.8408713	0.953727	0.0326558
PSO	Lin	0.250719	0.4454349	8.490209	1.1494235
	Exp-I	0.255177	3.3664704	60.044311	11.3723530
	Exp-II	0.240792	0.2680211	0.297390	0.0120433
	Log	0.202960	0.2194469	0.235205	0.0086264
TS	Lin	0.008472	0.0490931	0.131184	0.0237989
	Exp-I	0.014696	0.0870871	0.194864	0.0541084
	Exp-II	0.001047	0.0229366	0.115203	0.0230289
	Log	0.001015	0.0317221	0.343760	0.0712825

Firstly, it is important to highlight how all of the optimisation techniques give very acceptable performances. Results show that, across the different algorithms, the maximum time taken to complete a problem amounts to roughly 60.04 seconds (PSO for BSS-Exp-I). Furthermore, the maximum average execution time is 3.37 seconds (again, obtained by PSO for BSS-Exp-I).

To facilitate comparison, charts have been created which plot the techniques' average processing times for the different problems in the benchmark. There are four charts and each relates to tests with a particular case study (see Figures 29, 30, 31 and 32). Attention must be brought to the fact that charts use a logarithmic scale. In fact, in some problems, the different techniques present very different processing times; if a logarithmic scale is not used, the order of magnitude of some results is not recognisable.

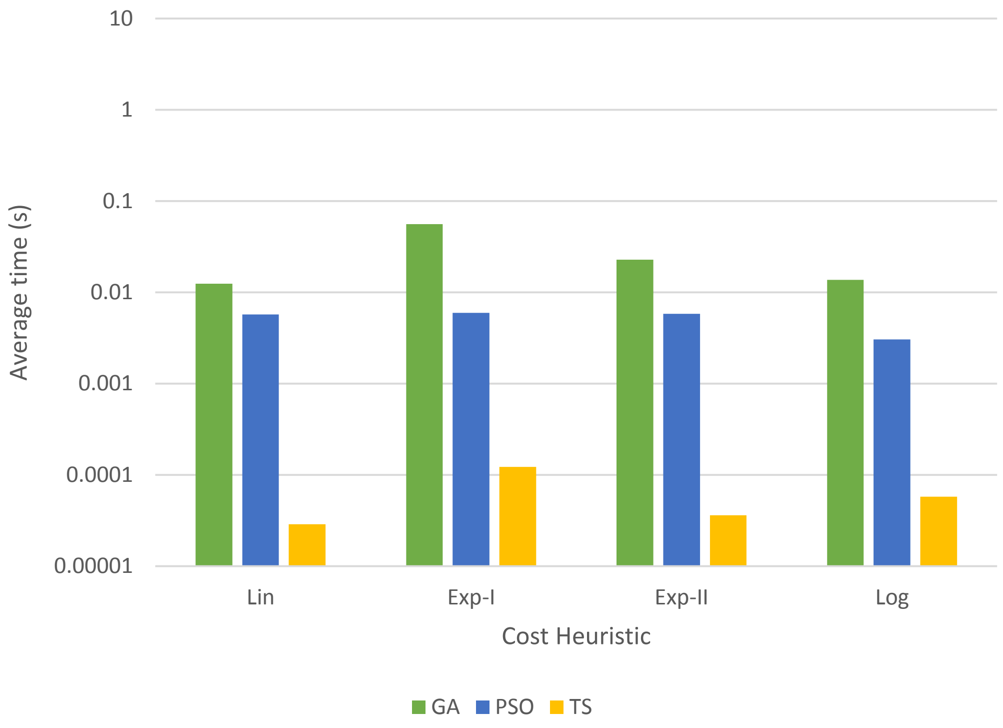


Figure 29 - Average processing times of GA, PSO and TS to complete HBSM1 problems. Smaller is better.

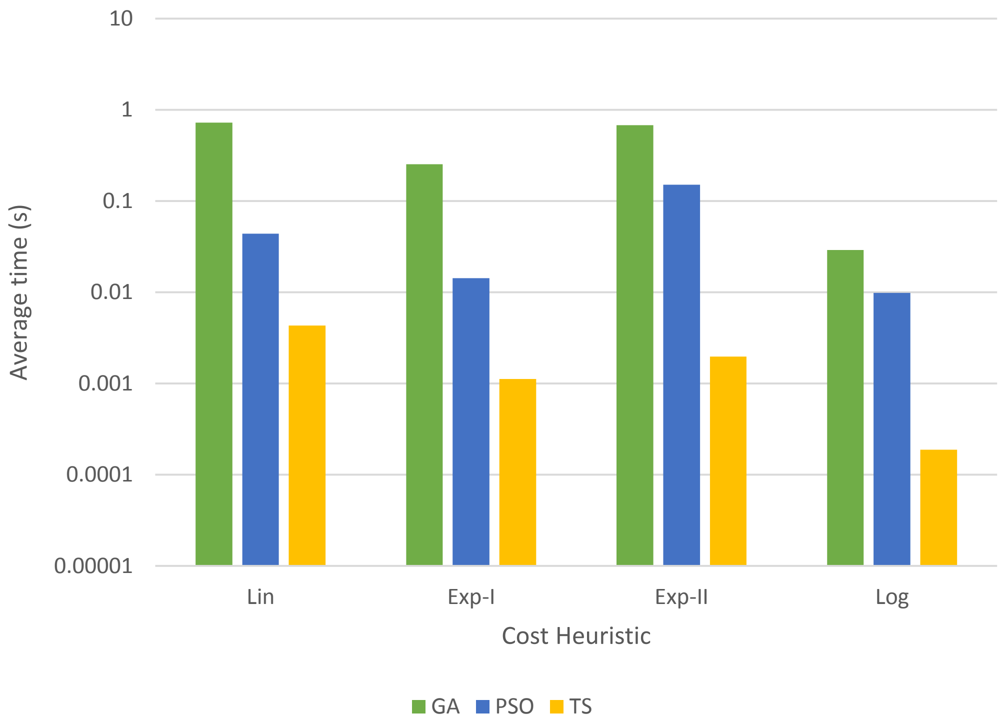


Figure 30 - Average processing times of GA, PSO and TS to complete HBSM2 problems. Smaller is better.

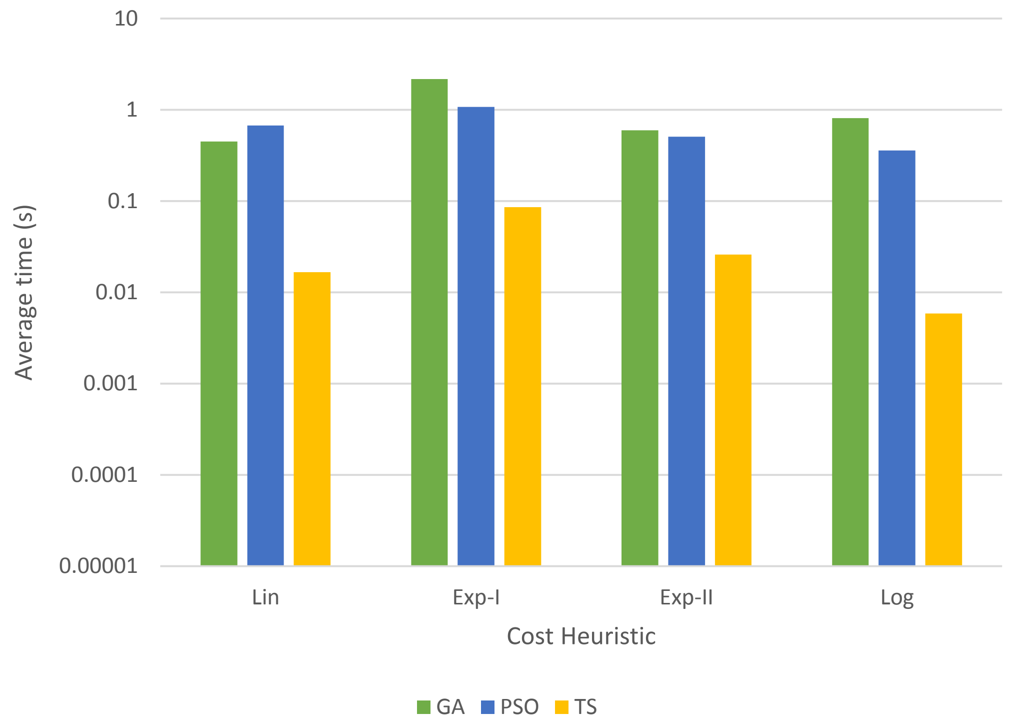


Figure 31 - Average processing times of GA, PSO and TS to complete HBSM3 problems. Smaller is better.

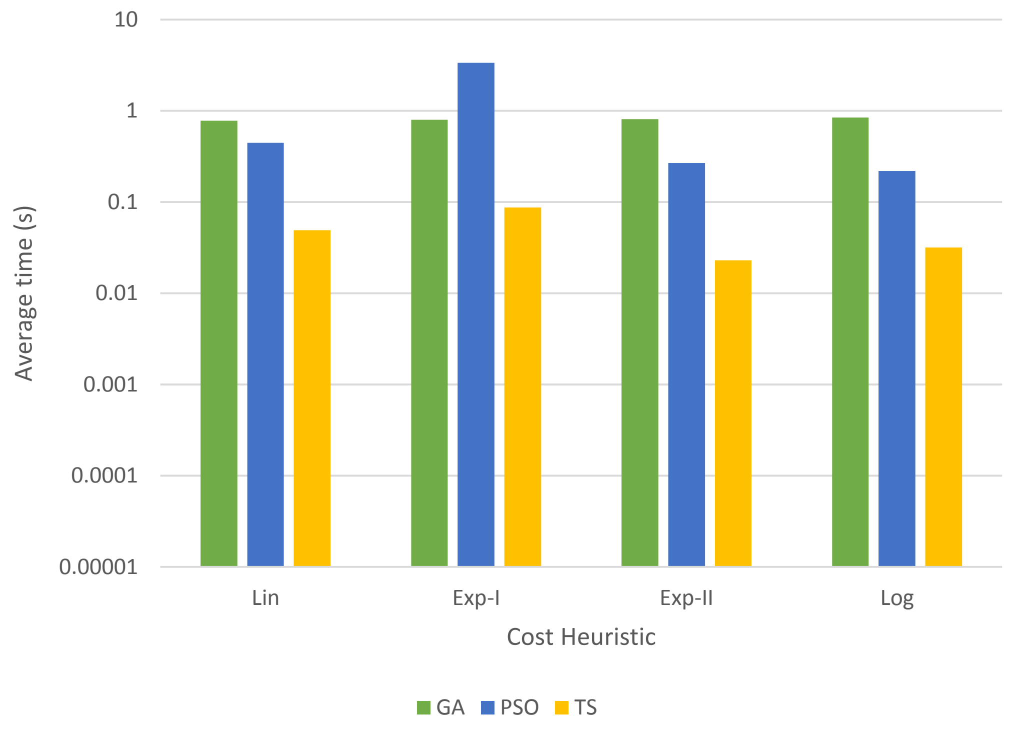


Figure 32 - Average processing times of GA, PSO and TS to complete BSS problems. Smaller is better.

The TS technique presents the fastest average processing times across all 16 problems defined. TS is at least one order of magnitude faster than GA in all problems but one (BSS-exp-I, for which it still is nine times more efficient). In turn, TS is faster than PSO by at least one order of magnitude in all problems but two: BSS-lin and BSS-log. However, for these two problems, TS remains significantly quicker; it is nine times faster than PSO for the first and it finishes the second seven times quicker. Comparing PSO to GA, the former is faster in 14 problems; GA only overcomes PSO for HBSM3-lin and BSS-exp-I. The differences between GA and PSO are generally less significant than when the two are compared to TS; HBSM2-lin and HBSM2-exp-I are the only problems for which one of the techniques is more than 10 times faster than the other (PSO being the fastest in these two cases).

Analysis of the tables of results gives more evidence of the overwhelming superiority of TS over GA and PSO, when applied to these problems. The average times of TS are smaller than the minimum times of PSO in every problem; in turn, only for one problem, TS's average time is greater than GA's minimum. Furthermore, even if TS's maximum times are taken, they are smaller than PSO and GA's minimum times in 14 and 13 problems, respectively. This means that even in the scenario where TS is guided by "the worst case randomness", and PSO and GA are steered by "the best case randomness", TS is still better for the majority of tests. Another interesting fact is that TS's maximum times are smaller than PSO's average times in every problem but one; in regards to GA's average times, in every problem TS's maximum times are smaller.

Analysing the total average time across the benchmark, it is no surprise that TS is the fastest technique. It presents an approximate total time of ≈0.33 seconds; PSO is the second fastest technique with ≈7.16 seconds, and GA comes last with ≈9.05 seconds. Across the benchmark, TS is ≈27 times faster than GA and ≈21 times faster than PSO.

TS's performance superiority in this study is clear. It is worth, however, underlying yet another important advantage of the technique: it provided a much easier tuning experience. Whereas TS integrates only four parameters to select, GA and PSO (counting penalty settings) include 10 and 13 parameters, respectively. It is important to point out that although GA includes a lower number of parameters than PSO, it seemed more susceptible to the effects of parameter selection and more effort was needed to make GA achieve its best results.

TS combines the best performances across every problem in the benchmark with the easiest tuning experience. It emerges as the far most promising technique from this study. The second most promising technique is PSO, which presents more efficient processing times than GA for 14 out 16 problems in the benchmark. Furthermore, it finishes the entire benchmark faster than GA. Finally, although PSO includes the largest number of parameters to select, it provided a better tuning experience than GA.

Having established TS as the most promising technique, a few possible explanations for PSO and GA's less promising performances are now advanced.

One first possible contributing factor is related to the processing effort associated with GA and PSO's population-based approaches and the size of the problems tackled. The results from the benchmark reveal that all the cases where GA and PSO are slower than TS by less than one order of magnitude are linked to the BSS case study. Additional analysis tells that on average, for both algorithms, this is the case study for which the relative differences to TS are smaller: PSO is on average ≈17 times slower than TS across the BSS problems, whereas GA is ≈22 times slower. The case study where the relative differences are greatest is HBSM1, for which TS is on average ≈115 times faster than PSO and ≈438 times faster than GA. The relative differences between TS and the two remaining techniques are then maximum for the smallest case study, HBSM1, and minimum for the one posing the biggest search space, BSS. This could be an indication that the processing effort from GA and PSO's population-based approaches is especially excessive for the smaller problem in the benchmark, but it starts paying off with the increase of search space size. TS is an trajectory-based metaheuristic; if the initial solution generation step determines that the optimisation process starts far from the optimal solution, search will have to evolve between solution regions, to some extent blindly, until the one from the optimum is reached. In the case of population-based algorithms, multiple search space regions can be examined initially and optimisation can progress towards the most promising one, which hopefully contains the optimum. Whereas in exploring larger solution spaces the population-based approach can present benefits, for smaller problems having multiple solutions scanning the search space will more likely represent unnecessary processing effort. That being said, although the relative differences between TS and the population-based approaches have their minimum for the largest case study, BSS, they are still substantial; in the future, it could be interesting to study larger case studies to evaluate if the relative differences between TS and the two population-based approaches can become more comparable.

One other possible cause for the less efficient performances of PSO and GA is related to how infeasible solutions are penalised. Consider an illustrative Problem 1 with two variables, $ASIL_{FM1}$ and $ASIL_{FM2}$. Problem 1 integrates the three AACs given next.

- AAC1: $\text{ASIL}_{\text{FM1}} \geq$ ASIL 3
- AAC2: $\text{ASIL}_{\text{FM1}} \geq$ ASIL 3
- AAC3: $\text{ASIL}_{\text{FM2}} \geq$ ASIL 3

Now consider two infeasible solutions for the same problem:

- Solution 1: ASIL_{FM1} = ASIL 1; ASIL_{FM2} = ASIL 3
- Solution 2: ASIL_{FM1} = ASIL 3; ASIL_{FM2} = ASIL 0

A solution's degree of infeasibility concerning a given AAC is calculated from the difference between the AAC's ASIL requirement and the ASIL achieved by the solution for that AAC. In turn, the *TDI* of a solution is given by the sum of the degrees of infeasibility across all the AACs of the problem. Solution 1 fulfils AAC3, but it violates AACs 1 and 2. AACs 1 and 2 have ASIL 3 as requirement, and for both, Solution 1 achieves ASIL 1 through ASIL_{FM1}. Hence, for each of these AACs, Solution 1 presents a degree of infeasibility of two units. In turn, the TDI of Solution 1 is the sum of the degrees of infeasibility for AACs 1 and 2, which amounts to four. Solution 2 fulfils AACs 1 and 2, and it violates AAC3. Since only AAC3 is not satisfied, the degree of infeasibility presented for this AAC gives the *TDI* of Solution 2. The ASIL requirement for AAC3 is ASIL 3 and the ASIL obtained for that AAC is ASIL 0 (through ASIL_{FM2}); the *TDI* for Solution 2 amounts to three units. Solution 1 presents a *TDI* of four, whereas Solution 2 has a *TDI* of three units. The penalty applied to a solution grows with the *TDI* it presents; as a result, Solution 2 will be favoured by the penalty function and the optimisation algorithm will possibly consider it as representing a more promising area in the search space. However, a closer look at the two solutions tells that Solution 1 is actually closer to feasibility than Solution

2. Observe Figure 33, which depicts the relationships between the AACs and the allocation variables of Problem 1.

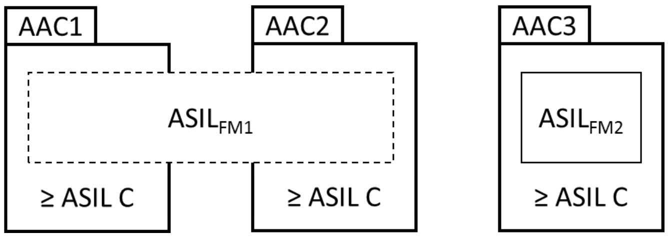


Figure 33 - AACs for Problem 1. $ASIL_{FM1}$ is shared by AAC1 and AAC2.

$ASIL_{FM1}$ belongs both to AAC1 and AAC2; as a consequence, it simultaneously contributes to satisfying the requirements of the two AACs. In Solution 1, a variable ($ASIL_{FM1}$) only needs to be incremented two units so that the solution becomes feasible; in Solution 2, on the other hand, $ASIL_{FM2}$ needs to be incremented by three units. Arguably, Solution 1 should be less penalised and the area around itself favoured for exploration.

Despite situations like the one of solutions 1 and 2, the penalty approach adopted in this thesis allows for optimal solutions to be found and within very acceptable times. In fact, there are scenarios for which the approach makes valid inferences. Consider a third solution for Problem 1:

- Solution 3: $ASIL_{FM1}$ = ASIL 3; $ASIL_{FM2}$ = ASIL 2

Solution 3 only violates AAC3, and it does so by one unit; in turn, this leads to a TDI of the same value. Solution 3 is less penalised than Solution 2, which has a TDI of three units. The higher penalisation for Solution 2 makes sense in terms of distance to feasibility; in Solution 3, incrementing $ASIL_{FM2}$ by one unit will be enough to reach feasibility, whereas in Solution 2, $ASIL_{FM2}$ needs to be incremented by three units. In this way, the penalty approach used in this thesis, although not perfect, remains as one that allows for some valid considerations and for good results to be achieved. It is possible, however, that a method that penalises a solution by determining how far it rests from feasibility can guide the algorithm better in exploring infeasible search space, and enables reaching optimal solutions in less iterations. Developing such a method promises to be a non-trivial task, nevertheless. As illustrated with Solution 1, in calculating the distance of a solution to feasibility, one needs to take into account the impacts of a given allocation variable in satisfying all the AACs it integrates. Determining a solution's distance to feasibility emerges as an optimisation problem itself, which consists of finding the minimum total increase to allocation variables that allows meeting all the AACs requirements. Solving this problem will likely require additional significant processing effort. Therefore, even if the optimisation algorithms can be better guided through infeasibility it is doubtful that the resulting approaches will be more efficient than the GA and PSO techniques presented in this thesis.

A last possible contributing factor for the less efficient performances presented by PSO and GA is related to the parameterisation experience they offer. A much greater amount of effort was dedicated to tuning these two approaches in comparison to TS. Given the number of parameters that need selection in PSO and GA, it is more likely that their performances are further away from the best that can be obtained, than in the case of TS.

Finally, a brief consideration about the superiority of PSO over GA. PSO's more promising results can possibly be linked to its improve-driven local search approach. This mechanism seems to give a quicker, but still effective, exploration of different regions in the search space.

5.3 ASIL allocation - Optimisation by divide and conquer

In tackling complex problems, it is common to find approaches that follow a divide and conquer methodology, i.e., they work by decomposing the original problem into multiple, simpler sub-instances and solving those independently. The solutions of the sub-instances are then combined to provide a solution for the original problem. In the literature, the definition of *divide and conquer* approaches is often more focused. For example, it is common to find it linked to recursion: problems are to be iteratively decomposed into sub-instances and then solved recursively (Dasgupta *et al.*, 2008). In this thesis, however, the definition adopted for *divide and conquer* merely refers to a procedure where a problem is divided and the resulting sub-instances solved to provide a solution for the original problem.

A divide and conquer approach has been devised for tackling ASIL allocation problems. The rationale for further improvements to the optimisation performance is that it will enhance the scalability of the allocation approach proposed in this thesis. This section starts by explaining how ASIL allocation problems can be divided into independently solvable sub-instances, which are themselves of the same type of the original one; subsequently, the applicability of the division method to the problems of the ASIL allocation benchmark is demonstrated; finally the performance of the overall divide and conquer optimisation process, making use of Tabu Search to conquer the sub-problems, is evaluated.

5.3.1 A guide to divide and conquer

Up to this point, optimisation algorithms have been inputted simultaneously with all the decision variables and all the ASIL Allocation Constraints (AACs) defined for a given allocation instance. In each of their runs, they would be searching for a promising solution that applies to the entire problem. Figure 34 provides an overview of the optimisation process used until now when applied to the problem posed by an illustrative System 1 (S1). The problem contains six variables and five AACs. ASIL cost information is neglected at this point for simplicity.

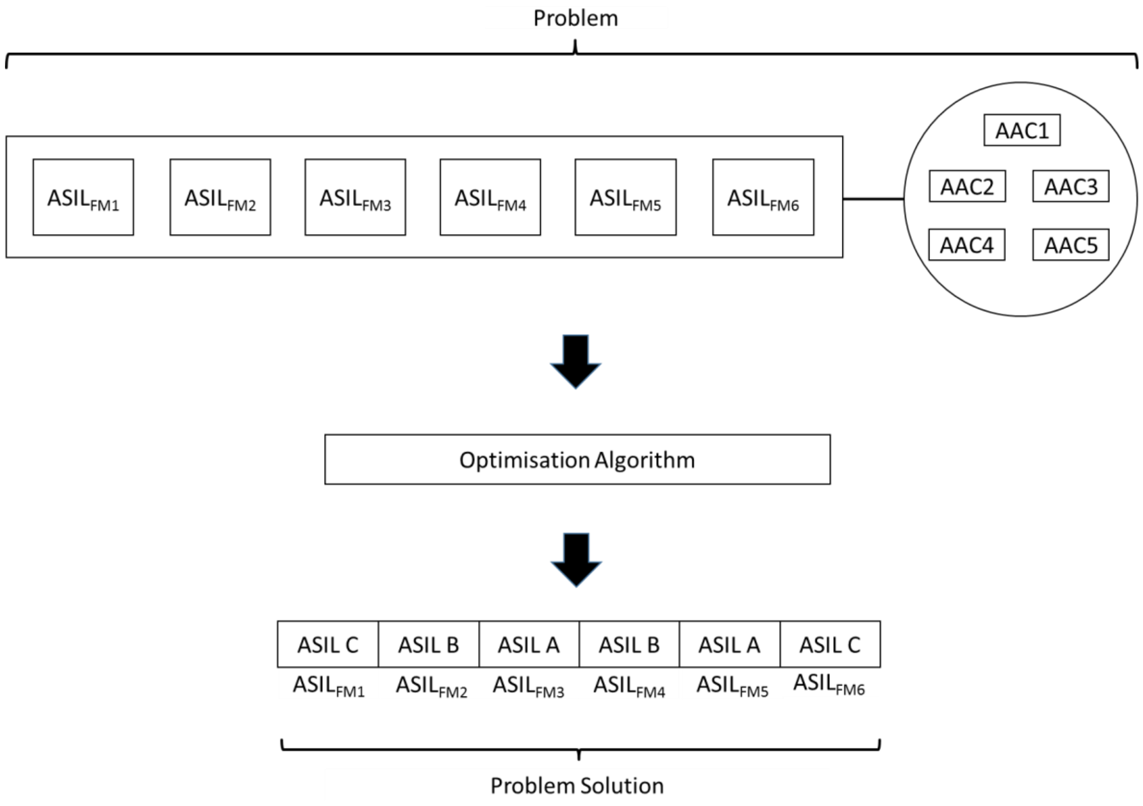


Figure 34 - ASIL allocation optimisation process for the problem of an illustrative System 1 (S1). The entire problem is solved in one step by the optimisation algorithm.

Although it is possible to solve the problem in the fashion presented in Figure 34, an analysis of the AACs defined for a system can identify subsets of decision variables whose allocations are independent from each other's, and that in fact pose optimisation problems that can be solved separately. This is the premise for the divide and conquer approach to ASIL allocation. Consider the AACs information for S1, which is presented in Figure 35.

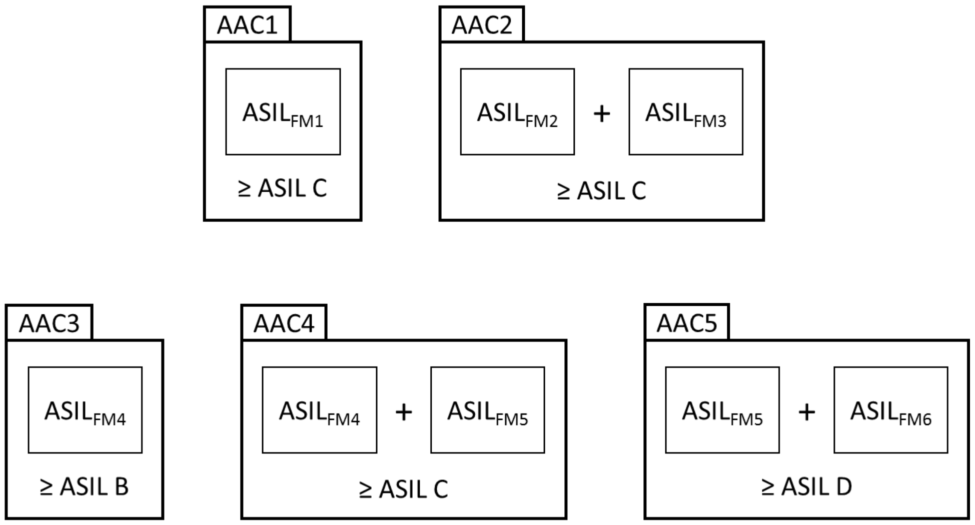


Figure 35 - AACs of S1.

$ASIL_{FM1}$ has the responsibility of meeting on its own the ASIL requirement of AAC1 (ASIL C). This decision variable is not involved in any other AAC and therefore it can immediately be allocated with ASIL C. Note that the allocation to $ASIL_{FM1}$ does not impact the rest of the problem. Assigning it with ASIL C will not contribute to fulfil any of the constraints $ASIL_{FM4}$ is part of, for example. Proceeding to $ASIL_{FM2}$, this variable is only involved in one AAC as well (AAC2). However, $ASIL_{FM3}$ also integrates AAC2 and therefore a decision on the allocation to $ASIL_{FM2}$ will have an impact on the allocation to $ASIL_{FM3}$ and vice-versa. If $ASIL_{FM2}$ is allocated with ASIL B, for example, $ASIL_{FM3}$ only needs to be assigned with ASIL A to meet the requirement of AAC2 (ASIL C). Hereinafter, variables presenting this type of dependencies are said to be *interconnected*. As with $ASIL_{FM2}$, $ASIL_{FM3}$ is not involved in AACs with other variables and therefore an optimal solution can be determined for both without considering any other allocations. Finally, $ASIL_{FM4}$,

$ASIL_{FM5}$ and $ASIL_{FM6}$ are interconnected. As it is highlighted in Figure 36, $ASIL_{FM4}$ is shared by AACs 3 and 4 and $ASIL_{FM5}$ integrates AACs 4 and 5.

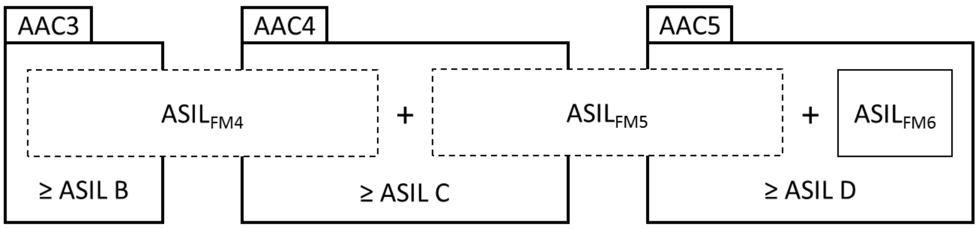


Figure 36 - AACs of S1. Decision variables that are shared by AACs have their limits drawn with dashes ($ASIL_{FM4}$ and $ASIL_{FM5}$).

AAC3 requires $ASIL_{FM4}$ to be at least ASIL B. If this allocation is adopted, to meet ASIL C in AAC4, the minimal allocation for $ASIL_{FM5}$ is immediate: ASIL A (ASIL B [$ASIL_{FM4}$] + ASIL A [$ASIL_{FM5}$] = ASIL C); finally, assuming the two previous decisions are taken, $ASIL_{FM6}$ would only need to be allocated with ASIL C to meet the requirement of AAC5 (ASIL A [$ASIL_{FM5}$] + ASIL C [$ASIL_{FM6}$] = ASIL D).

This last scenario shows that two variables, namely $ASIL_{FM4}$ and $ASIL_{FM6}$, do not need to be in the same AAC to be interconnected. It is important to also note that if $ASIL_{FM6}$ was involved in an AAC with a hypothetical $ASIL_{FM7}$, $ASIL_{FM7}$ would have an impact on the allocation to $ASIL_{FM6}$ and consequently on $ASIL_{FM4}$ and $ASIL_{FM5}$. Two generic variables, $ASIL_{FMX}$ and $ASIL_{FMY}$, are interconnected if they exist in the same AAC or if between them there is a path of variables joined by AACs. Figure 37 shows an example of such a path.

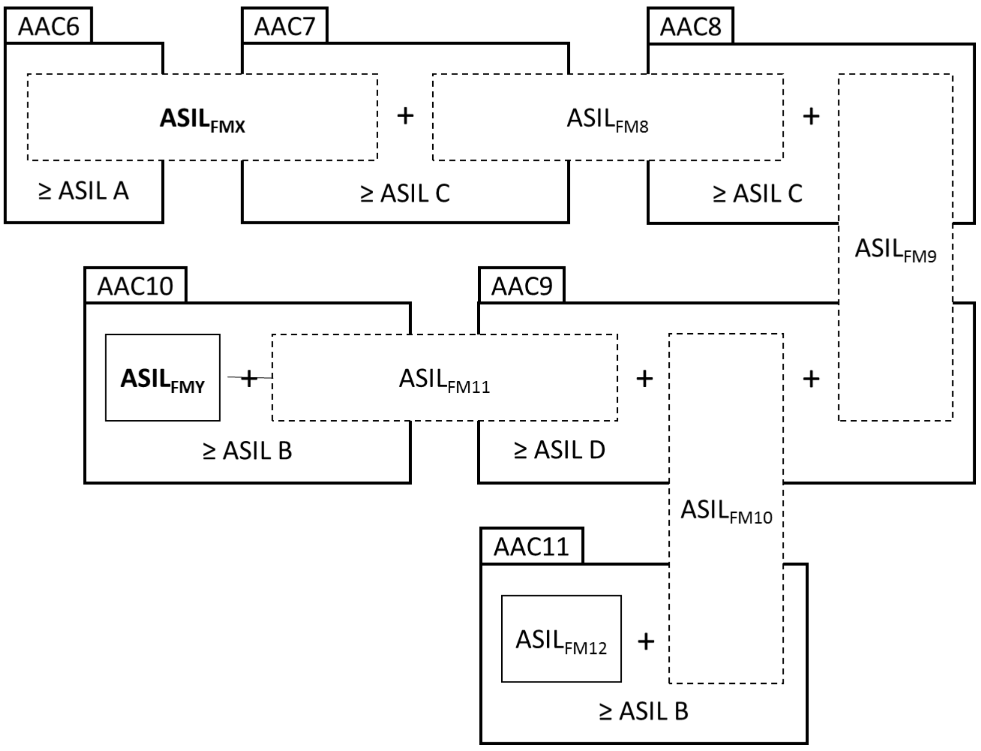


Figure 37 - Example of a path of variables across different AACs connecting two generic variables - $ASIL_{FMX}$ and $ASIL_{FMY}$.

Through AAC7, $ASIL_{FMX}$ impacts $ASIL_{FM8}$, which in turn has an influence on $ASIL_{FM9}$ through AAC8; $ASIL_{FM9}$ influences $ASIL_{FM11}$ via AAC9 and the latter finally impacts $ASIL_{FMY}$ in AAC10. It is clear that not only variables at the ends of the path are related; all the variables in the path are themselves part of the same interconnected set – in fact, a path of the same type can be drawn between any two of these variables. Furthermore, note that $ASIL_{FM10}$ is not necessary to form a path between $ASIL_{FMX}$ and $ASIL_{FMY}$; however, it impacts $ASIL_{FM9}$ and $ASIL_{FM11}$ through AAC9 and therefore it is interconnected with all the previous variables. Finally, a path can be delineated between $ASIL_{FM12}$ and any variable in this example through its interaction with $ASIL_{FM10}$ in AAC11. At the end of the analysis, all the variables in Figure 37 form a single interconnected set.

While dividing an ASIL allocation problem, every variable needs to be associated with one interconnected set. For simplicity, variables which are the only element in all of their AACs are here considered to be part of an interconnected *set*, although it is one constituted of a single element.

Returning to system S1, the original problem it poses can then be divided into three instances of smaller dimensionality that are completely independent. As is it shown in Figure 38, the different sub-problems do not share variables, nor AACs (if variables integrate the same AAC, they are interconnected by definition). Also, note that no AACs are added or removed when creating each sub-problem - the conjunction of the AACs for each sub-instance results in the set of AACs defined for the original problem.

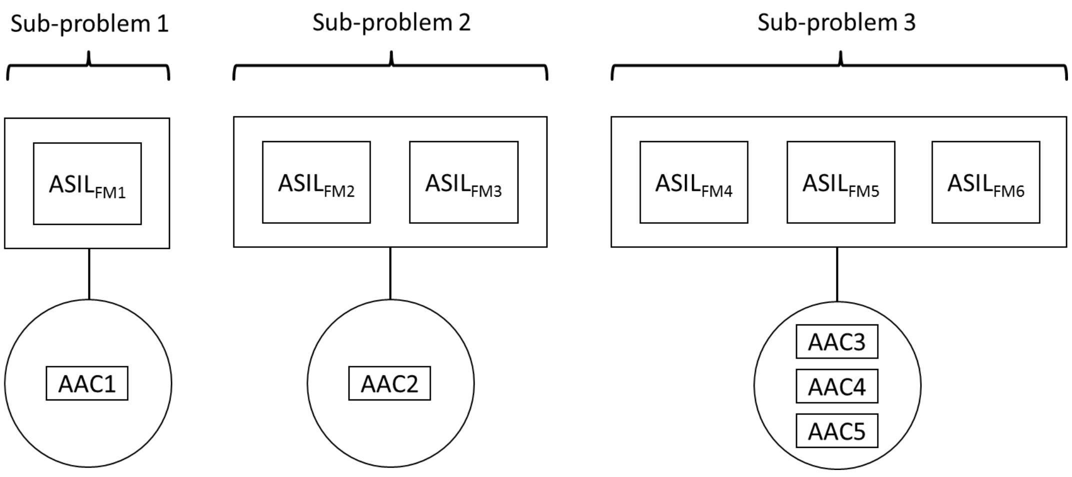


Figure 38 - Sub-problems devised for S1.

After the problem division step is performed, optimisation can be applied to each of the sub-problems separately. For interconnected sets composed of a single variable, such as the one of ASIL$_{FM1}$, no optimisation algorithm is utilised. Instead, a Direct Allocation Algorithm analyses the AACs associated with the variable, determines the highest ASIL requirement the variable needs to comply with and immediately allocates it with that value. At the end, the solutions yielded for each of the instances can be directly combined to provide an overall allocation for the original problem. The full approach is represented in Figure 39, using the example of problem S1.

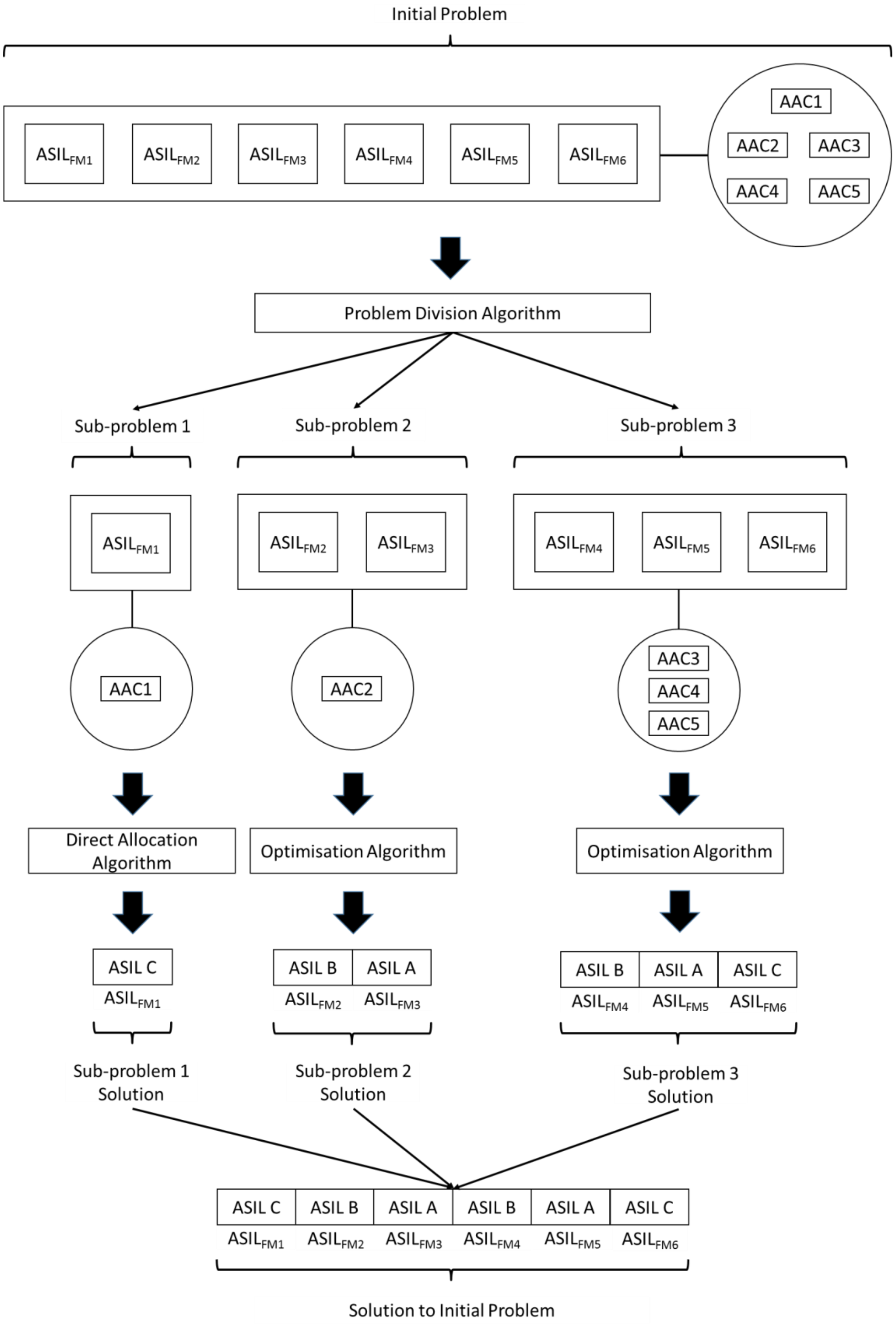


Figure 39 - Overview of the ASIL allocation divide and conquer optimisation approach.

Costs have been omitted from the description of the approach to facilitate understanding. Nevertheless, ASIL-dependent costs are calculated separately for each variable on the basis of the allocation they receive; as the final solution is just a conjunction of the different sub-solutions, the final cost is then given by the sum of the sub-solutions costs.

It is important to point out that the divide and conquer approach is not tied to a particular optimisation algorithm. Different users may utilise the optimisation technique of their choice.

One of the biggest challenges with ASIL allocation is bound to its combinatorial nature; the search space to explore grows exponentially with the number of variables in the problem. The reduced dimensionality of sub-instances produced by the divide and conquer process can be key in improving the optimisation performance. The problem posed by S1 includes six variables, meaning that there are 15625 solutions (5^6) to be explored if the problem is not divided. Alternatively, for each of the sub-problems the associated search space sizes are:

- Sub-problem 1: 5 solutions (5^1)
- Sub-problem 2: 25 solutions (5^2)
- Sub-problem 3: 125 solutions (5^3)

Algorithms tackling the sub-problems independently, will be exploring search spaces that are clearly smaller. There are a total of 155 solutions across the three sub-instances, a value which is more than 100 times smaller than the original search space size. The difference is potentially even more dramatic when larger problems are considered. For example, a problem with 60 variables has an associated search space of approximately 8.67×10^{41} solutions. If divided into three problems of dimensionality 20, the search space of each one of them is approximately 9.54×10^{13}; a combined search space of approximately 2.86×10^{14} solutions is obtained. The latter is 27 orders of magnitude smaller than the original search space size!

Another promising aspect to the divide and conquer approach is the possibility of applying parallelisation to speed up the optimisation process. Solving ASIL allocation sub-instances concurrently is an *embarrassingly parallel problem*. The sub-instances are completely independent and little effort is required to parallelise their execution; it would be embarrassing not to take advantage of the opportunity to enhance performance.

There are some important promising features to the divide and conquer approach, but it is also necessary to point out latent downsides. Firstly, determining the sets of interconnected variables consumes processing time. Furthermore, defining multiple sub-problems also brings the extra overhead of building data structures for AACs evaluation for each sub-instance. Finally, the approach necessarily adds to the size and complexity of the implementation.

5.3.2 Dividing the ASIL allocation benchmark

The previous section described how an original ASIL allocation problem can be divided into independently solvable sub-problems. However, no division can occur if all variables in the problem are interconnected. The question remains if generally systems present characteristics that allow dividing the original allocation problem they pose. This was investigated by revisiting the ASIL allocation benchmark introduced in chapter 4. Table 28 provides a first insight at the results.

Table 28 - Number of sub-problems obtained per case study.

Case study	Number of sub-problems
HBSM1	13
HBSM2	31
HBSM3	8
BSS	102

The original problems from each of the case studies can be divided into several independent sub-instances. This indicates that the divide and conquer optimisation approach presented in this thesis has more general applicability.

It is interesting that for three case studies - HBSM1, HBSM2 and BSS - the number of generated sub-problems is more than half the number of variables in the original problem. This tells that the dimensionality of the largest sub-instance in each of these case studies is necessarily smaller than half the one of the original problem. Table 29 details on the dimensionality of all the sub-problems devised across the benchmark.

Table 29 - Sub-problems for each case study in the ASIL allocation benchmark organised by dimensionality.

	Dimensionality of sub-problems						
Case study	*1*	*2*	*4*	*8*	*24*	*26*	*84*
HBSM1	10	1	1	1	-	-	-
HBSM2	28	1	1	-	-	1	-
HBSM3	4	1	1	-	1	1	-
BSS	101	-	-	-	-	-	1

As with the remaining case studies, HBSM3's largest sub-problem also contains less than half the number of variables of the original instance. In the case of HBSM1, Table 29 reveals that its largest sub-problem is even of dimensionality three times smaller than the one of the original problem. For BSS, more than half of its variables (101 out 185) can be directly allocated with the maximum ASIL requirement of the AACs they integrate.

There is only one sub-instance posing an optimisation problem in this case study and it includes 84 variables. This sub-instance is the one of highest dimensionality in the subdivided benchmark.

A closer look is now taken upon HBSM1 to illustrate how FMs of a system architecture interact to generate interconnected instances. In HBSM1 there are interconnected instances with only one variable, relating to FMs that do not share cut sets with any other FM. This is the case, for example, of IWM.VFailure1 which is only included in one cut set and is a single-point of failure for the hazard “braking with wrong value”. On the other extreme, there is the most complex interconnected instance of HBSM1, which includes eight variables (see Figure 40). A line between two FMs represents a cut set comprised of the two FMs.

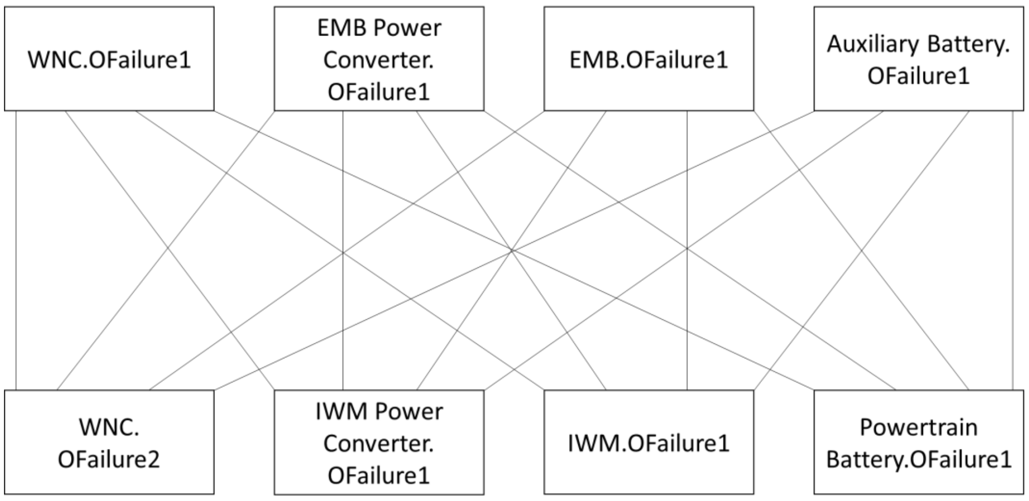

WNC.OFailure1
EMB Power Converter. OFailure1
EMB.OFailure1
Auxiliary Battery. OFailure1
WNC. OFailure2
IWM Power Converter. OFailure1
IWM.OFailure1
Powertrain Battery.OFailure1

Figure 40 – HBSM1 interconnected instance of size eight.

Two FMs that share a cut set are interconnected by definition. If WNC.OFailure1 is taken as the initial point in the analysis, it is immediate that it is interconnected with all the FMs from the bottom row. As a consequence, all five variables are interconnected.

In turn, each of the FMs in the bottom row is in a cut set with each FM in the top row. This means that all variables in Figure 40 are interconnected.

It is worth drawing a connection between the interconnected instance and the system architecture. The eight FMs in Figure 40 belong to the components of the power architecture of the HBSM1, including the WNC. FMs on the top row refer to FMs of components involved in EMB braking, whereas the ones in the bottom row refer to IWM braking. Each FM in a row is in a cut set with each FM in the other row. This is a consequence of the redundant braking architecture intended for the HBSM1, where the hazard “No braking” is only originated when there is braking omission in both the EMB and IWMs braking channels. For the “No braking” hazard to happen, an omission failure must occur in a component in each of the channels.

The interconnected instance identified in Figure 40 is a special case: setting an allocation for one of the variables does not just influence the allocation of the remaining variables - it immediately defines their minimum allocations. All cut sets represented in Figure 40 have to meet an ASIL D requirement, since they create the “No braking” hazard. Allocating, for example, WNC.OFailure1 with ASIL A means that each variable of the IWM channel needs to be at least ASIL C (WNC.OFailure1 is in a cut set with each FM of the IWM channel); in turn the FMs that are left to be allocated in the EMB channel need to be allocated with at least ASIL A (each FM in the IWM channel is in a cut set with a FM in the EMB channel). The constraints from the cut sets associated with this interconnected instance mean that to achieve a minimum solution, the FMs of a given braking channel are to be allocated with the same ASIL, and the total ASIL of the two channels needs to add up to ASIL D.

Analysis turns now to compare the original search space of each case study and the total search space of the sub-problems obtained after applying the divide and conquer procedure. Differences are immense (see Table 30).

Table 30 - Search space size of the original problems posed by each of case studies in the ASIL allocation benchmark and the total search space size across their sub-problems.

Case study	Original search space	Sum of sub-problems' search space
HBSM1	5.96046×10^{16}	391325
HBSM2	8.67362×10^{41}	1.49012×10^{18}
HBSM3	8.67362×10^{41}	1.54972×10^{18}
BSS	2.03916×10^{129}	5.16988×10^{58}

Of the different case studies, BSS receives the most dramatic change, with a resulting search space that is 71 orders of magnitude smaller than the initial. For HBSM1, the new search space is actually under one million solutions. HBSM2 and HBSM3 both receive reductions of 23 orders of magnitude.

AACs create relationships between variables; however, a higher number of AACs does not necessarily imply a larger set of interconnected variables. Multiple AACs can be composed of the same variables, for instance. Also, one AAC would be enough to link all variables in a problem. Table 31 (see next page) shows the dimensionality and number of AACs associated with each sub-instance in the subdivided ASIL allocation benchmark. The column furthest to the right shows what percentage of the number of AACs in an original problem belongs to one of its sub-instances (only sub-problems with sizes greater than 1 are shown).

Table 31 - Dimensionality and number of AACs of each sub-problem in the ASIL allocation benchmark (SP – Sub-Problem).

Sub-Problem	Dimensionality	Number of AACs	% of AACs of original problem
HBSM1-SP1	2	1	≈3.23
HBSM1-SP2	4	4	≈12.90
HBSM1-SP3	8	16	≈51.62
HBSM2-SP1	2	1	≈1.06
HBSM2-SP2	4	4	≈4.26
HBSM2-SP3	26	61	≈64.89
HBSM3-SP1	2	5	≈0.04
HBSM3-SP2	4	12	≈0.10
HBSM3-SP3	24	1296	≈11.20
HBSM3-SP4	26	10256	≈88.62
BSS-SP1	84	8096	≈98.52

In meeting the reflection that the size of a given interconnected instances does not necessarily grow with the number of AACs associated with it, consider HBSM2-SP3 and HBSM3-SP4; both have the same dimensionality, but the first includes a much smaller number of AACs (61 versus 10256). That being noted, within each of the case studies, sub-problems with a greater number of AACs are of higher dimensionality.

It is interesting that although BSS-SP1 is less than half the size of the problem it originates from, it keeps more than 98.52% of the original AACs. Also, in the case of HBSM3, there are two bigger sub-problems of similar dimensionalities, but HBSM3-SP4 concentrates 88.62% of the AACs in the original problem. The latter integrates the highest number of AACs across the entire subdivided benchmark. Also, most of its AACs are of sizes six and eight (see Table 32); for all other sub-problems, AACs contain less than five variables.

Table 32 - Number of AACs for each sub-problem defined across the ASIL allocation benchmark, organised by size.

	Size of AACs						
Sub-problems	1	2	3	4	5	6	8
HBSM1-SP1	-	1	-	-	-	-	-
HBSM1-SP2	1	3	-	-	-	-	-
HBSM1-SP3	-	16	-	-	-	-	-
HBSM2-SP1	-	1	-	-	-	-	-
HBSM2-SP2	1	3	-	-	-	-	-
HBSM2-SP3	-	61	-	-	-	-	-
HBSM3-SP1	-	5	-	-	-	-	-
HBSM3-SP2	5	7	-	-	-	-	-
HBSM3-SP3	-	-	-	1296	-	-	
HBSM3-SP4	-	5	72	540	162	2916	6561
BSS-SP1	32	720	5040	2304	-	-	

BSS-SP1 is by far, the sub-problem with the highest number of variables; it also includes the second highest number of AACs. HBSM3-SP4 integrates the biggest number of AACs, which are, for most part, larger than in any other sub-problem; it is also the sub-problem with the second highest dimensionality. Dimensionality defines the size of the search space to explore; feasibility evaluation, on the other hand, impacts the processing effort per iteration. The characteristics of these two sub-problems suggest that they could be the two requiring more processing effort to solve.

The cost of the optimal solutions for the different case studies, as a function of the ASIL cost heuristics defined in chapter 4, have been previously introduced (see section 4.4). The optimal cost of a given original problem can help in determining the costs of the optimal allocations for its respective sub-problems. Take the example of an illustrative ASIL allocation problem A that can be divided into three sub-problems (see Figure 41).

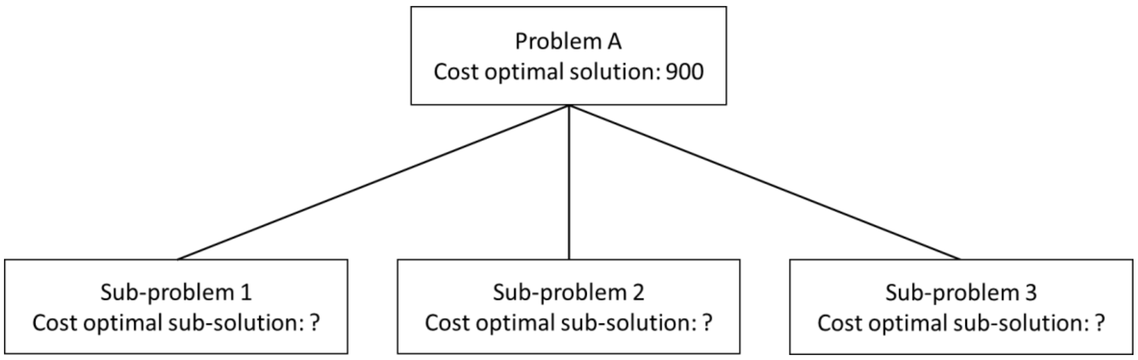

Problem A
Cost optimal solution: 900
Sub-problem 1
Cost optimal sub-solution: ?
Sub-problem 2
Cost optimal sub-solution: ?
Sub-problem 3
Cost optimal sub-solution: ?

Figure 41 - Illustrative ASIL allocation problem (problem A), which can be decomposed in to three sub-problems.

As previously established, the cost of a solution to an original problem can be calculated from the sum of the costs of the solutions from its sub-problems. The optimal cost for problem A is known to be 900 units. If solutions can be found for sub-problems 1, 2 and 3, whose total cost amounts to 900, they are known to be optimal themselves.

Tabu Search was run across the subdivided benchmark and the costs of the best solutions it found are given in Table 33. These costs are known to be optimal: the total cost of the sub-problems of each case study, together with the costs of the allocations to interconnected sets of size 1, amount to the optimal cost of their original problem.

Table 33 - Optimal costs for the sub-problems of the ASIL allocation benchmark as a function of the different ASIL cost heuristics.

	ASIL Cost Heuristic			
Sub-Problem	*Linear*	*Experiential-I*	*Experiential-II*	*Logarithmic*
HBSM1-SP1	40	40	40	200
HBSM1-SP2	80	90	90	10200
HBSM1-SP3	160	160	160	800
HBSM2-SP1	40	40	40	200
HBSM2-SP2	80	90	90	10200
HBSM2-SP3	520	520	520	2600
HBSM3-SP1	40	40	40	200
HBSM3-SP2	80	100	95	10300
HBSM3-SP3	240	240	120	240
HBSM3-SP4	250	250	165	410
BSS-SP1	1560	1920	1695	138720

5.3.3 TS or TS Divide and Conquer?

After demonstrating the general applicability of the divide and conquer technique to ASIL allocation problems, it is now time to evaluate if it brings additional benefits to the optimisation performance. In section 5.2, a comparison has been conducted between three optimisation techniques in solving the ASIL allocation benchmark from chapter 4. They were metaheuristic optimisation methods based on concepts of Genetic Algorithms, Particle Swarm and Tabu Search. The Tabu Search technique has shown the most promising results, overcoming the two other approaches in every problem – in most cases being at least one order of magnitude faster. Furthermore, it offered the least complex tuning experience. Having shown such an important superiority, Tabu Search was integrated with the divide and conquer approach and research has been conducted to evaluate if the method brings further optimisation performance benefits.

Hereinafter, the optimisation process combining the divide and conquer approach with the Tabu Search algorithm will be referred to as Tabu Search Divide and Conquer, or TSDC. On the other hand, the optimisation methodology where an initial problem is solved altogether using Tabu Search is simply named after the optimisation algorithm itself – Tabu Search, or TS.

To compare the TS and TSDC methods, an initial experiment has been designed in which the two techniques are run using the same, large set of parameterisations. In the process of exhaustively trying each of these parameterisations, information of the best performance obtained by the two algorithms is kept. In the end, these best performances are compared to evaluate if there are significant differences. As in the study between GA, PSO and TS, for each parameterisation the two techniques are run

50 times for the problems in the ASIL allocation benchmark. From now onwards in this document, every time a reference is made to an algorithm being run across the benchmark, it should always be assumed that each problem is solved 50 times. In some runs, it can take a very small amount of time for the techniques to solve a problem. In order to minimise measurement errors introduced by timers, instead of measuring the time an algorithm takes to solve a given problem in each of the 50 runs, the total time to execute the 50 runs is recorded. Finally, algorithms were run on the same machine utilised in the experiments of the previous section, which is equipped with an Intel i5-3570K processor clocked at 3.40GHz and with 8GB of RAM.

Before introducing the set of parameterisations utilised in this experiment, perhaps it is worth briefly revisiting the functions of the parameters to tune in the Tabu Search algorithm. In its optimisation process, Tabu Search forbids certain variables from being decremented or incremented. Because a decrement to a variable's allocated ASIL means reducing the cost of a solution, it is called a *descent move*. Increments to allocations are, on the other hand, associated with an increase to the solution's cost and are termed *ascent moves*. Selected descent moves are forbidden by the algorithm for a duration of p iterations; ascent moves are forbidden for q iterations. p and q are dynamic and their values are incremented throughout the search process with every $updatePeriod_p$ and $updatePeriod_q$ iterations, respectively. In the beginning of the search both p *and* q are set to zero and are incremented until reaching their limits - $limit_p$ and $limit_q$; when the limits are reached, they are reset to zero. To summarise, $updatePeriod_p$, $updatePeriod_q$, $limit_p$ and $limit_q$ are the user defined parameters of the algorithm.

Starting with *updatePeriod*$_p$ and *updatePeriod*$_q$, one of the parameterisations considered in this experiment is the one provided by Hansen and Lih (1996):

- (*updatePeriod*$_p$; *updatePeriod*$_q$) = (3; 4)

In the current study, similarly to the values proposed by Hansen and Lih, *updatePeriod*$_p$ is always set to a value one unit smaller than *updatePeriod*$_q$. As discussed when tuning TS for comparison with GA and PSO, experimentation has shown good results with such an approach.

The values found to perform well in section 5.2.1 are also used in this experiment:

- (*updatePeriod*$_p$; *updatePeriod*$_q$) = (10 ; 11)

Another parameterisation utilised relates to the minimum update period:

- (*updatePeriod*$_p$; *updatePeriod*$_q$) = (1 ; 2)

In the three parameterisations presented to this point, the values for *updatePeriod*$_p$ follow an approximate multiplier of three (1; 3; 10). Two more parameterisations have been defined for larger update periods in which *updatePeriod*$_p$ continues to grow in the same fashion:

- (*updatePeriod*$_p$; *updatePeriod*$_q$) = (30 ; 31)
- (*updatePeriod*$_p$; *updatePeriod*$_q$) = (100 ; 101)

In regards to *limit*$_p$ and *limit*$_q$, Hansen and Lih (1996) propose setting them to 0.4n; *n* stands for the number of variables in a problem. In section 5.2.1, setting both limits to 2n provided good results. However, other trials have also shown promising performances when the two limits were not set to the same value. In this way, in the current experiment, the two parameters are allowed to be set differently. Furthermore, they are both permitted to vary from 0n to 10n (with intervals of 0.5n). The upper bound for the two parameters is five times greater than the values used in section 5.2.1.

The total number of parameterisations is given by the number of different pairs of values for $limit_p$ and $limit_q$ (441) multiplied by the number of parameterisations for $updatePeriod_p$ and $updatePeriod_q$ (5). In total, TS and TSDC are run with 2205 different parameterisations.

The best run of each of the optimisation approaches is the one where they solve all the problems in the smallest amount of time. No fixed time limit is defined to apply across the different algorithm runs. Instead, as the best time is improved, it becomes the new stopping criterion. This experiment was designed to evaluate if there are significant differences between the best performances of TS and TSDC; runs which do not finish before the best processing time previously found are stopped and no information about them is recorded. The first run of each of the optimisation approaches made use of a parameterisation that was known to give reasonable performances; this contributed to minimise the total time consumed by the experiment.

The parameterisations and processing times of the best runs obtained by TS and TSDC are presented in Table 34. The times shown refer to the cumulative total of solving every problem in the benchmark 50 times.

Table 34 - Time and parameters for best runs of TS and TSDC

Optimisation approach	**Best Parameterisation**				**Total Time of Best Run (seconds)**
	$limit_p$	$limit_q$	$updatePeriod_p$	$updatePeriod_q$	
TS	2.5n	1.5n	10	11	14.860153
TSDC	3.5n	1n	10	11	5.054777

TSDC shows a significant improvement over TS, presenting a best run which is nearly 3 times faster. The time shown in Table 34 for TSDC can be divided in two. 3.453074 seconds are spent by the optimisation algorithm to solve all the sub-problems containing more than one variable. The remaining 1.601703 seconds are consumed by deterministic tasks: generating the different optimisation sub-problems and associated data structures, applying the direct allocation algorithm to the interconnected instances of size 1 and reconstructing solutions to original problems from solutions of their sub-instances. If only the optimisation time is considered, TSDC is actually more than 4 times faster than TS. Also, it must be said that the time consumed to solve instances of size one and in solution reconstruction is marginal (less than 3 milliseconds). The largest overhead of the TSDC approach concerns problem division.

On the basis of the best runs of the two approaches, results show that TSDC is substantially faster. However, this experiment only demonstrates that TSDC is faster than the best TS for a single parameterisation. How does TSDC perform when tuned differently?

A second experiment was designed to evaluate the sensitivity of TSDC's performance to parameterisation selection. TSDC is again applied to the ASIL allocation benchmark, using the same set of parameterisations defined for the previous experiment. However, now, the stopping criterion for a run is not updated throughout the process of exploration of the different parameterisations. Instead, the time obtained by the best run of TS (14.860153 seconds) is defined as a fixed stopping criterion. Using this time limit, the parameterisations with which TSDC overcomes the best TS can be determined.

Table 35 shows the number of parameterisations enabling TSDC to finish the benchmark under the time limit and also under half that same time limit.

Table 35 - Number of tests solved by TSDC under the best time obtained by TS (left) and half the best time obtained by TS (right).

Tests solved under 14.860153 seconds	Tests solved under 7.4300765 seconds
424	73

TSDC does not just perform better than the best TS for an isolated parameterisation; the technique was able to solve all problems under the time of TS's best run using a total of 424 parameterisations. 424 parameterisations correspond to approximately one fifth of all the parameterisations defined. Furthermore, for 73 parameterisations, TSDC is actually more than two times faster than the best TS. In meeting the considerations that led to this experiment, TSDC performs well, and is consistently better than the best TS across a vast range of parameterisations.

5.3.4 Tuning TSDC for efficient conquering

TSDC wins in the comparison with TS. This section investigates which parameterisations enable TSDC's most efficient performances, in order to inform parameter selection for future uses of the approach. Initially, the parameterisations with which TSDC obtained improvements over the best TS are identified. Figure 42 plots how these parameterisations are distributed as a function of $limit_p$ and $limit_q$. Across this chapter, multiple charts will be presented concerning $limit_p$ and $limit_q$ parameterisations. The values shown in the charts for the two limits are given as a function of problem dimensionality, acting as multipliers. For example, one of the parameterisations shown in Figure 42 has $limit_p$ and $limit_q$ set to 2.5; this refers to tests where TSDC was run with $limit_p$ and $limit_q$ set to 2.5 times the dimensionality of each problem it solved.

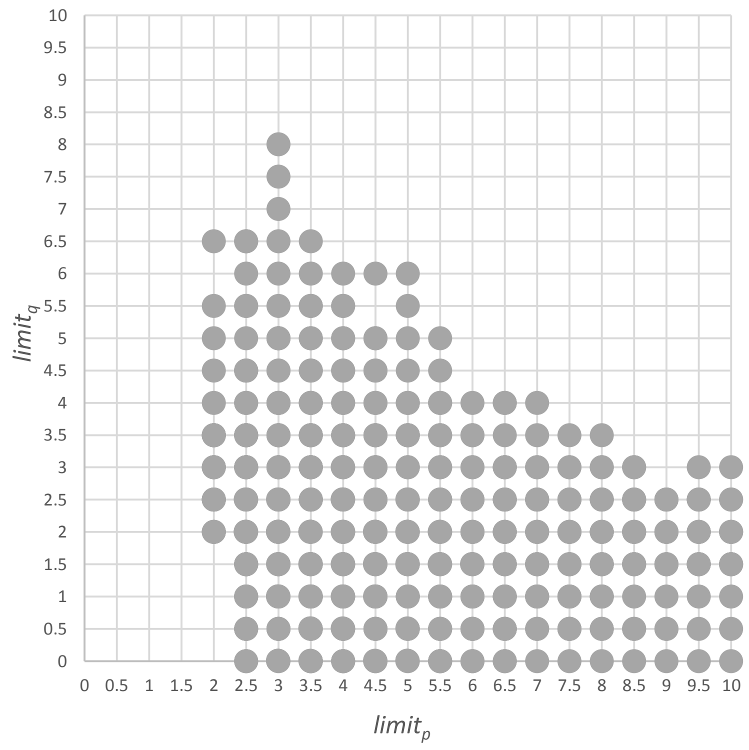


Figure 42 - Combinations of $limit_p$ and $limit_q$ included in the 424 best parameterisations of TSDC.

Figure 42 shows that the parameterisations are distributed along a single, predominantly contiguous region. It seems important not setting $limit_p$ to values smaller than 2n; curiously, the value suggested by Hansen and Lih (1996) is 0.4n. These authors tackle a redundancy allocation problem, which is inherently different than ASIL allocation; possibly, this is a contributing factor for the difference found between the value they suggest and the ones obtained in this experiment. Continuing the analysis of Figure 42, parameterisations where $limit_q$ is set to very high values do not seem to yield promising results. 3n is the value for $limit_p$ showing across more $limit_q$ parameterisations; as for $limit_q$, 2n and 2.5n are the parameterisations appearing for more values of $limit_q$.

There are 424 parameterisations for which TSDC finished the benchmark before the best time of TS. However, Figure 42 only shows 173 different pairs of values for *limit*$_p$ and *limit*$_q$. In fact, as it can be observed in Figures 43, 44, 45 and 46, the same values for these two parameters are often found across different parameterisations for *updatePeriod*$_p$ and *updatePeriod*$_q$. Five different pairs of values were defined for (*updatePeriod*$_p$; *updatePeriod*$_q$), but there are only four figures showing the combinations of *limit*$_p$ and *limit*$_q$ for a particular parameterisation of (*updatePeriod*$_p$; *updatePeriod*$_q$). This is because TSDC was not able to complete the benchmark under TS's best time with any parameterisation that has (*updatePeriod*$_p$; *updatePeriod*$_q$) set to (1; 2).

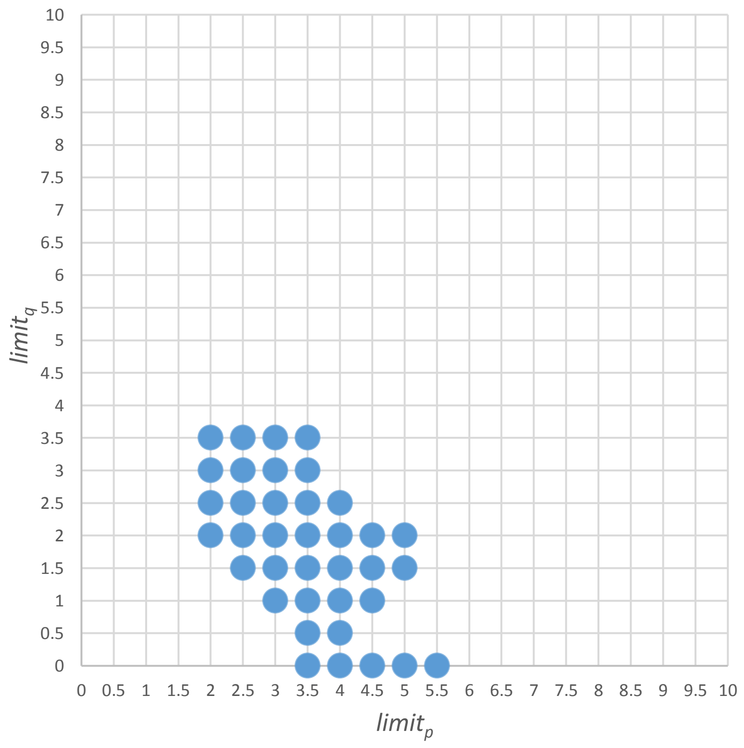


Figure 43 - Parameterisations of *limit*$_p$ and *limit*$_q$ for which TSDC, using (*updatePeriod*$_p$; *updatePeriod*$_q$) set to (3; 4), overcame TS's best performance.

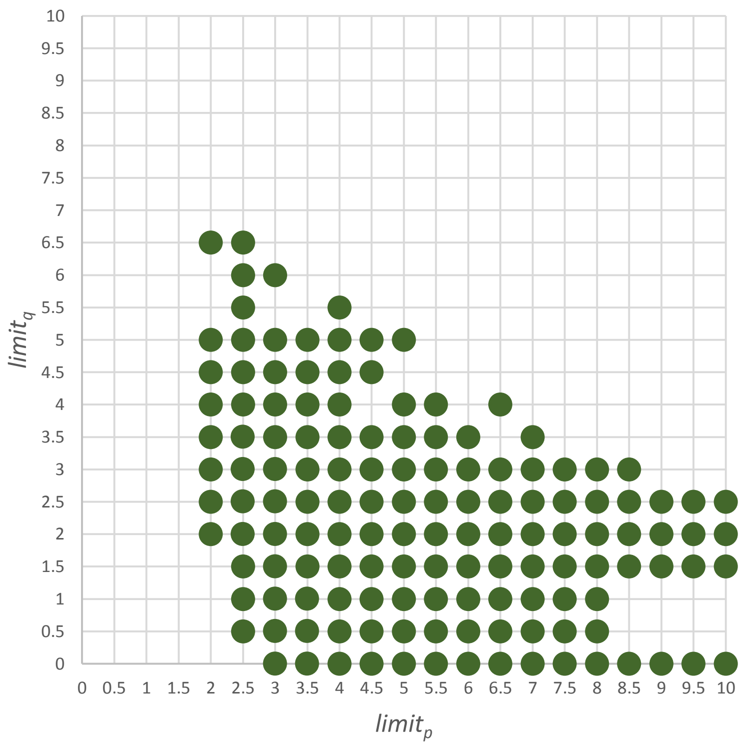


Figure 44 - Parameterisations of *$limit_p$* and *$limit_q$* for which TSDC, using (*$updatePeriod_p$*; *$updatePeriod_q$*) set to (10; 11), overcame TS's best performance.

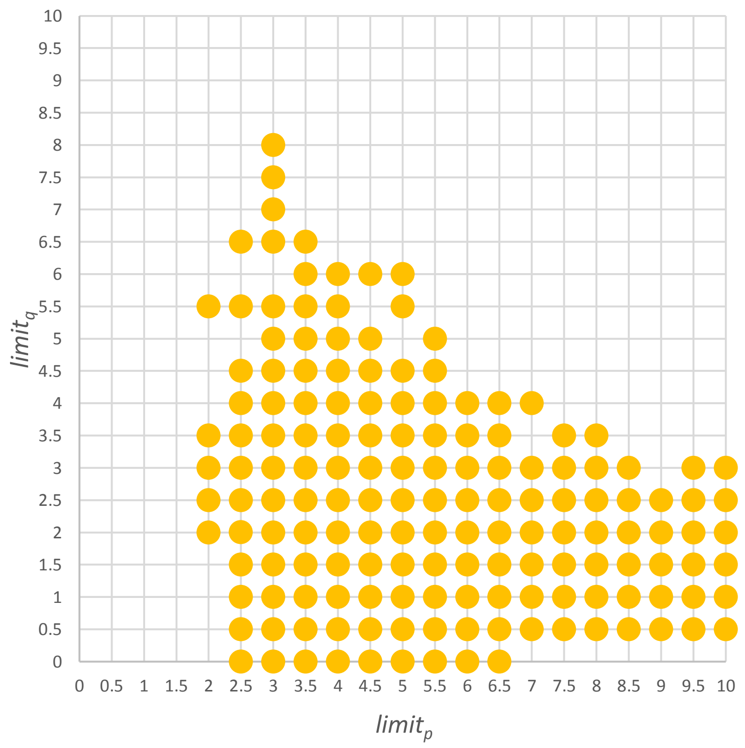


Figure 45 - Parameterisations of *$limit_p$* and *$limit_q$* for which TSDC, using (*$updatePeriod_p$*; *$updatePeriod_q$*) set to (30; 31), overcame TS's best performance.

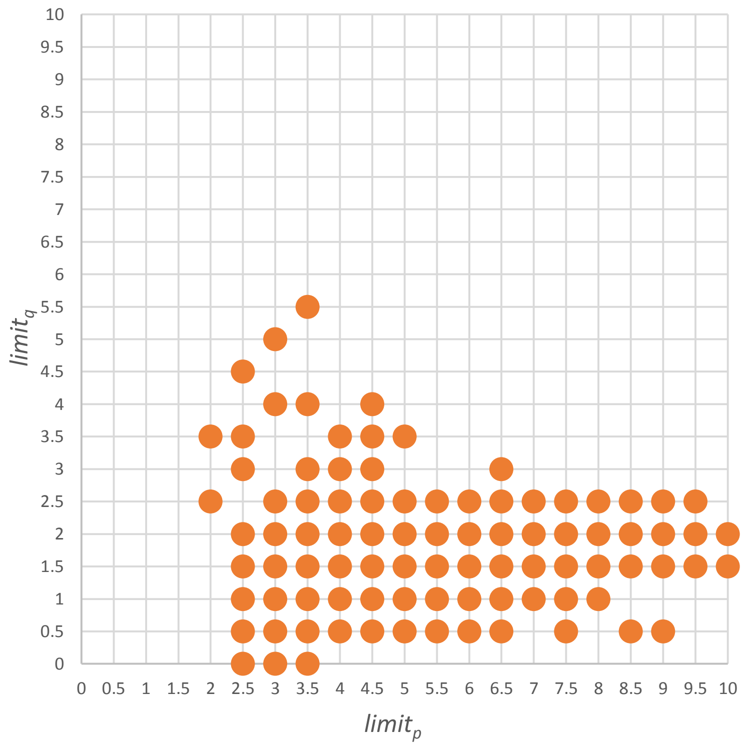


Figure 46 - Parameterisations of *limit$_p$* and *limit$_q$* for which TSDC, using (*updatePeriod$_p$*; *updatePeriod$_q$*) set to (100; 101), overcame TS's best performance.

In the 424 best parameterisations of TSDC, the values most often found for (*updatePeriod$_p$*; *updatePeriod$_q$*) are (30; 31), followed by (10; 11). After the pair (1; 2), which does not appear amongst the best parameterisations, (3; 4) is the least represented. The exact distribution of parameterisations, as a function of (*updatePeriod$_p$*; *updatePeriod$_q$*), is shown in Figure 47.

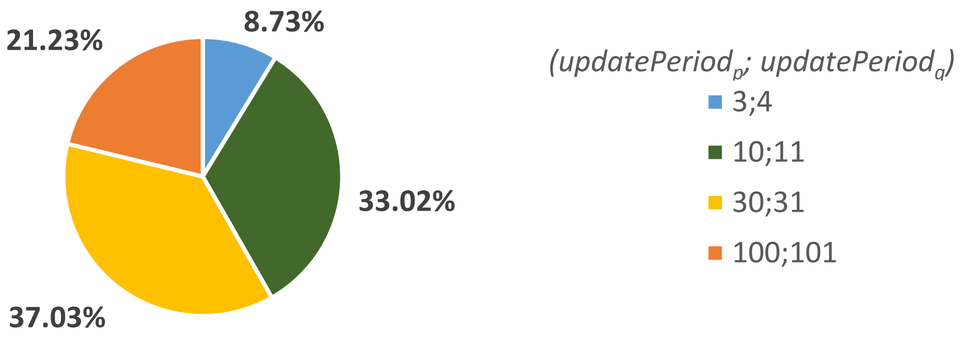


Figure 47 - Distribution of TSDC's best 424 parameterisations as a function of (*updatePeriod$_p$*; *updatePeriod$_q$*).

The previous analysis identifies the most represented pairs of values for (*updatePeriod*$_p$; *updatePeriod*$_q$) amongst the 424 best parameterisations; however, it does not tell if those give the most promising performances. Figure 48 can help understanding this; for each pair of values defined for (*updatePeriod*$_p$; *updatePeriod*$_q$), it shows the number of associated parameterisations that have solved the benchmark after a given processing time.

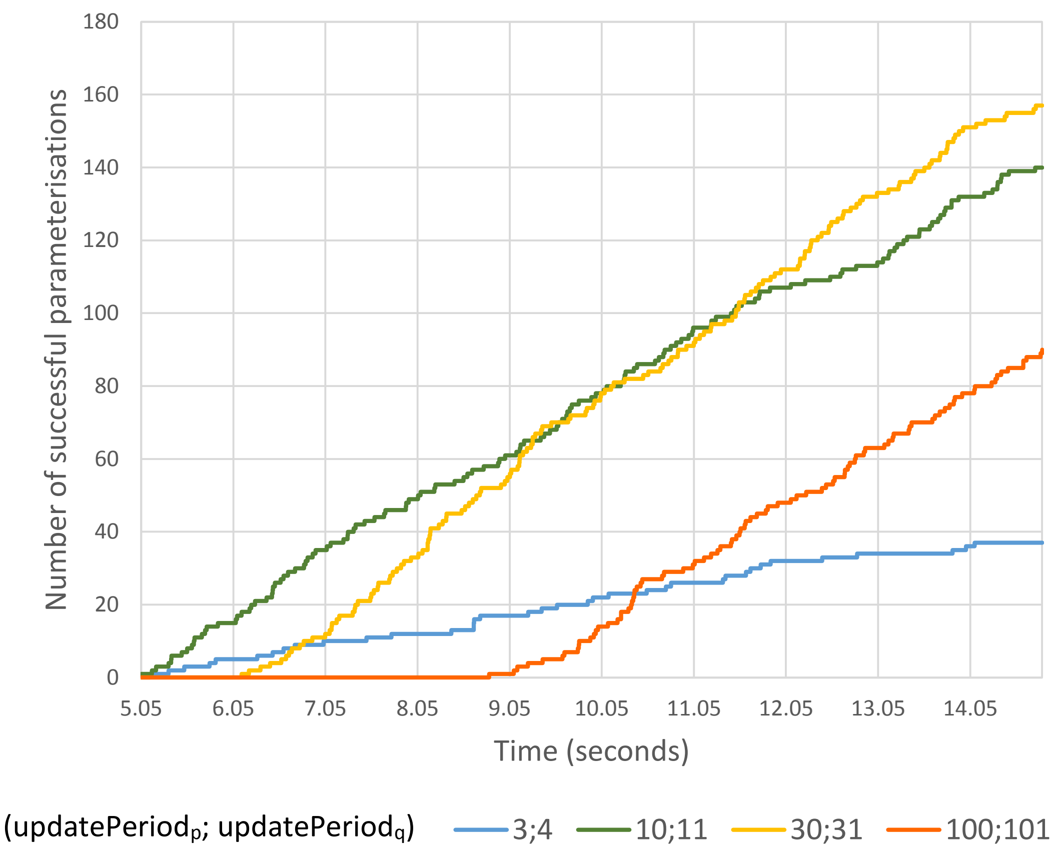


Figure 48 - Number of successful parameterisations as a function of (*updatePeriod*$_p$; *updatePeriod*$_q$) after a given processing time. The maximum time shown corresponds to the moment where TS's best run finished.

The fastest parameterisation has (*updatePeriod*$_p$; *updatePeriod*$_q$) set to (10; 11). As previously identified, (30; 31) is the most represented pair of values for (*updatePeriod*$_p$; *updatePeriod*$_q$); however, when its fastest parameterisation finished, (10; 11) already had 18 successful parameterisations and (3; 4) had five. It is interesting to observe that (3; 4) presents some of the fastest results, but its number of successful parameterisations grows slowly. Also, (30; 31) and (100; 101) initially spend approximately 1.1 and 3.8 seconds, respectively, without solving any test; after these intervals their number of successful parameterisations grows steadily. It seems like these larger update periods for *p* and *q* introduce a delay in the search process. This could be the case, for example, when the search is in a state where it finished investigating a given solution area and it needs to switch to a global exploration mode; if *p* and *q* are set to relatively small values, given the higher update periods, it will take more iterations to reach the high values for *p* and *q* that would force the search to evolve somewhere else.

Thus far, the set of 424 best parameterisations has been studied. The pairs of values for $limit_p$ and $limit_q$ found amongst these best parameterisations has been identified in Figure 42; it corresponds to approximately 40% of all the pairs of values defined for $limit_p$ and $limit_q$. Furthermore, (30; 31) and (10; 11) are the most robust pairs of values for ($updatePeriod_p$; $updatePeriod_q$). Figure 45 and Figure 44 show for (30; 31) and (10; 11), respectively, the areas of correspondent promising combinations of $limit_p$ and $limit_q$; each of them covers approximately one third of the entire set of pairs of values defined for the limits. It has been shown in Figure 48 that the two highest pairs of update periods, (30; 31) and (100; 101), present a delay in starting to produce successful runs and that update periods (10; 11) and (3; 4) are linked to the 23 fastest parameterisations. It can also be seen in Figure 48, that there are parameterisations with processing times not very far from the one of the best TS. The optimisation algorithm makes use of randomness in its search process and it is possible that, if re-run, some of the least promising parameterisations could yield worse performances than TS's best. Hence, it seems reasonable to focus on parameterisations that present significant improvements over the best TS. Furthermore, the parameterisations giving significant improvements are arguably the ones justifying the use of TSDC in the first place. In this way, an arbitrary limit is defined, and the parameterisations with which TSDC is at least two times faster than the best TS will be analysed (half of TS's best time corresponds to 7.4300765 seconds). In the experiment of section 5.3.3, there were 73 parameterisations enabling TSDC to finish the benchmark under this time limit. A new experiment was conducted, and TSDC was re-run across the entire benchmark 50 times using all the parameterisations previously defined to compare the two algorithms. This is to analyse the impacts of randomness over the parameterisations considered to be most promising; simultaneously, it allows identifying with more confidence the parameterisations giving faster performances.

From the new experiment, it was possible to observe that out of the initial 73 parameterisations with which TSDC completed the benchmark under 7.4300765 seconds, 66 presented average times (over the 50 runs) under the same time limit. Furthermore, two extra parameterisations have been identified. Figure 49 shows the combinations of $limit_p$ and $limit_q$ included in this group of 68 parameterisations. The pairs of values found for the limits are very similar to the ones obtained before re-running the experiments 50 times - there are 41 common pairs of values out of an initial group of 45. The four combinations of $limit_p$ and $limit_q$ that have been removed are: ($limit_p$; $limit_q$) = [(3.5; 3), (4; 3), (5.5; 2.5), (6; 1)].

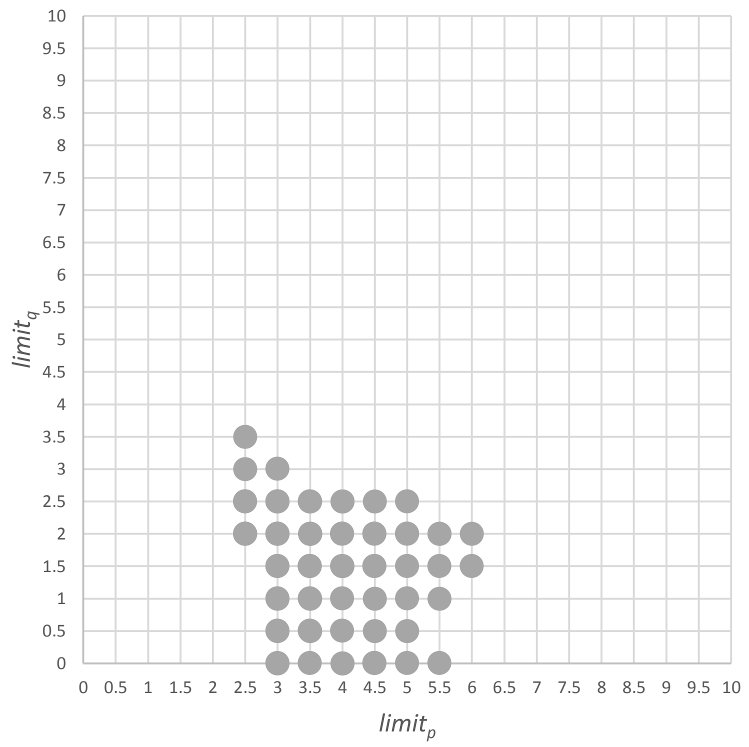


Figure 49 - Combinations of *$limit_p$* and *$limit_q$* included in the 68 best parameterisations of TSDC (data is based on average times from 50 runs).

The 41 pairs of values for $limit_p$ and $limit_q$ shown in Figure 49 are a subset of the ones found amongst the 424 overall best parameterisations of TSDC (compare with Figure 42). It is interesting that these values for $limit_p$ and $limit_q$ are still concentrated along a single region. There was a significant reduction in the maximum bounds of $limit_p$ and $limit_q$: the first decreased from 10n to 6n; the second was previously 8n and it is now 3.5n. The minimum value of $limit_p$ went from 2n to 2.5n. Furthermore, there are still parameterisations for which forbidding moves only in the descent direction provides enough diversity to the search process – six parameterisations exist with $limit_q$ set to zero.

Figure 50 shows $limit_p$ and $limit_q$ parameterisations as a function of the update periods for *p* and *q*.

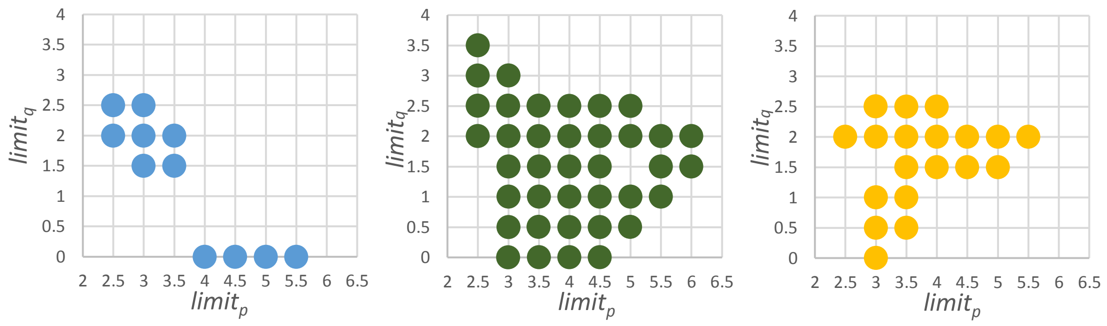


Figure 50 - Parameterisations of *limit$_p$* and *limit$_q$* as a function of (*updatePeriod$_p$*; *updatePeriod$_q$*), for which TSDC is at least two times faster than the best TS (data is based on average times from 50 runs). Blue: (*updatePeriod$_p$*; *updatePeriod$_q$*) = (*3*; *4*); green: (*updatePeriod$_p$*; *updatePeriod$_q$*) = (10; 11); yellow: (*updatePeriod$_p$*; *updatePeriod$_q$*) = (30; 31).

It seems important not choosing update periods that are very small or very high: the smallest pair of update periods, (1; 2), was not part of the 424 best parameterisations, and it is not part of any of the 68 best; furthermore, the highest update periods, (100; 101), were present in approximately 21% of the best 424 parameterisations but are not amongst any of the best 68. Parameterisations with (*updatePeriod$_p$*; *updatePeriod$_q$*) set to (10; 11) have the highest number of combinations of *limit$_p$* and *limit$_q$*. Out of the set of 41 combinations of *limit$_p$* and *limit$_q$* shown in Figure 49 only three of them do not give performances two times faster than the best TS when (*updatePeriod$_p$*; *updatePeriod$_q$*) is set to (10; 11).

The combinations of *limit$_p$* and *limit$_q$* presented in Figure 50 are based on average results after running TSDC 50 times; again, the results shown are very similar to the ones obtained when the technique was only run once. In both scenarios, there are no parameterisations with (*updatePeriod$_p$*; *updatePeriod$_q$*) set to (1; 2) and (100; 101). For update periods (3; 4), the results based on 50 runs present one parameterisation more, with *limit$_p$* and *limit$_q$* set to 3n and 1.5n. Regarding update periods (10; 11), there are four less combinations: (*limit$_p$*; *limit$_q$*) = [(3.5; 3), (4; 3), (5; 0), (6; 1)]. Finally, concerning update periods (30; 31), there are three less parameterisations: (*$limit_p$*; *$limit_q$*) = [(4.5; 1), (5.5; 2.5), (6; 1.5)]; one extra has been obtained as well: (*$limit_p$*; *$limit_q$*) = (3.5; 2.5). It is interesting that the parameterisations removed and added all belong to outlying positions of the promising areas in Figure 50.

The exact distribution of the 68 best parameterisations, as a function of update periods for *p* and *q*, can be seen in Figure 51.

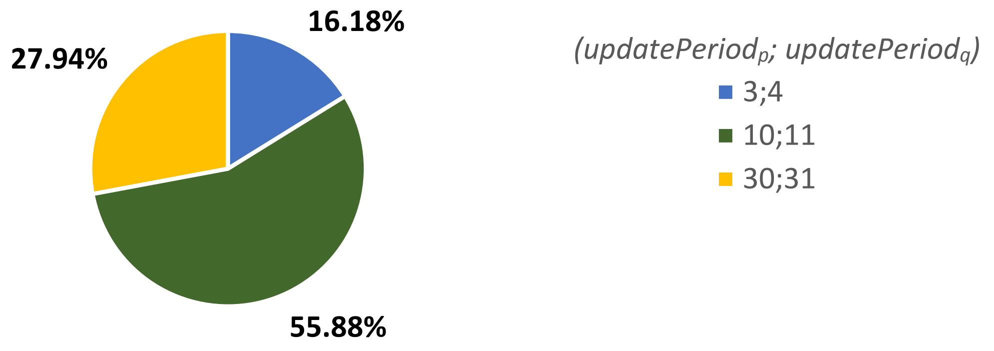


Figure 51 - Distribution of TSDC's parameterisations, which give performances more than two faster than the best TS, as a function of (*$updatePeriod_p$*; *$updatePeriod_q$*) (data is based on average times from 50 runs).

(30; 31) had been identified as the most represented pair of values for (*$updatePeriod_p$*; *$updatePeriod_q$*) amongst the best 424 parameterisations; now, the pair (10; 11) doubles the number of parameterisations that include (30; 31). The quota for update periods (3; 4) grew from 8.73% to 16.18%.

There are no significant differences when the data from Figure 51 is compared to the results obtained when TSDC was run only once with each parameterisation. The biggest change concerns update periods (3; 4) and even in this case, the variation corresponds to less than 3%: the percentage of parameterisations associated with (3; 4) was 13.70% when the algorithm was run once and it went to 16.18% when TSDC is run 50 times. In turn, the parameterisations quota associated with (10; 11) was reduced from 57.53% to 55.88%. Finally, the percentage of parameterisations associated with (30; 31) went from 28.77% to 27.94%.

Figure 52 shows the number of successful parameterisations associated with the three pairs of values for (*updatePeriod*$_p$; *updatePeriod*$_q$) as a function of execution time. If one were to focus the analysis to smaller and smaller time limits, there would always be more successful parameterisations associated with (10; 11) than with any other pair of update periods.

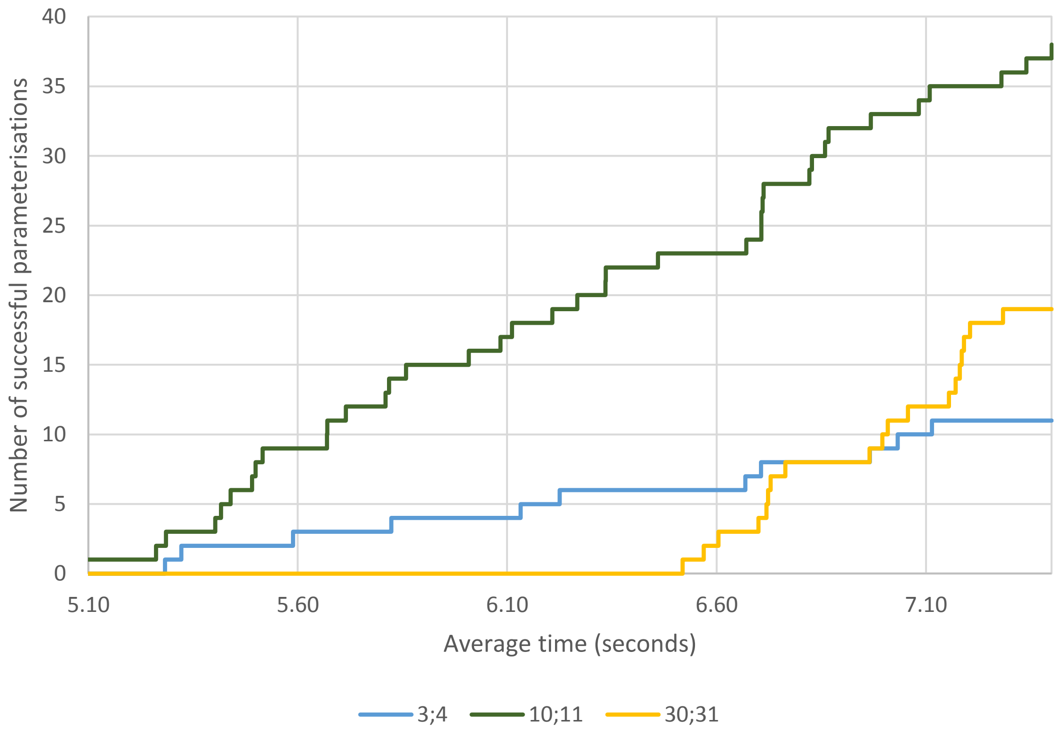


Figure 52 - Number of successful parameterisations as a function of (*updatePeriod*$_p$; *updatePeriod*$_q$) after a given processing time (data is based on average times from 50 runs). The maximum time shown corresponds to half the time taken by the best TS to finish the benchmark.

When analysing the 424 best parameterisations, a delay had been identified for parameterisations with (*updatePeriod*$_{p;}$ *updatePeriod*$_q$) set to (30; 31) to start solving the benchmark. However, for times that are half the ones of TS's best, the number of successful parameterisations with (30; 31) still overcomes the one from (3; 4). In this way, when choosing to set (*updatePeriod*$_p$; *updatePeriod*$_q$) to (3; 4) one would have the possibility of finding a better parameterisation than when selecting (30; 31); on the other hand, adopting the latter gives a higher number of parameterisations that improve TS by a factor of two. Previously, in Figure 48, a same type of analysis to the one of Figure 52 had been introduced. The data shown in Figure 48, however, had TS's best time as a maximum time limit. Furthermore, the data had been obtained from running TSDC only once across the benchmark. If Figure 48 is only analysed up to a time limit corresponding to half TS's best time, the resulting information is very similar to the one of Figure 52. Once again, this indicates that the results produced when running TSDC once or 50 times are not very different.

The 68 best parameterisations have now been examined. The combinations of *limit*$_p$ and *limit*$_q$ included in these parameterisations have been identified in Figure 49; setting (*updatePeriod*$_p$; *updatePeriod*$_q$) to (10; 11) gives good results with their vast majority (see Figure 50). In fact, (10; 11) is the pair of update periods associated with the highest number of parameterisations amongst the ones improving the best TS by a factor of two; (30; 31) is associated with the second highest number of parameterisations (see Figure 51). Finally as the maximum time considered is shortened, (10; 11) always provides more successful parameterisations than any other pair of update periods for *p* and *q* (see Figure 52). These results were obtained on the basis of average times after running TSDC across the benchmark 50 times. Given the randomness present in the optimisation algorithm, this allowed to single out with more confidence the most promising parameterisations. However, across the different analysis conducted over the best parameterisations, it has become apparent that there are no major variations between the set of best parameterisations identified when TSDC was run only once and when it was run 50 times. Even given the random elements of the algorithm, it is able to navigate the search space successfully and with reasonably consistent performance for a given parameterisation.

Finally, it is interesting to analyse the relative differences between the average performances of the 68 best parameterisations. In Figure 53, darker circles mark the fastest parameterisations. 100% transparency means that a given parameterisation finished in a time equals to half the one of the best TS; 0% transparency is assigned to the best parameterisation overall.

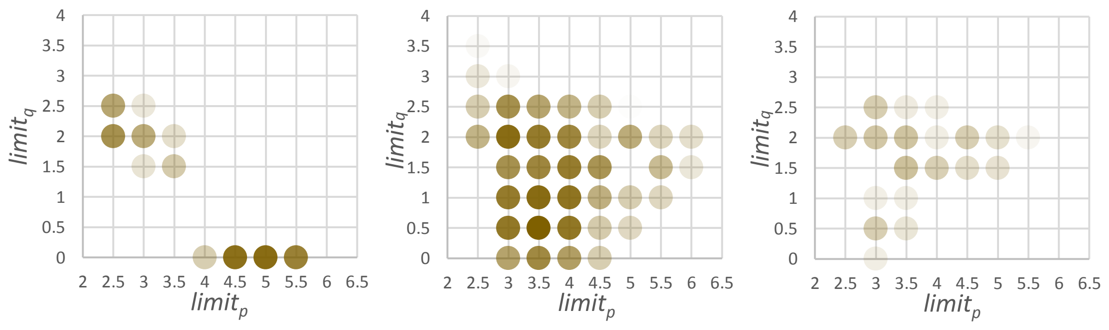


Figure 53 - Relative performance comparison between parameterisations (data is based on average times from 50 runs). Left: (*updatePeriod$_p$*; *updatePeriod$_q$*) = (3; 4); centre: (*updatePeriod$_p$*; *updatePeriod$_q$*) = (10; 11); right: (*updatePeriod$_p$*; *updatePeriod$_q$*) = (30; 31). The maximum time considered for a parameterisation corresponds to half the time taken by the best TS.

Figure 53 confirms that the fastest parameterisations are linked to the update periods (10; 11) and (3; 4). There are three especially good performances linked to update periods (3; 4), all of them considering 0 for *limit$_q$*: (*limit$_p$*; *limit$_q$*) = [(4.5; 0), (5; 0), (5.5; 0)]. The fastest parameterisation is, however, obtained with the update periods (10; 11), whilst setting *limit$_p$* and *limit$_q$* to 3.5n and 0.5n, respectively. Moreover, there seems to be a rectangular region of particularly good parameterisations linked to update periods (10; 11) in which *limit$_p$* varies from 3n to 4n and *limit$_q$* between 0 and 2.5n.

The considerations about TSDC's performances across the benchmark with the parameterisations of Figure 53 do not allow drawing a direct generalisation about how TSDC would perform with such parameterisations for all future problems. However, they enabled efficient performances of TSDC across a variety of 44 problems and as such are certainly worth considering for initial parameterisations to explore when tackling new problems. In particular, the ones identified as most promising in the previous paragraph, which are linked to update periods (10; 11) and (3; 4). It is valuable noting here as well, the robustness of the TSDC technique to parameter selection; using 424 parameterisations, approximately a fifth of all parameterisations defined, TSDC was able to finish the benchmark before the best time of TS. For the worst parameterisation of these 424, the cumulative time of solving each problem in the benchmark 50 times takes less than 15 seconds, which in turn, relates to solving all the problems once under 0.3 seconds (on average). This gives an indication that even if the user ends up not selecting one of the most promising, TSDC can still perform very acceptably across a wide range of parameterisations.

5.3.5 Is conquering even more efficient if tuned differently for each problem?

A set of promising parameterisations have been identified for future use with the TSDC approach, based on the efficient performances they give across the entire ASIL allocation benchmark. However, could it be that tailoring parameter selection on a per-problem basis results in even better performances? To investigate this, an experiment has been carried out where the parameterisations defined in section 5.3.3 are revisited; the best times and corresponding best parameterisations are recorder for each sub-problem. In an effort to capture a clearer picture of the differences between using a same parameterisation across the benchmark and tailoring parameterisations per sub-problems, each experiment was repeated 50 times. Table 36 depicts a performance comparison of the two parameterisation approaches, using the average times obtained across the 50 runs contemplated in the experiments. The times shown refer only to optimisation tasks.

Table 36 - Times to solve each of the sub-problems using the best parameterisation across the benchmark (left) and using the best parameterisation per sub-problem (center); the rightmost column shows the improvements of using the best parameterisation per sub-problem over utilising the best parameterisation across the benchmark (data is based on average times from 50 runs).

Sub-problem	**Time best parameterisation across benchmark (seconds)**	**Time best parameterisation per sub-problem (seconds)**	**Improvement over single parameterisation across benchmark (seconds)**
HBSM1-SP1	0.0013414	0.0013362	0.0000053
HBSM1-SP2	0.0019217	0.0018739	0.0000479
HBSM1-SP3	0.0044754	0.0034945	0.0009809
HBSM2-SP1	0.0013380	0.0013335	0.0000044
HBSM2-SP2	0.0019152	0.0018678	0.0000474
HBSM2-SP3	0.0979994	0.0699031	0.0280963
HBSM3-SP1	0.0014897	0.0014800	0.0000097
HBSM3-SP2	0.0023632	0.0022269	0.0001363
HBSM3-SP3	0.0815619	0.0557909	0.0257710
HBSM3-SP4	0.9582194	0.6006271	0.3575924
BSS-SP1	2.3626952	1.8429444	0.5197508
Total	3.5153205	2.5828783	0.9324424

For each sub-problem, there are parameterisations that obtain better performances than the ones achieved with the best parameterisation across the benchmark. When the improvements attained for the different sub-problems are added up, total gains amount to ≈0.93 seconds. These improvements correspond to saving roughly a quarter (27%) of the time that is consumed when a same parameterisation is used to tackle each sub-problem in the benchmark. In absolute terms, the sub-instance for which the impacts of tailoring parameterisations on a sub-problem basis are felt more, is BSS-SP1 – an improvement of ≈0.52 seconds is registered. In relative terms, HBSM3-SP4 benefits the most - its execution time is decreased by nearly 37%. For sub-problems of dimensionality four and two, very small improvements were obtained, both in relative and absolute terms. These problems present very small search spaces and a large number of parameterisations were able to find their optimal solutions in times very close to the best results presented in Table 36.

Tuning the parameters of TSDC on a per sub-problem basis gives faster performances. Whereas the improvements exist, they are less than one second across the benchmark; if the problem division step is considered, this means gains of a little more than 18%. These improvements are much less substantial than the ones obtained when there was an evolution from TS to TSDC; in that transition, execution times were cut down by roughly 10 seconds, meaning that TSDC was up to three times faster than TS.

So that these further enhancements can be obtained in practice, investigation needs to be conducted to evaluate the possibility of automatically setting TSDC's parameters as a function of problem characteristics. It should be pointed out, that this research would not just be important to improve TSDC's performance even further. In fact, if a stable, efficient relationship between problem characteristics and TSDC's parameterisations can be identified, the need from the user to experiment with different algorithm tunings would be eliminated. Based on the results obtained across the benchmark, a first effort was undertaken towards deriving a method for the automatic parameterisation of TSDC. Whereas this data did not allow establishing a definitive method, it enabled advancing some hypothesis and laying useful foundations for future investigation.

In researching a method to automatically setting TSDC parameters, one of the most obvious characteristics to investigate is problem dimensionality. Table 37 shows the dimensionality of each sub-problem in the benchmark, and the parameterisation with which TSDC was able to finish them faster. Sub-problems are sorted by an ascending order of dimensionality.

Table 37 - Sub-problems in the ASIL allocation benchmark, their dimensionality and TSDC's parameterisations giving the fastest performance for each sub-problem. Sub-problems are sorted in an ascending order of dimensionality.

Sub-problem	**Dimensionality**	***limit$_p$***	***limit$_q$***	***updatePeriod$_p$***	***updatePeriod$_q$***
HBSM1-SP1	2	7n	2n	1	2
HBSM2-SP1	2	6.5n	8.5n	1	2
HBSM3-SP1	2	10n	9.5n	1	2
HBSM1-SP2	4	0n	7n	1	2
HBSM2-SP2	4	0n	4.5n	1	2
HBSM3-SP2	4	8.5n	7n	1	2
HBSM1-SP3	8	2n	0n	1	2
HBSM3-SP3	24	3n	0n	1	2
HBSM2-SP3	26	2n	1.5n	3	4
HBSM3-SP4	26	1.5n	1n	3	4
BSS-SP1	84	6.5n	0n	3	4

The parameterisations in Table 37 only consider two update periods for *p* and *q*: (1; 2) and (3; 4). Larger sub-problems, of dimensionality 26 or greater, use the highest of the two pairs: (3; 4); in turn, smaller sub-problems make use of (1; 2). In regards to *limit$_p$* and *limit$_q$*, analysis of Table 37 does not yield such a clear picture. The biggest observable trend relates solely to three sub-problems - HBSM1-SP3, HBSM3-SP3 and BSS-SP1 – which all have *limit$_q$* set to 0n. Between these three cases, higher dimensionality problems have higher *limit$_p$* values; however, this growth seems to occur in a non-linear way. The coefficients multiplying by dimensionality to give the *limit$_p$* values are not constant; they increase with the sub-problems' dimensionalities, possibly indicating that the rate of growth of *limit$_p$* needs to increase with sub-problem dimensionality. One way to describe this type of relationship is through a quadratic model; using polynomial interpolation a function was found that passes through the points given by the dimensionalities and *limit*$_p$ values shown in Table 37 for HBSM1-SP3, HBSM3-SP3 and BSS-SP1 (see Equation 13). For dimensionalities ≥ 1, this function is always positive, continuously increasing, and it has a growing change rate.

$$limit_p = 0.057895n^2 + 1.647368n - 0.884211 \qquad \textbf{Equation 13}$$

It is important to emphasise that the relationship depicted in Equation 13 was merely obtained on the basis of three points mapping problem dimensionalities to *limit*$_p$ values. Furthermore, these three points were found from experiments with a set of parameterisations which do not necessarily present the required granularity for the optimal parameterisations to be identified. If an efficient relationship between *limit*$_p$ and dimensionality is confirmed, where *limit*$_p$ grows with dimensionality following an increasing rate of change, it is possible that other functions obtained from analysis of more data could provide a better model. That improved expression could be another quadratic function, or even another type of function (e.g. exponential).

An experiment has been carried out to evaluate if good performances could be obtained by TSDC when using parameterisations for *limit*$_p$ provided by the initial model in Equation 13 (whilst keeping *limit*$_q$ to 0). The parameterisations for (*updatePeriod*$_p$; *updatePeriod*$_q$) of Table 37 are maintained for each of the sub-problems. The rightmost column of Table 38 shows the results of the experiment. Values in bold represent cases for which improvements have been obtained over the results from using the best parameterisation across the benchmark. The results from the best parameterisation across the benchmark and from the best parameterisations per sub-problem are also given in the table for reference.

Table 38 - Times to solve each of the sub-problems using the best parameterisation across the benchmark (left), using the best parameterisation per sub-problem (center), and using parameterisations where *limit*$_p$ varies as per Equation 13 (right) (data is based on average times from 50 runs).

Sub-problem	Time best parameterisation across benchmark (seconds)	Time best parameterisation per sub-problem (seconds)	Time parameterisations with *limit*$_p$ growing as per Equation 13
HBSM1-SP1	0.0013414	0.0013362	0.0013438
HBSM1-SP2	0.0019217	0.0018739	**0.0019060**
HBSM1-SP3	0.0044754	0.0034945	**0.0034945**
HBSM2-SP1	0.0013380	0.0013335	0.0013429
HBSM2-SP2	0.0019152	0.0018678	**0.0019012**
HBSM2-SP3	0.0979994	0.0699031	**0.0963215**
HBSM3-SP1	0.0014897	0.0014800	0.0014945
HBSM3-SP2	0.0023632	0.0022269	Not finished after 1 sec
HBSM3-SP3	0.0815619	0.0557909	**0.0557909**
HBSM3-SP4	0.9582194	0.6006271	**0.8889190**
BSS-SP1	2.3626952	1.8429444	**1.8429444**

For seven out of the 11 sub-problems, better performances are achieved in comparison to the best parameterisation across the benchmark. Furthermore, for three other sub-problems, very acceptable times were also obtained; again, in comparison to the best parameterisation across the benchmark, the maximum increase in time happens for HBSM2-SP1 with roughly 4.9 microseconds (corresponding to 0.4%). Just for one sub-problem, HBSM3-SP2, the approach did not work well; the sub-instance could not be solved after one second (the time using the best parameterisation across the benchmark amounts to less than 3 milliseconds). However, according to Equation 13, *limit*$_p$ is to be set to seven moves in the case of HBSM3-SP2. It is interesting that if just one more move is considered, and *limit*$_p$ is set to eight for this sub-problem, TSDC's performance is considerably improved; in fact, better times are obtained than when the best parameterisation across the benchmark is utilised. In the same conditions, the two other sub-problems of dimensionality four are still executed in very efficient times; they take more time to be concluded than when TSDC is run with the parameterisation from the expression of Equation 13, but they are still finished faster than when TSDC is run with the best parameterisation across the benchmark. Eight moves may just be the minimum bound to be used across sub-problems of dimensionality four. From the experiments previously presented to find the best parameterisation per case study, setting *limit*$_p$ to 10 moves, for example, gives similar performances for sub-problems of dimensionality four as the ones obtained with *limit*$_p$ set to eight. As stated earlier, the model from Equation 13 was obtained with very limited data and it is likely than it needs improving. One with similar characteristics can be defined which enables good performances for TSDC in HBSM3-SP2 as well. For example, a function that adds one unit to the result provided by Equation 13 gives parameterisations that work well across every problem in the benchmark. Nevertheless, more problems need to be investigated in order to acquire an accurate model. It is important to study additional problems of the dimensionalities in Table 37 to evaluate if the *limit*$_p$ values identified for those dimensionalities can consistently provide good performances; furthermore, there is a big gap between dimensionalities 26 and 84 without representative problems that needs to be examined.

The hypothesis of setting parameterisations as a function of sub-problem dimensionality has been advanced. For (*updatePeriod*$_p$; *updatePeriod*$_q$), there seem to be benefits in growing their values with dimensionality. The sub-problems in the benchmark indicate that in tackling sub-problems with dimensionality under 26, (1; 2) should be used; in turn, sub-problems with dimensionality higher than 24 should be solved with (3; 4). Possibly, larger sub-problems could require greater update periods, but research with sub-instances with such characteristics needs to take place. For *limit*$_p$ and *limit*$_q$, it has been theorised that efficient performances could be found when *limit*$_p$ was set according to a non-linear function of dimensionality, keeping *limit*$_q$ to 0n. An initial function was proposed (see Equation 13) which yielded good performances for all problems in the benchmark except one. A single increment to the value of *limit*$_p$ defined by Equation 13 for that sub-problem allowed to obtain very efficient results; possibly the model from Equation 13 needs to be improved from more data. This study suggests that it may be possible to define efficient parameterisations for TSDC as a function of sub-problem dimensionality. However, more research needs to be undertaken with additional sub-problems in an effort to validate further this hypothesis; furthermore, if the hypothesis is supported, a study of more sub-instances will also be important to acquire an accurate relationship between sub-problem dimensionality and TSDC parameterisations.

Setting the parameterisations on the basis of dimensionality obtained some efficient performances from TSDC, but in finding the most promising parameterisations, other sub-problem characteristics may need to be considered. This is hinted by the best parameterisations per sub-problem presented in Table 37, which, in great part, do not reveal any clear relationship with dimensionality (at least in regards to *limit*$_p$ and *limit*$_q$). Another fact that points to the need of studying other characteristics is that a same parameterisation can yield very different performances for sub-problems of the same dimensionality. For example, sub-problems HBSM3-SP4 and HBSM3-SP3 are both of dimensionality 26. The best parameterisation of HBSM3-SP4 ranks 419th in the case of HBSM2-SP3! TSDC is almost three times slower than when using the best parameterisation for HBSM2-SP3. Dissimilarities between these two sub-problems are also found in the sets of parameterisations that enabled better performances of TSDC across each one of them when compared to using the best parameterisation across the benchmark. A first remark is that these sets integrate a different number of parameterisations: 100 parameterisations for HBSM3-SP4, and only 33 for HBSM2-SP3. Figures 54 and 55 show for HBSM3-SP4 and HBSM2-SP3, respectively, how these parameterisations are distributed as a function of *limit*$_p$ and *limit*$_q$.

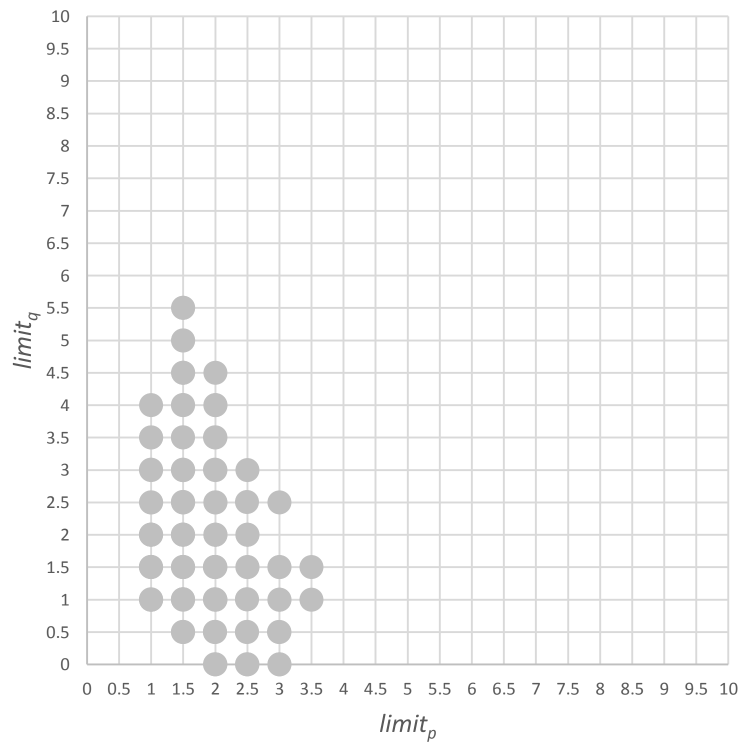


Figure 54 - Distribution, as a function of *limit*$_p$ and *limit*$_q$, of the parameterisations that allowed TSDC to complete HBSM3-SP4 faster than when using the best parameterisation across the benchmark (data is based on average times from 50 runs).

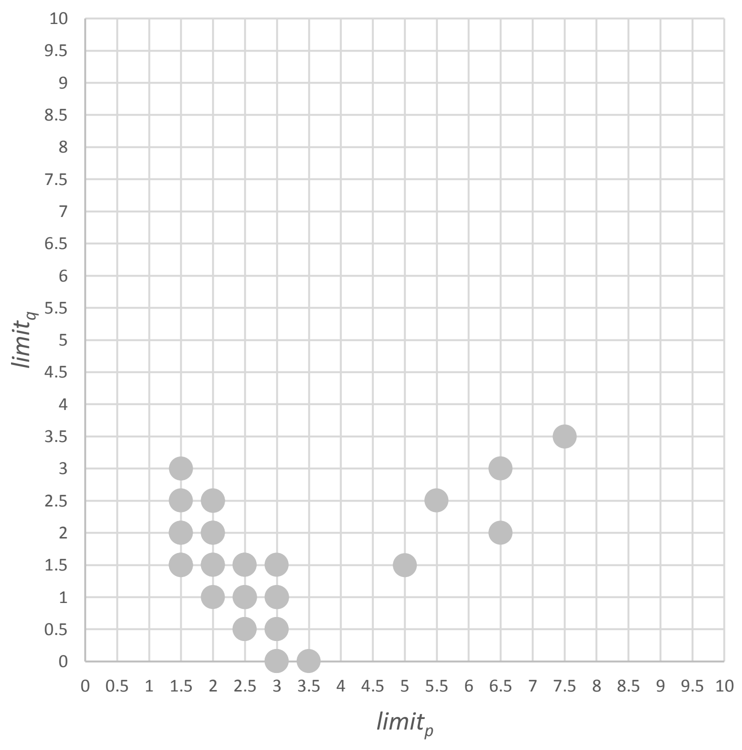


Figure 55 - Distribution, as a function of *limit*$_p$ and *limit*$_q$, of the parameterisations that allowed TSDC to complete HBSM2-SP3 faster than when using the best parameterisation across the benchmark (data is based on average times from 50 runs).

The minimum value for *limit*$_p$ is smaller in the case of HBSM3-SP4 than for HBSM2-SP3 (1n versus 1.5n). The maximum value of *limit*$_p$ for HBSM3-SP4 is less than half the one of HBSM2-SP3 (3.5n versus 7.5n). In regards to *limit*$_q$, its minimum value is the same for both sub-problems (0n); as for its maximum value, the one of HBSM3-SP4 is larger than the one of HBSM2-SP3 (5.5n versus 3.5n). In terms of (*updatePeriod*$_p$; *updatePeriod*$_q$), HBSM3-SP4 is the only sub-problem of the two presenting parameterisations with (30; 31). These observations support the theory that TSDC can perform differently with a same parameterisation for sub-problems of the same dimensionality and that, in turn, the best parameterisations are possibly only achievable if other sub-problems characteristics are considered. It is worth noting, however, that there is a set of parameterisations which for both problems improve TSDC's performance in comparison to when the best parameterisation across the benchmark is used. There are 15 common pairs of values for *limit*$_p$ and *limit*$_q$ and three common parameterisations for (*updatePeriod*$_p$; *updatePeriod*$_q$): (1; 2), (3; 4); (10; 11). In total, the two sub-problems share 26 overall parameterisations. This suggests that a set of parameterisations could be available which enables efficient performances across different sub-problems of the same dimensionality, giving further strength to the possibility of automatically setting TSDC with parameterisations as a function of problem dimensionality. Notwithstanding, it is not guaranteed that the set of parameterisations allowing for efficient performances across the multiple sub-problems of the same dimensionality includes the optimal parameterisation for any of the sub-instances; in the particular case of HBSM3-SP4 and HBSM2-SP3, as introduced earlier, the best parameterisation found for HBSM3-SP4 performs very poorly for HBSM2-SP3. The results for the remaining sub-problems in the benchmark that present the same dimensionality, were also analysed to evaluate if common efficient parameterisations existed. Whereas this was always the case, it is important to say that these sub-problems are of dimensionalities four and two and for them, in general, a vast number of parameterisations performed very well.

Finding a method to automatically parameterise TSDC as a function of the problem to solve is important as it provides improved optimisation performance. Furthermore, if the method can be applied generically to any problem, it eliminates the need for parameterisation experimentation from the user side. On the basis of the benchmark defined in this thesis, this section presented an important effort towards such an approach. An initial method has been proposed based on sub-problem dimensionality, which works across well across all the sub-problems in the benchmark, except one.

However, small modifications to the approach extend the good results to that remaining problem. Furthermore, based on the analysis of the data produced across the different sub-problems, there have been indications that other characteristics may need to be investigated in order to obtain the most efficient performances. These characteristics are not clear from the results obtained; two constraints-related characteristics that may be worth investigating are AACs size and AACs ASIL requirement. The automatic parameterisation of TSDC is not a trivial problem and it requires a substantial additional amount of work, namely with the study of a variety of new case studies. This investigation can be pursued as future work.

5.3.6 Initial evaluation of parallel conquering

Currently, multi-core processors are widespread across desktops, laptops and mobile phones; this is also the case with servers in corporate environments. Intel, one of the biggest processor manufacturers, currently sells a range of 1054 different processors, of which only 169 are single-core (Intel Corporation, 2015). Intel’s desktop and mobile processors include up to eight cores; in the case of server processors, this number grows up to 61 in Intel’s Xeon Phi family!

Multi-core processors can execute multiple independent tasks in parallel; this potentially saves time when compared to the scenario where all tasks are executed sequentially. Sub-problems generated in the Divide and Conquer approach are completely independent and are natural candidates to be solved concurrently. Given the current widespread use of multi-core processors, it is interesting to investigate if TSDC’s performance can be improved further through parallelisation. In this section, an initial approach for parallel conquering is described. TSDC is then applied to the ASIL allocation benchmark, making use of the parallelisation approach, and the results are compared to the scenario where sub-problems are solved sequentially.

The general concept of parallel conquering is quite intuitive: when a series of sub-instances are generated from an original problem, they are assigned to a team (or pool) of threads, which are executed concurrently. This can be illustrated with the example introduced in section 5.3.1 of a problem from which three independent sub-instances are obtained (see Figure 56).

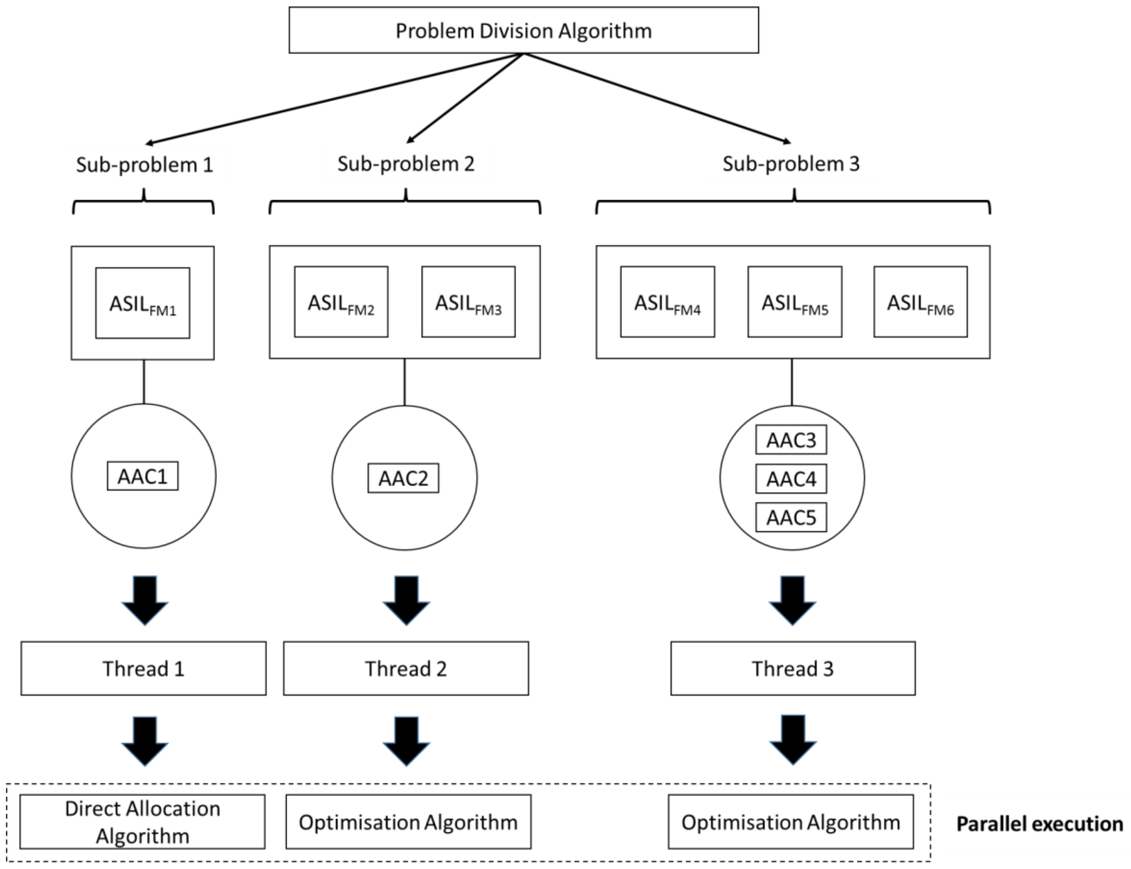


Figure 56 - Parallel divide and conquer approach overview for an illustrative problem with three sub-instances.

Three threads are created to deal with sub-problems 1, 2 and 3. The Direct Allocation Algorithm, and two instances of an optimisation technique (e.g. TS), are run in parallel to solve the three sub-problems. If the sub-problems take the same time to be completed, the total execution time can ideally be reduced by a factor of three. However, as it could be observed with the experiments of previous sections, sub-problems can take very different times to be executed. Also, there is always an overhead of thread creation and sub-problem to thread scheduling that reduces this factor.

Sub-problems take different times to be solved, and these times are not available before execution. Furthermore, since the optimisation algorithm includes randomness in its search process, even for a same sub-problem, execution times can vary. In this way, it is not feasible to build an exact sub-problem to thread schedule that minimises execution time. In the approach proposed here, each of the available threads is initially allocated with one sub-problem, and as soon as they finish their initial sub-instance, they are immediately assigned with another from the set that has been kept pending. This process is followed until no more sub-instances are left to solve. The strategy basically works for a load balance between the different threads; it avoids having threads that at some point have no sub-problem to execute, whereas others might still have multiple sub-instances in their work queue. This is a common tactic in parallelisation when the effort of the different tasks to be solved pose varying efforts.

Disregarding for now the overhead of thread creation and sub-problem to thread scheduling, in the scenario where the number of threads is equal to or higher than the number of sub-instances, the total execution time is given by the duration of the longest sub-problem (see Figure 57).

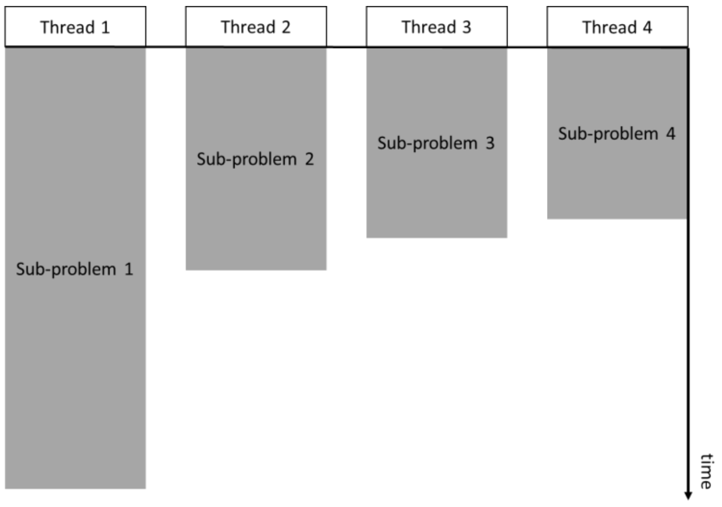


Figure 57 - Possible allocation of four sub-problems with different execution durations to four threads. The total execution time is given by the duration of the longest sub-problem.

If there are more sub-problems than threads, it is a given that the execution time will at least be as high as the duration of the longest sub-problem. Acknowledging this, an initial sub-problem solving order is proposed in which the most complex sub-instances have a higher priority to be executed. If the most complex sub-problems are scheduled to be solved first, then simpler sub-instances wait until they can be allocated to a thread and, possibly, the minimum execution time can still be met (See Figure 58).

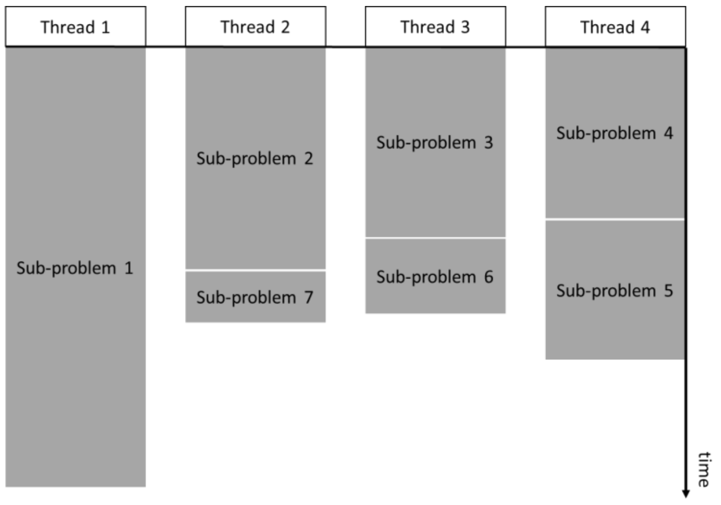


Figure 58 - Allocation of seven sub-problems with different execution durations to four threads. Sub-problems are allocated to avaialable threads in a decreasing order of execution duration. The total execution time is given by the time taken to solve the longest problem.

In this example, the execution time is given by the duration of sub-problem 1. Also, note that if thread 1 and sub-problem 1 are disregarded, the others have also achieved an efficient distribution. Now take the example of Figure 59, where the same sub-problems are randomly scheduled to be solved.

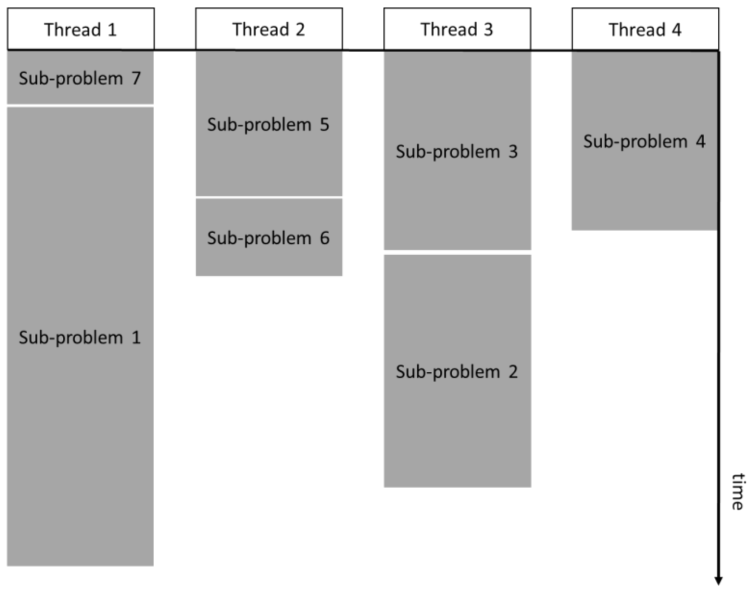


Figure 59 - Allocation of seven sub-problems with different execution durations to four threads. Sub-problems are allocated to available threads in a random order. The total execution time is given by the duration of the longest problem, sub-problem 1, combined with the duration of sub-problem 7.

Since the longest sub-problem is not solved initially, the minimum execution time cannot be obtained anymore. Also, note that the sub-problems in the remaining threads are not distributed efficiently.

It is important to emphasise here that this approach is not based on knowledge about exact execution times, but rather on the relative differences in the times taken to solve the different sub-problems. To obtain these relative differences, the experiments carried out in the previous sections have given indications that useful inferences can be made if sub-problem dimensionality and number of AACs are analysed. The performances of the best parameterisation across the entire benchmark and of the best parameterisations per case study have been introduced in section 5.3.5; Table 39 recovers that information and adds data on sub-problem dimensionality and number of AACs.

Table 39 - Dimensionality and number of AACs of each sub-problem in the benchmark, and the times taken by TSDC to solve each one of them across all cost functions. Times on the left hand side refer to using the best parameterisation across the benchmark; the times on the right hand side refer to the using the best parameterisations per case study (data is based on average times from 50 runs).

Sub-problem	Dimensionality	Number of AACs	Time best parameterisation across benchmark (seconds)	Time best parameterisations per case study (seconds)
HBSM1-SP1	2	1	0.0013414	0.0013362
HBSM1-SP2	4	4	0.0019217	0.0018739
HBSM1-SP3	8	16	0.0044754	0.0034945
HBSM2-SP1	2	1	0.0013380	0.0013335
HBSM2-SP2	4	4	0.0019152	0.0018678
HBSM2-SP3	26	61	0.0979994	0.0699031
HBSM3-SP1	2	5	0.0014897	0.0014800
HBSM3-SP2	4	12	0.0023632	0.0022269
HBSM3-SP3	24	1296	0.0815619	0.0557909
HBSM3-SP4	26	10256	0.9582194	0.6006271
BSS-SP1	84	8096	2.3626952	1.8429444

The longest sub-problems are consistently the ones with higher dimensionality. Furthermore, between sub-problems with the same number of variables, but which integrate a different number of AACs, the ones with the higher number of constraints are the ones taking longer to be solved. This conclusion also holds true for size one sub-problems. The Direct Allocation Algorithm works by verifying each of the ASIL requirements that the single variable of a given size one sub-instance needs to comply with; size one sub-instances with more AACs take a longer time to be completed. If all sub-instances execution times are analysed together as a function of a specific cost heuristic, there are only a few cases where inferences on relative differences between execution times do not hold true. It is important to say here that in situations where execution time inferences are not verified, the dynamic sub-problem to thread scheduling method adopted can be important in minimising total execution time. Consider the example in Figure 60, which shows a situation where four threads are available and seven sub-problems exist. Inferences are made about the processing time to execute them and longer sub-problems are scheduled to be solved first.

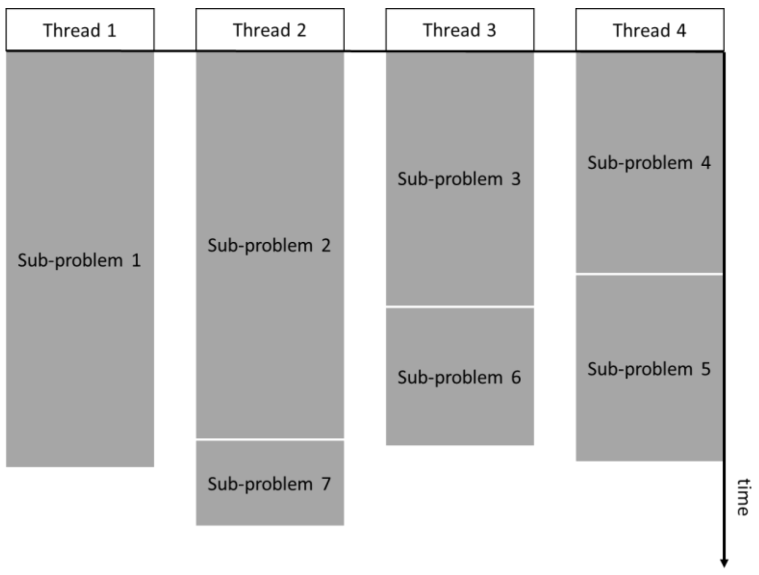


Figure 60 - Threads and the sub-problems they execute over time. Inferences about the relative durations of sub-problems hold true and longer sub-problems are executed first.

The scheduling adopted allows reaching the minimum total execution time for this set of sub-problems. Now, contemplate the situation where the inferences made about sub-problems 5 and 6 are not correct. In reality, it is the case that sub-problem 5 takes less time to be completed than sub-problem 6, to a point where the total time to execute sub-problems 3 and 6 is greater than the one of executing sub-problems 4 and 5 (see Figure 61). Sub-problem 2 now takes more time than executing sub-problems 4 and 5 sequentially.

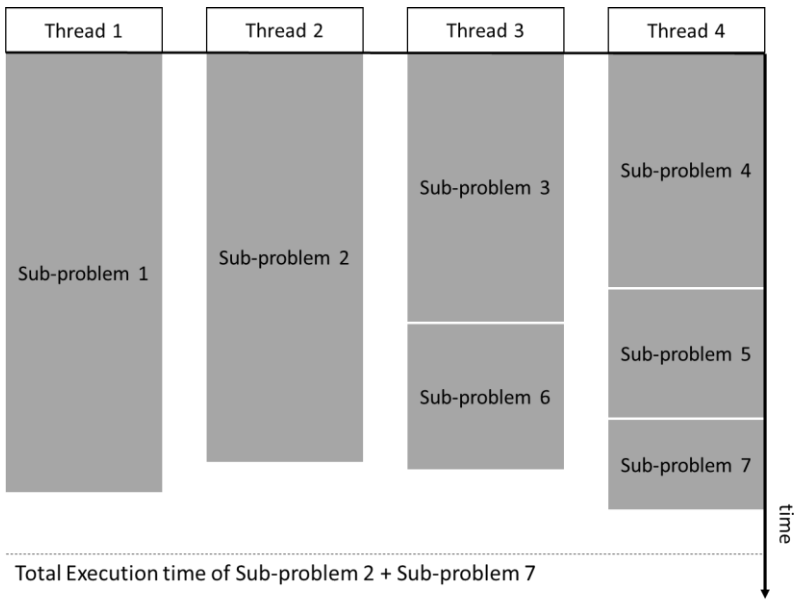


Figure 61 - Threads and the sub-problems they execute over time, using dynamic sub-problem to thread scheduling. Inferences about the relative durations of sub-problems 5 and 6 do not hold true. Dashed line represents the total time consumed when a fixed sub-problem to thread scheduling is used.

In this situation, the dynamic thread load balancing scheme assigns sub-problem 7 to thread 4, after sub-problem 5 is finished. If a fixed sub-problem to thread scheduling had been defined before the sub-instances started being solved, sub-problem 7 would have been executed after sub-problem 2 and the total execution time would be greater than the minimum possible. This example illustrates how dynamic scheduling can be important when there are incorrect inferences about the sorting of the sub-problems. Changing the same example slightly can also serve to illustrate how even if all inferences hold true, dynamic scheduling can be key in dealing with the uncertainty about exact execution time of the sub-problems. Consider that sub-problem 5 takes longer to solve than sub-problem 6, as it was initially expected. The total execution time of sub-problems 5 and 4 can still be smaller than the total time of sub-problems 3 and 6, and also than the one of sub-problem 2 (see Figure 62).

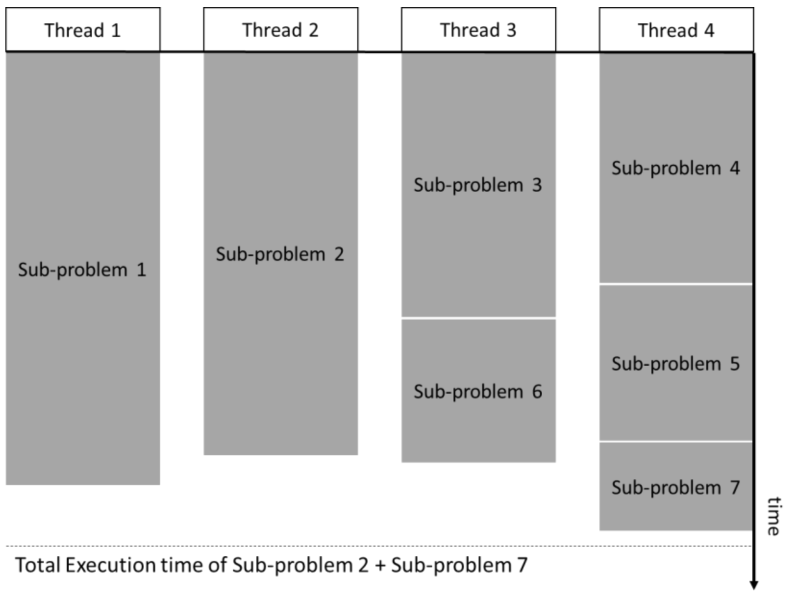


Figure 62 - Threads and the sub-problems they execute over time, using dynamic sub-problem to thread scheduling. Inferences about what problems take longer hold true, but their exact durations are unkown. The dashed line shows a possible total time for a fixed sub-problem to thread scheduling.

In this case, using dynamic scheduling, sub-problem 7 would be solved after sub-problem 5, and time would still be saved when compared to a situation where a fixed schedule would initially establish that sub-problem 7 is to be executed after sub-problem 2 in thread 2.

Having presented an approach for the parallelisation of ASIL allocation sub-problem conquering, it is now time to evaluate if it can bring further improvements to the TSDC optimisation method. The ASIL allocation benchmark will be used again for the evaluation. TSDC with parallel conquering is referred to as Parallel TSDC from now onwards. Experiments have been carried out on a machine using an Intel i5-3570K processor, which includes four cores and can handle up to four threads concurrently. With this processor, it was possible to experiment with scenarios where a maximum of 1, 2, 3 or 4 threads could be created. Tests with a higher numbers of threads were not considered; this would likely originate extra overhead, namely with threads being interrupted for others to run. Furthermore, in an initial effort to evaluate the possible benefits of using the sub-problem execution order proposed in this section, where the most complex sub-problems are executed first, results are compared to when sub-problems are solved in a random order. The two modes will be referred to as *Complex First* and *Random Sorting* henceforth.

The Tabu Search algorithm includes randomness in its search process, which means that if a same sub-instance is run multiple times there will be variations in execution time. Since sub-instances are dynamically allocated to threads, these execution time variations can impact the overall sub-problem to thread scheduling scheme. Acknowledging this, to promote a fair comparison between experimental setups, the optimisation algorithm has been forced to behave exactly in the same manner when tackling a given sub-problem. Randomness in the implementation of the Tabu Search method is provided by a pseudo-random number generator. The generator accepts a seed, which determines the sequence of pseudo-random numbers it outputs. When fed with different seeds, it produces different sequences of pseudo-random numbers; if fed with the same seed, the exact same sequence is obtained. In the experiments carried out, for an experimental setup with a given number of threads and sub-problem sorting mode, the seeds used to initialise the pseudo-random number generator are stored and then re-used for the remaining experimental setups. Furthermore, in an effort to stretch further the differences between the different experimental setups, the benchmark is re-run 50 times for each one of them.

As with all algorithms presented thus far, Parallel TSDC was implemented using C++ in Microsoft Visual Studio 2012 (Microsoft, 2015). In this initial attempt at parallelising TSDC, OpenMP 2.0 (OpenMP ARB, 2015) was utilised. OpenMP is a simple, directive-based API for multi-platform shared-memory parallel programming and it is supported by the Microsoft Visual Studio C++ compiler, Visual C++. In the implementation of Parallel TSDC, the different sub-instances are initially stored in a vector, which is then sorted according to the sub-problem sorting mode of the current experiment; a *for loop* can then be used to iterate through the multiple sub-problems using the indexes of the vector. OpenMP allows to parallelise a *for loop* simply by preceding it with the following compiler directive:

```
#pragma omp parallel for
//for loop declaration
```

Additional options can be set in the directive. The *num_threads* clause is used to define the size of the pool of threads to be created. If one wishes to create four threads to run the iterations of a *for loop*, the directive presented next can be used:

```
#pragma omp parallel for num_threads(4)
//for loop declaration
```

Furthermore, as mentioned in the description of the approach for parallel conquering, threads are to be allocated with a new sub-problem as soon as they are done with another. This relates with threads being assigned with an iteration of the *for loop* as soon as they complete the current one. In OpenMP this behaviour can be defined by setting the *schedule* clause to “dynamic”, as shown now:

```
#pragma omp parallel for num_threads(4) schedule(dynamic)
//for loop declaration
```

One issue encountered here, is that while using dynamic scheduling, there is no guarantee that the iterations of the *for loop* will be visited in order, i.e., implementations of OpenMP can decide that the iteration 4 of a for loop is assigned to be executed by a thread before iteration 3. In this way, the effect of sub-problem sorting would be lost. Experimentation has however revealed that across the benchmark, in the vast majority of situations, the desired execution order is maintained. This allows to compare with some confidence the two sorting modes that were utilised.

Experiments have been carried out using the best parameterisation found across the benchmark, as identified in section 5.3.4. The parameterisations are recovered in Table 40.

Table 40 - Best parameterisation across the ASIL allocation benchmark obtained in section 5.3.4.

$limit_p$	$limit_q$	$updatePeriod_p$	$updatePeriod_q$
3.5n	0.5n	10	11

The average results after running each experimental setup 50 times can be observed in Figure 63. Times only refer to sub-problem solving tasks; problem division and sub-instance sorting overheads are not included. The yellow line represents the total time taken to solve solely the most complex sub-instance of each problem. It is included in the chart as a constant value; it is neither linked with a sorting mode or maximum number of threads.

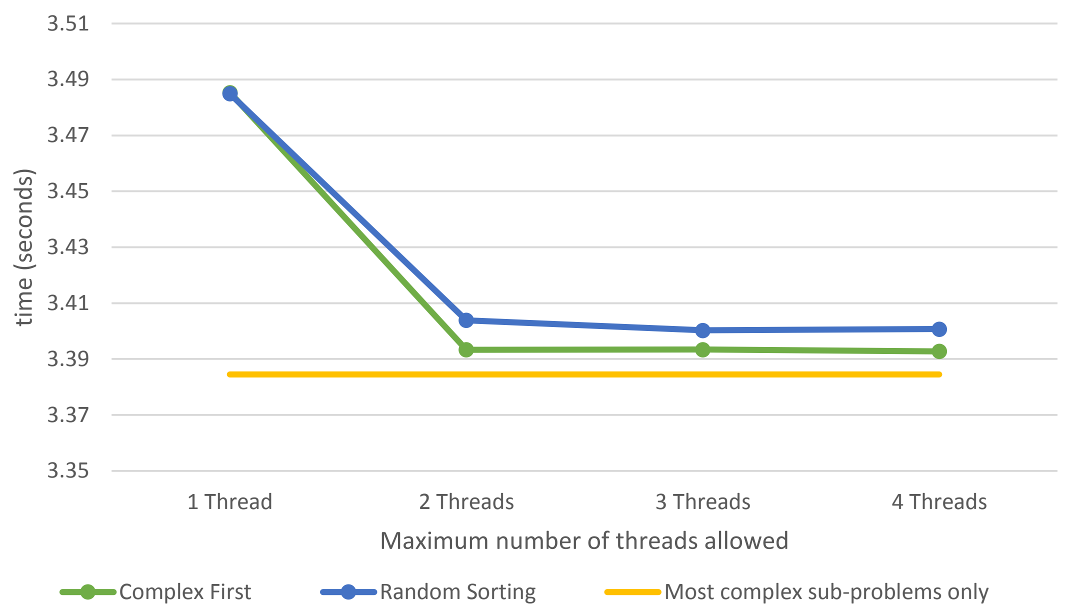


Figure 63 - Parallel TSDC performance in solving the ASIL allocation benchmark, as a function of the maximum number of threads available and sub-problem sorting mode (data is based on average times from 50 runs).

Figure 63 shows that independently of the sorting mode adopted, the use of parallelisation always improves the time to finish the benchmark. The Complex First sorting mode gives the biggest improvements, regardless of the number of threads used. When Random Sorting is utilised, a small improvement is noticeable with an increase from two to three threads. However, in the case of Complex First, the times change very little across two, three and four threads. This can be possibly explained by the characteristics of the benchmark. Analysis of Table 39 indicates that in general, for the different problems in the benchmark there is one sub-instance that takes much longer to solve than the remaining. Likely, when using two threads alone, whilst the most complex instance is solved by one thread, nearly all of the work regarding the least complex sub-problems can be finished by a second thread. When going from two to three threads, and from three to four threads, it is possible that small improvements that can be achieved by Complex First are masked by overhead of extra thread creation, for example. Furthermore, from three to four threads the difference between Complex First and Random Sorting is not reduced. It is hypothesised that this is caused by a more noticeable interference from the operating system. The processor where experiments were carried out on integrates four cores; when four threads are used by the Parallel TSDC, the operating system will necessarily have to interfere with the execution of at least one of them if it has other tasks to compute. Arguably, it is more likely that a thread executing many small tasks would have a gap in execution at the point that the operating system needs to execute another task. When Complex First is used, the longest problem is executed in one thread, and the remaining three threads will be in charge of solving all the simpler sub-problems. If the operating system needs to execute other tasks, it will then likely execute them in between the sub-problems allocated to these three threads. Given that for each problem in the benchmark the most complex sub-instance is much lengthier than the remaining, the simpler sub-instances and other operating system tasks can possibly be executed in parallel without affecting the time to solve the entire problem. When Random Sorting is used, however, simpler sub-problems can initially be scheduled to all four threads; if the operating system needs to execute other tasks and it slots them between sub-problems across the different threads, then, the execution time before the most complex sub-instance increases, and so does the total execution time of the problem being tackled. It is possible that this effect can also have some impact when a smaller number of threads is used. Whereas the justification provided is just a hypothesis, it would be certainly interesting to conduct investigation on this matter as further work.

The difference between the times obtained by the most efficient sorting mode, Complex First, and the time taken to solve only the most complex sub-problems could be explained by the overhead of multiple thread creation and scheduling. Another possible contributing factor is the fact that the OpenMP implementation sometimes does not respect the sorting order defined. Finally, the need of the operating system to execute other tasks can also play a role here. When only the most complex sub-instance of each problem is solved, only one thread is created; the operating system can schedule it to a single core and run the other tasks, in parallel, in the remaining cores. When multiple threads are created, more cores are likely used, and arguably there is a higher probability that other tasks that the operating system needs to run will interfere. It has been hypothesised that these tasks would be more likely scheduled to execution gaps of threads solving multiple small sub-problems, but it is also possible that the operating system interferes with the execution of the thread of the longest sub-problem.

Using the Complex First sorting mode, it has been possible to recover approximately 92% of the time that separates running all tests without parallelisation and only solving the most complex sub-problems. Here, “most complex sub-problems” stands for the ones deemed as such from the analysis of dimensionality and number of AACs. However, since TS includes randomness in its search process, it is possible that variations in times to finish the sub-problems can be present, which in turn could impact the considerations about the longest sub-instances and the correctness of the sub-problem sorting adopted. There are 16 problems in the benchmark, and each of them is run 50 times; this amounts to a total of 800 problems. Furthermore, as stated previously, the trials with these 800 are repeated 50 times, resulting in a final count of 40,000 problems. A study about the execution times of the sub-instances in each of these 40,000 problems was carried out and it revealed that only in 198 occasions the complexity sorting order given by the method proposed in this section was not right.

These results further validate the rationale proposed to sort problems by complexity. Moreover, from the same study it was possible to observe that only in 28 occasions the longest sub-instance is not the one deemed as such by the same method. These 28 occasions represent a rather small increase (under 60 microseconds) to the average time considered for running the most complex sub-problems presented in Figure 63. This gives further strength to the conclusion that 92% of the maximum time that could be gained through parallelisation was saved. In a final effort to confirm this value, the execution times of the 40,000 problems were analysed for an ideal parallelisation scenario (i.e. without considering overhead of thread creation and scheduling, for example) where four threads could be run concurrently; it was verified that if the longest sub-problem is scheduled to be solved initially, all other sub-problems can be solved in parallel in a smaller time. This means that in an ideal parallelisation scenario with four threads, the execution time of the longest sub-instance would give the total execution time for a problem.

The comparison between Complex First and Random Sorting has to this point been made solely on the basis of the time taken to execute the different sub-problems. Random Sorting is meant to simulate the scenario where there is not a concern about the order for sub-problems execution and therefore the order in which the problem division algorithm produces them can directly be taken. This is to say that in a real usage situation, there is no sorting overhead related to following a random execution order. However, in making use of Complex First, the overhead exists as there is no guarantee that the problem division algorithm will produce the sub-instances in the desired order. In the experiments carried out, the sorting overhead associated with Complex First amounts to roughly 0.056 seconds, which overcomes the improvements this sorting approach achieves over Random Sorting. This conclusion is true for any number of threads - the maximum time gained by Complex First over Random Sorting happens for two threads and it amounts to ≈0.011 seconds. In this way, Random Sorting is deemed the most promising approach. When three and four threads are used, it allows to recover approximately 84% of the maximum time that could have been saved through parallelisation.

The approach proposed in this thesis for Parallel TSDC improves the time consumed to finish the benchmark. Enhancements can be felt as soon as two threads are used. However, whilst the improvements do exist, they are small. Approximately 0.085 seconds are recovered through parallelisation. This represents reducing the time spent in optimisation tasks only by ≈2.44%. If the time spent in problem division is also taken into account, the time saved across the benchmark through parallelisation amounts solely to ≈1.67%. This can be explained by the characteristics of the benchmark, where for the different problems there is a sub-instance that is more complex and then others which are in general much faster to solve. Whilst the characteristics of the problems in the benchmark inherently limit the improvements that can be achieved by parallelisation, it would certainly be interesting to study different instances - namely problems where the relationship of a single dominant sub-instance is not present. Furthermore, it would also be important to study the impacts of parallelisation when tackling even larger, more complex problems. In a first effort in that direction, a larger model has been simulated; basically, for each case study - HBSM1, HBSM2, HBSM3 and BSS - their respective sub-problems are taken and each replicated 50 times to form the Ultimate ASIL Allocation Problem (UAAP). The UAAP is formed by a total of 550 sub-instances of size greater than one and 7150 of size one. The Experiential-I cost heuristic is used to evaluate the allocations across this new problem. To evaluate Parallel TSDC's performance across the UAAP, the setups of the experiments previously carried out in this section are reused. Results are shown in Figure 64; time consumed with problem division and sub-problem sorting steps is not included.

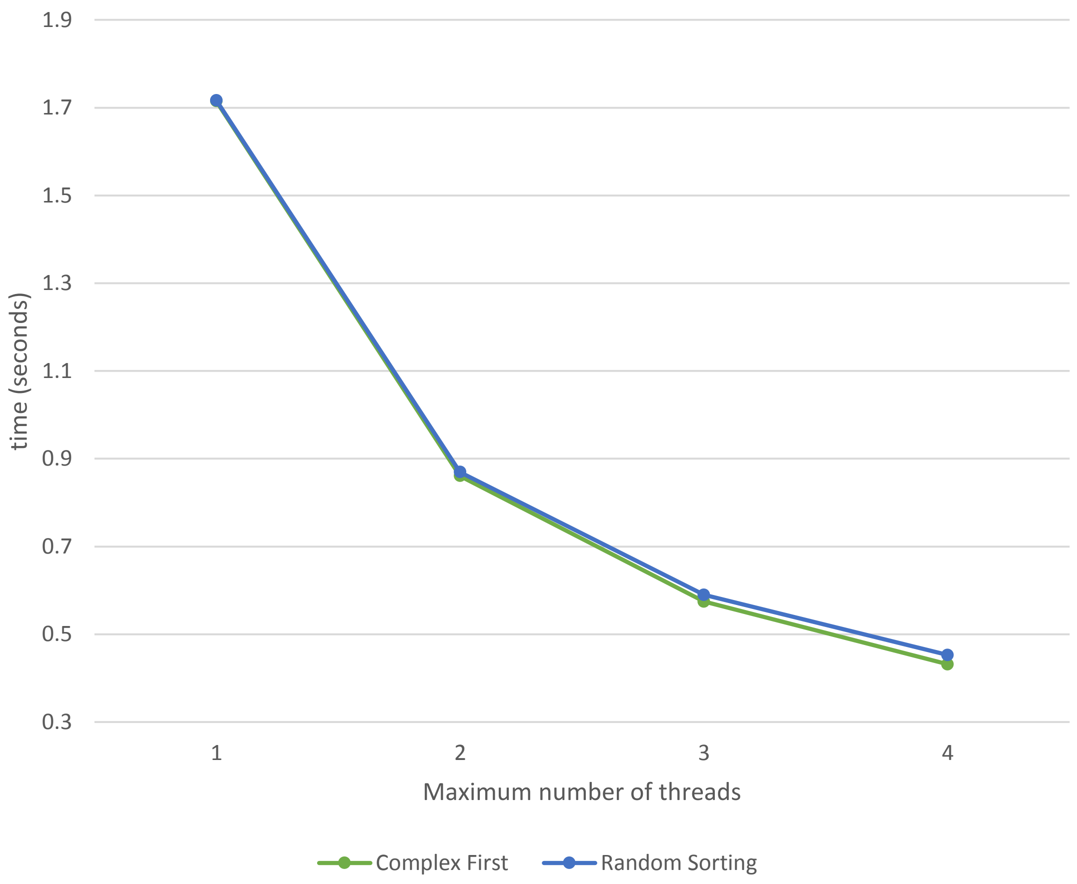


Figure 64 - Parallel TSDC performance in solving the ultimate ASIL allocation benchmark, as a function of the maximum number of threads available and sub-problem sorting mode (data is based on average times from 50 runs).

For both sub-problem sorting modes, with the increase of the maximum number of threads always comes an important decrease in execution time. Complex First is faster than Random Sorting when two or more threads are used. However, the differences are very small in comparison with the overall running time. The biggest difference between the two sorting modes is approximately 0.021 seconds (for four threads). As mentioned in the description of the parallel conquering approach, a thread gets assigned with a new sub-problem as soon as it finishes another, and therefore there is always a constant effort towards thread load balancing. The worst case scenario for the execution time of a given problem relates to threads reaching a perfect load balance just before the last sub-instance is scheduled to be solved, and then that last sub-instance being the most complex. The maximum difference between the worst and the ideal sub-problem sorting modes would actually be even smaller than the time taken to solve the complex sub-problem. It would be obtained by subtracting the execution time of the complex sub-problem by the same execution time divided by the number of threads. This corresponds to the situation where the work associated with solving the complex sub-instance is equally divided by the number of available threads.

In the results of Figure 64, the differences between the two sorting modes grow with the number of threads, which could link again to situations in Random Sorting where one complex sub-problem appears towards the end of the sub-instance execution queue. One thread would be assigned to deal with the complex sub-problem, whereas the other threads would be in charge of solving the remaining work (which here is assumed to be less than the one of executing the complex sub-problem). More threads dealing with the smaller amount of work can ideally finish it faster; this means more resources not being in use until the complex sub-problem is finally finished and therefore that a worse thread load balance has been accomplished.

Complex First has always achieved better results than Random Sorting. Further analysis of the quality of results by Complex First can be made on the basis of the data presented in Figure 65. The performance curve of Parallel TSDC with Complex First is shown together with three lines representing the time of solving all the sub-problems sequentially, divided by 2, 3 and 4.

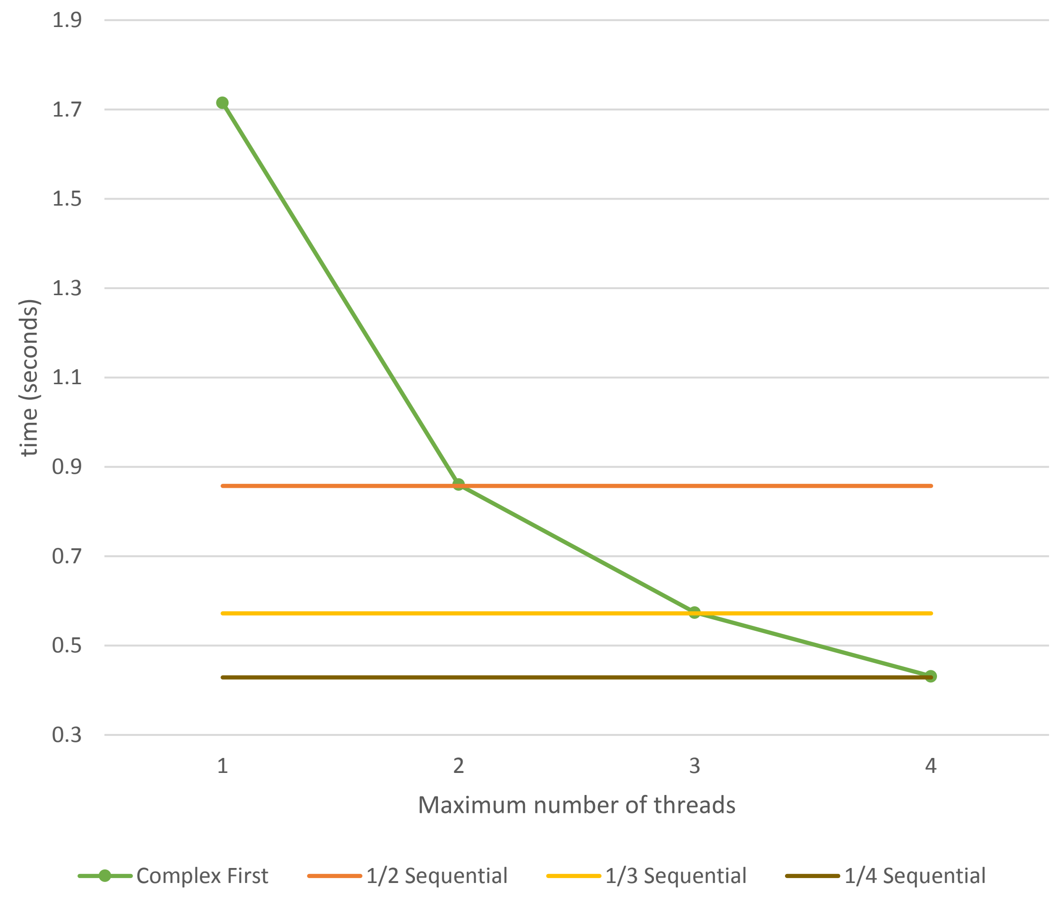


Figure 65 - Comparison of Parallel TSDC's performance using the Complex First sorting mode against the ideal parallelisation times relating to the use of 2, 3 and 4 threads (data is based on average times from 50 runs).

The use of Parallel TSDC with Complex First gave the biggest improvements, to the point where the execution times are very close to ideal divisions of the time to solve the sub-problems sequentially by the number of threads used. In every scenario, the differences are between 2 and 3 milliseconds; this indicates a very good thread load balancing, independently of the number of threads. The UAAP contains a very high number of sub-problems, and many of them are very fast to execute; in the Complex First sorting method, the smaller sub-instances are left to be solved towards the end.

This allows for thread load balancing to be executed with high precision. Also, the small differences to the ideal times hint to a minimal overhead with thread creation and scheduling. In fact, a smaller overhead was to be expected here, in comparison with the previous experiments concerning the ASIL allocation benchmark. In the UAAP, thread creation only occurs once, whereas in the experiments with the benchmark, every time a different problem was to be tackled, threads would be created. Given that there are 16 problems in the benchmark (four case studies, each being associated with four cost heuristics), and that they would get executed 50 times, threads would be created in 800 occasions. Furthermore, the overhead with thread scheduling is expected to be smaller in the UAAP. The number of sub-problems present in the UAAP is only a quarter of all the sub-problems in the ASIL allocation benchmark: the sub-problems in the benchmark are obtained from four case studies associated with four cost functions, whereas in the UAAP the sub-problems relate to the same four case studies, but they are only associated with the Experiential-I cost function. Since there are less sub-problems to solve in the UAAP, thread scheduling happens less often.

Although Complex First achieved the best performance for sub-problem conquering, again there is the need to consider the overhead of sub-problem sorting which only exists while using the Complex First approach. For the UAAP, this overhead amounts to ≈0.015 seconds. It overcomes the improvement of Complex First over Random Sorting when two threads are used; Random Sorting becomes faster by roughly 6 milliseconds. When three and four threads are utilised, Complex First remains faster, but its improvements over Random Sorting are reduced to less than 1 and 6 milliseconds, respectively. In this way, depending on the number of threads available, the best sorting approach varies. A common evidence across the different number of threads is that the sorting overhead brings the performances of the two approaches even closer; for any given number of threads, the difference between the two sorting methods amounts to less than 1% of the ideal improvement that parallelisation could bring over executing sub-problems sequentially. When analysing the benchmark, however, Complex First is worse than Random Sorting independently of the number of threads utilised. Furthermore, in its best performance, the former only recovers approximately 64% of the maximum improvement that could be achieved through parallelisation, versus roughly 84% from Random Sorting. On the basis of these experiments, it seems more advantageous that Parallel TSDC executes the different sub-problems in the order they are provided by the problem division algorithm. The use of Random Sorting also contributes to reduce complexity in the code. Nonetheless, Complex First was always more promising when only the time to execute the different sub-instances was considered and would be worth revisiting if the time to sort the different sub-problems by complexity can be somehow minimised.

In the UAAP, increasing the number of threads always resulted in an improved performance. In experiments with the benchmark, whereas increasing the number of threads did not always yield improvements, it also did not make results worse. In this way, in future uses of the approach Parallel TSDC, it is recommended that the maximum number of threads that can be executed concurrently is utilised.

This section presented a parallelisation approach which successfully improved the performance of TSDC across the ASIL allocation benchmark. Improvements were not very significant, but this was mainly because generally, for different the problems in the benchmark there was one sub-instance that took much longer to solve than the remaining. However, Parallel TSDC has shown to be significantly faster than TSDC across a new, larger allocation problem, which integrated thousands of sub-instances – the UAAP. Some paths for future investigation have been laid out across the section. One consists of investigating ways of minimising the overhead of sorting sub-problems. Another considers conducting research on the impacts of the operating system task scheduling mechanism in the sub-problem sorting approach to adopt. In addition to these, it would be interesting to try parallelisation frameworks other than OpenMP, namely ones that can guarantee sub-problem order of execution; it would be interesting to compare the results with the ones presented in this section. Finally, it would also be stimulating to evaluate the performance of Parallel TSDC with processors with more cores and/or that make use of Hyper-threading. Hyper-threading is an Intel technology (Intel Corporation, 2003), where a single physical processor core is presented as two logical processors to the operating system. As a result, the operating system can schedule two threads to the single physical core. Some components of the processor are actually duplicated, but the execution resources are shared between the two logical processors. The two together can make a more efficient use of those shared resources; when one logical processor is stalled, the other can make use of the execution resources, which would otherwise be idle if only one thread was being executed. Intel claims Hyper-threading can reach performance gains up to 30%.

5.4 A more refined cost optimisation?

Up to this point in the thesis, a cost model has been used which assumed that ASIL imposed costs vary equally across all the components of a system architecture. As practitioners develop more projects according to ISO 26262, more information on project costs will be available, and when performing ASIL allocation they may want to use a finer granularity in cost estimation. For example, they may want to define categories of components (e.g. sensors), and assign them with a cost weight which is different from the one allocated to another category (e.g. actuators). This section aims at exploring the impacts of a finer granularity in cost definition over the ASIL allocations obtained across a system architecture. It also provides a first test to the performance of Parallel TSDC in dealing with optimisation problems where ASIL imposed costs do not vary equally for each of the allocation variables. Part the work presented in this section has been published in Azevedo *et al*. (2014a).

The one wheel version of the hybrid braking system, HBSM1, was selected for the study performed in this section. All of its 125 non-dominated solutions are known; it is clear that the optimal solution is non-dominated, and therefore, given any cost function, by evaluating the 125 non-dominated solutions the optimal allocation can be identified.

In a first instant, three categories of components have been devised for the architecture of HBSM1. The categories were formulated for illustration, and it is possible that others that are more meaningful can be identified. Furthermore, it has also been considered that within a component, the costs of dealing with omission and value failures are the same. The three categories of components identified can be observed in Table 41.

Table 41 - Components of HBSM1 divided in three categories.

Programmable Electronics	Electronic Low Voltage	Electronic High Voltage
Electronic Pedal	Auxiliary Battery	IWM
WNC	EMB Power Converter	IWM Power Converter
Communications Buses	EMB	Powertrain Battery
-	Braking Pedal	-

Programmable Electronics was assumed to be the least expensive category, and was taken as basis for relative cost definition. The components of the *Electronic Low Voltage* category were estimated to be three times more expensive than the ones in Programmable Electronics. Growing complexity is usually tied to an increase of the risk of defect and consequently the investment in safety measures escalates. A much larger complexity is usually present in the hardware and software components of the high voltage hardware architecture and therefore *Electronic High Voltage* has been allocated with the highest cost jump; its components are assumed to incur in safety activities which are five times more expensive than the ones of Programmable Electronics. It is worth noting that Electronic High Voltage includes the main components of the traction drive system, integrating multiple control units with embedded software.

The non-dominated solutions for the HBSM1 have been evaluated using the Experiential-I cost function, together with the different cost weights defined for the three categories of components. A single optimal solution was identified and it is presented in Table 42, on the left (see next page). The right hand side of the table shows the optimal solution for the HBSM1 when the Experiential-I cost function is used without the cost weights.

Analysis of Table 42 tells that the use of categories of components gives an optimal solution which is different than the one obtained when the categories were not considered. In fact, if all the cost functions presented in this thesis are analysed, this solution only evaluates as optimal for the linear cost heuristic. It is interesting to observe that the high cost weight applied to *Electronic High Voltage* clearly biases the optimal solution towards low ASILs for the failure modes of its components.

Table 42 - HBSM1 optimal ASIL allocations for Experiential-I cost function with and without the cost weights defined for the categories of components.

	Exp-I with categories Cost: 1030	**Exp-I** Cost: 390
Braking Pedal.OFailure1	4	4
Braking Pedal.VFailure1	1	1
Electronic Pedal.OFailure1	2	2
Electronic Pedal.OFailure2	2	2
Electronic Pedal.VFailure1	1	1
Electronic Pedal.VFailure2	0	0
Bus1.OFailure1	2	2
Bus2.OFailure1	2	2
WNC.OFailure1	4	2
WNC.OFailure2	0	2
WNC.VFailure1	1	1
WNC.VFailure2	1	1
Auxiliary Battery.OFailure1	4	2
Auxiliary Battery.VFailure1	1	1
Powertrain Battery.OFailure1	0	2
Powertrain Battery.VFailure1	1	1
EMB Power Converter.OFailure1	4	2
EMB Power Converter.VFailure1	1	1
IWM Power Converter.OFailure1	0	2
IWM Power Converter.VFailure1	1	1
EMB.OFailure1	4	2
EMB.VFailure1	1	1
IWM.OFailure1	0	2
IWM.VFailure1	1	1

Having identified the optimal solution for the problem, it is now time to evaluate if Parallel TSDC can find it. Taking from the conclusions of section 5.3.6, the experiments were carried out with Parallel TSDC utilising the maximum number of threads that can be executed concurrently. The CPU utilised in the experiments of this section is the same used throughout the thesis; it integrates four cores and allows for a maximum of four concurrent threads. Furthermore, no specific order is enforced for the execution of the problem's sub-instances; they are simply solved in the order the problem division algorithm provides them. The optimisation algorithm is tuned with the parameterisation found to perform better across the ASIL allocation benchmark (see Table 43).

Table 43 - Parameterisation adopted for Parallel TSDC

$limit_p$	***$limit_q$***	***$updatePeriod_p$***	***$updatePeriod_q$***
3.5n	0.5n	10	11

In the experiments carried out, Parallel TSDC always found the optimum solution. As with previous experiments, the cumulative time taken for the algorithm to solve the problem 50 times was measured. This was repeated 50 times and the average execution time is presented in Table 44 (centre column). The rightmost column of the table shows the execution time of Parallel TSDC in tackling HBSM1-Exp-I without categories of components.

Table 44 - Execution times of Parallel TSDC to complete HBSM1, when allocations were evaluated using the Experiential-I cost heuristic with and without categories of components (data is based on average times from 50 runs).

Cost Function	Exp-I with categories	Exp-I
Time (seconds)	0.00229932	0.00184298

Parallel TSDC was able to complete the HBSM1 with categories of components in every of its runs and within very acceptable times.

As a further experiment, a limit situation for the granularity of costs definition was considered, where the system designer defines different ASIL cost functions for each of the failure modes of an architecture. This scenario was simulated by producing random ASIL cost functions for each failure mode in HBSM1. As it does not impose safety-related activities, ASIL QM was always assigned with cost zero across the different FMs. Four different random costs, from 1 to 250 units, were generated for each failure mode.

These values were associated with the different ASILs - higher costs were linked to the higher ASILs. The generated random cost functions can be consulted in Appendix II of the thesis.

According to the random costs produced, a single optimal solution was identified from the evaluation of the 125 non-dominated solutions of HBSM1 (see Table 45).

Table 45 - HBSM1 Optimal ASIL allocations for Experiential-I cost function with random cost heuristics generated for each FM.

	Random Costs Cost: 1650
Braking Pedal.OFailure1	4
Braking Pedal.VFailure1	1
Electronic Pedal.OFailure1	2
Electronic Pedal.OFailure2	2
Electronic Pedal.VFailure1	1
Electronic Pedal.VFailure2	0
Bus1.OFailure1	4
Bus2.OFailure1	0
WNC.OFailure1	2
WNC.OFailure2	2
WNC.VFailure1	1
WNC.VFailure2	1
Auxiliary Battery.OFailure1	2
Auxiliary Battery.VFailure1	1
Powertrain Battery.OFailure1	2
Powertrain Battery.VFailure1	1
EMB Power Converter.OFailure1	2
EMB Power Converter.VFailure1	1
IWM Power Converter.OFailure1	2
IWM Power Converter.VFailure1	1
EMB.OFailure1	2
EMB.VFailure1	1
IWM.OFailure1	2
IWM.VFailure1	1

The optimal solution obtained is different from the ones found in the previous experiment of this section. As with the best allocation obtained when categories of components are defined, if the cost of the current solution is calculated using the original cost model, it only evaluates as optimal for the linear cost function. With a known optimal solution, experiments can now be carried out to evaluate the performance of Parallel TSDC. The same experimental setup used in the trials with categories of components was followed. Results are presented in Table 46.

Table 46 - Execution times of Parallel TSDC to complete HBSM1, when allocations were evaluated according to random ASIL cost heuristics; for each FM, cost heuristics are different (data is based on average times from 50 runs).

Cost Function	Random
Time (seconds)	0.001971800

For the scenario where cost definition is taken to its finer granularity, Parallel TSDC was again able to always find the optimal solution and in very acceptable times.

The current section has shown that making use of a more granular cost definition results in obtaining optimal solutions for the HBSM1, which are, for most situations, different from the ones obtained in the experiments with the original cost model. This meets the conclusions previously introduced in section 4.3 about the need from practitioners to work towards obtaining more precise cost information. If more precise data is used to inform optimisation approaches, such as the one introduced in this section, more accurate cost-efficient solutions can be identified. Parallel TSDC has shown some robustness in dealing with more refined costs models; it always found the optimal solution for HBSM1, both in the scenario where categories of components were utilised, and also when ASIL cost functions were defined for each FM.

5.5 Results summary

The chapter started by demonstrating that an approach purely based on generating random allocations yields poor results. Four problems were tackled with such an approach, and for two of them, after a day of processing time, no feasible solutions were produced. As for the remaining two problems, the best allocations found were 24% and 78% more expensive than the optimal ones. In finding promising solutions for complex systems more advanced optimisation approaches need to be considered.

The investigation conducted in chapter 3 resulted in three effective metaheuristic optimisation techniques for the ASIL allocation problem. They are based on GA, PSO and TS concepts. The three techniques successfully solve to optimality every problem defined in the ASIL allocation benchmark, and within very satisfactory times too. The longest an algorithm took to solve one problem was ≈60.04 seconds. The algorithms include randomness in their search processes, and their performances vary between runs. The maximum time of 60.04 seconds was obtained in the poorest trial. Across the three techniques, the maximum average time spent in solving one of the allocation instances actually amounts to ≈3.37 seconds (over 50 runs).

TS has been established as the most efficient of the three techniques. The method presented smaller average processing times for every problem in the benchmark. In fact, across the majority of the tests, TS's average execution times are at least one order of magnitude smaller than the ones of GA and PSO. Furthermore, TS took on average ≈0.33 seconds to complete the entire benchmark; this time is ≈21 and ≈27 times faster than the ones presented by PSO and GA, respectively.

Comparing PSO and GA, the first takes the lead. Not only is PSO faster to complete the benchmark, but it also presented smaller average processing times than GA for 14 out of the 16 problems defined.

A divide and conquer methodology to tackle ASIL allocation problems was proposed as means to possibly obtain further improvements to the efficiency of the optimisation process. The methodology consists of dividing a problem into multiple sub-problems of smaller dimensionality, and then solving those independently to provide a solution to the original problem. The problem division tactic proposed was shown to have applicability to the case studies investigated in this thesis.

An algorithm was created combining the divide and conquer concept with the most efficient optimisation technique, TS. The algorithm, coined Tabu Search Divide and Conquer (or TSDC), was tested against the TS version that did not consider problem division. The two approaches were run with the same large number of parameterisations. In its best run, TSDC completed the benchmark in ≈0.10 seconds; the best time obtained by TS is approximately three times slower.

TSDC did not just overcome TS's best performance with the use of a single parameterisation. The technique obtained better results than the best TS using a variety of 424 parameterisations. These represent approximately one fifth of the entire set of parameterisations defined. Furthermore, across 73 parameterisations, TSDC was able to complete the benchmark more than two times faster than the best TS.

In the light of its superiority over TS, TSDC was selected for further analysis. To inform parameter selection in future uses of the approach, research was conducted to identify the parameterisations enabling TSDC's most promising performances across the benchmark. TSDC's best results were found when *(updatePeriod$_p$; updatePeriod$_q$)* were set to (3; 4) and (10; 11). In achieving these most promising performances, (10; 11) allowed for more pairs of values to be selected for *limit$_p$* and *limit$_q$*. A particular set of promising parameterisations associated with (10; 11) had *limit$_p$* varying from 3n to 4n and *limit$_q$* between 0 and 2.5n. Finally, the best parameterisation found for TSDC is given next.

- *(updatePeriod$_p$; updatePeriod$_q$)* = (10; 11)
- *limit$_p$* = 3.5n;
- *limit$_q$* = 0.5n

A subsequent experiment demonstrated that if TSDC's parameters were tailored on a sub-problem basis, the technique's performance could be improved even further. Each sub-problem was executed faster; across the benchmark, TSDC's total execution time was improved by ≈18%.

The hypothesis of programmatically and automatically setting TSDC's parameterisations as a function of problem characteristics was advanced. Obtaining such an approach would not just enable finding faster performances with the use of TSDC; if the method can be generically applied to any problem, it would eliminate any need for parameterisation experimentation from the user. Analysis of the data obtained from applying TSDC to the ASIL allocation benchmark did not allow drawing definitive conclusions on this hypothesis. Nevertheless, some trends were observed, and it has been theorised that a method based on analysis of problem dimensionality could produce parameterisations giving efficient TSDC performances. The method defined that *(updatePeriod$_p$; updatePeriod$_q$)* should grow with dimensionality; in tackling sub-problems with dimensionality smaller than 26, *(updatePeriod$_p$; updatePeriod$_q$)* should be set to (1; 2); sub-problems presenting dimensionalities between 26 and 84 should use (3; 4). No data with problems of dimensionality higher than 84 was available, but it is possible that for those problems higher update periods are required. In the approach proposed, *limit$_q$* is to be set to 0. In regards to *limit$_p$*, it should grow with problem dimensionality following an increasing change rate. The approach yielded good results across the benchmark, but more work is needed to validate its general applicability. Furthermore, it is likely that it can be improved from analysis of more sub-problems. Analysis of the data produced across the benchmark also suggested that in achieving the best possible parameterisations for each sub-instance, additional problem characteristics need to be investigated. Two constraints-related characteristics that may be worth investigating are AACs size and AACs ASIL requirement.

In a final effort to improve further the performance of the optimisation process, an initial method has been proposed for the parallelisation of sub-problems conquering. The method was integrated with TSDC and Parallel TSDC was born. Parallel TSDC gave additional performance benefits across the benchmark; it improved the execution time of the non-parallelised version of the algorithm by ≈1.67%. Parallelisation only allows to reduce time spent in optimisation tasks, i.e., it does not have an impact on the time spent in the problem division step. If only optimisation tasks are considered, the improvements obtained through parallelisation represent a reduction in time of 2.44%. It should be noted that due to the characteristics of the benchmark, these improvements were limited from the start to a maximum of ≈2.89%. When a new, larger problem, with thousands of sub-instances was tackled, the improvements to the time spent in optimisation tasks were far more substantial; times obtained with the parallelisation approach were close to dividing the sequential execution time by the number of threads utilised.

Experiments with the Parallel TSDC approach have given indications that the technique should make use of the maximum number of threads that can be executed concurrently. In regards to the sub-problem sorting mode, if only optimisation tasks are taken in to account, Complex First gives the most efficient results. However, the overhead of sub-problem sorting associated with Complex First makes it a less promising approach than just solving the sub-instances in the order they are provided by the problem division step.

Parallel TSDC allows solving every problem defined in the thesis to optimality; furthermore, it is the technique providing the most efficient performances. In this way, it should be regarded as the ultimate product from this thesis towards the cost efficient allocation of ASILs in complex systems.

The final experimental work from the thesis is presented in Section 5.4. Up to that point, a cost model had been assumed where ASIL dependent costs vary in the same way across the different variables in a problem. In Section 5.4, the HBSM1 case study was taken and two scenarios were considered for a more granular cost definition: in the first, different cost weights were assigned to different categories of components; in a second case, different ASIL cost functions were formulated for each allocation variable. In these two scenarios, the solutions regarded as optimal are different. Furthermore, they are also different from when the original cost model was used with the experiential-I, experiential-II and logarithmic heuristics. Parallel TSDC was applied to the HBSM1 in the presence of the two new cost definition scenarios; in both cases the technique was able to find the optimal solutions very efficiently.

6. Conclusions

6.1 Contributions

The following hypothesis was stated in the introduction chapter:

The allocation of Automotive Safety Integrity Levels in large and/or complex architectures can be efficiently supported through automated model-based safety analysis and the optimisation of development costs with metaheuristics.

Efficient model-based safety analysis techniques were already available at the start of this project (e.g. HiP-HOPS). Therefore, research focused on the development of suitable metaheuristic optimisation techniques. A series of objectives were defined, which if met would support the stated hypothesis. They are revisited now, together with a discussion on how they have been met.

1. **Investigate, amongst the state-of-the-art in metaheuristic optimisation, a set of techniques that can be applied to solve the novel problem of scalable allocation of ASILs.**

Over the years, various authors have created a multitude of metaheuristics, which have been applied with varying success to different problems. Three of the most popular metaheuristics were investigated in the context of this objective: Genetic Algorithms (GA), Particle Swarm Optimiser (PSO) and Tabu Search (TS).

Metaheuristics are generic search guiding strategies and need to be adapted to the specifics of the problem at hand. In reality, their performance greatly depends on the suitability of this adaptation. Successful applications of the three selected metaheuristics were analysed. The study focused on problems with similar characteristics to ASIL allocation, namely within the reliability engineering domain. Together with this literature review examination, problem specific search operators and infeasible search space exploration strategies were proposed. Also, hybrid tactics to combine two metaheuristics were investigated.

Several optimisation techniques were developed from this work. However, three of them have shown particularly strong performances. These three techniques are all based on different metaheuristics, i.e., there is one method based on GA, another on PSO and finally one on TS. On the basis of their performances, these techniques were compared in the context of objective 3.

2. **Establish a set of tests and metrics that allow comparison of the performance of the metaheuristic techniques being investigated.**

Four different system case studies were created in this thesis. Their models and dependability properties allowed the generation of ASIL allocation problems with a diversity of characteristics. For example, the problem with the smallest dimensionality counted 24 variables, whereas the largest counted 185. The latter (named BSS), generates an extraordinary total search space of $\approx 2.04 \times 10^{219}$ solutions! AACs characteristics also varied across the different problems. For instance, HBSM1 counted 31 AACs, whereas HBSM3 counted 11,573. The maximum size of AACs in HBSM1 was 2, whereas in HBSM3 there were AACs of sizes up to 8. The four case studies were combined with four different ASIL cost functions, formulating a benchmark counting 16 problems in total.

The optimisation techniques developed include randomness in their search process. If they are run two times, they might find solutions of different qualities or take a different number of iterations to find a given allocation. In this way, it was decided that performance would be compared on the basis of running an algorithm 50 times. Initially, two evaluation metrics were defined: 1) the number of runs returning the optimal solution for a problem; 2) the relative cost increase against the optimal cost for a problem. However, it was verified that the three techniques developed in the context of objective 1 were able to solve each of the 16 allocation problems in every run, and within satisfactory time. At this point, the goal of the comparison became the identification of the most efficient technique. Since the concept of iteration is different across the algorithms, and that within the same algorithm iterations can pose varying efforts, a comparison based on iteration count was set aside. It was decided, instead, that the algorithm performance would be evaluated in terms of execution time to find the optimal solution.

3. **Select the best performing metaheuristic and examine the possibility of improving it further and enhance its usability.**

The comparison study between the three techniques created to tackle ASIL allocation yielded TS as the most promising. For every problem in the benchmark, TS was always the fastest technique to find an optimal solution. The technique took, on average, ≈0.33 seconds to solve the entire benchmark, being ≈21 times faster than PSO, and ≈27 times faster than GA. Together with performance benefits, TS brings the advantage of a better tuning experience. There are only four parameters to select, whereas PSO includes 13 and GA integrates 10.

For the problems in the benchmark, TS was already capable of providing high quality solutions within time spans that clearly supported the iterative nature of ASIL allocation. However, targeting scalability to larger and more complex problems, the technique was examined for further performance improvements. The enhancements considered were based on a methodology which contemplated the *division* of an original problem into multiple sub-instances of smaller dimensionalities. In turn, these sub-problems could be solved (or *conquered*) independently to provide a solution for the original problem. The TS technique was combined with the *divide and conquer* approach, creating a method called Tabu Search Divide and Conquer (TSDC). TSDC was compared with TS across a vast number of parameterisations and the best performance obtained by TSDC was almost three times faster than TS's best. Further analysis revealed that TSDC was faster than the best TS using many parameterisations (424 parameterisations, i.e. approximately one fifth of all parameterisations used). The total time needed by TSDC to complete the benchmark amounted, on average, to ≈0.10 seconds. Given that the sub-problems obtained in the division step of TSDC are completely independent, an initial approach for parallelisation was evaluated. Parallel TSDC improved the benchmark's completion time further by 1.67% (less than 2 milliseconds). Parallelisation is only used during the conquering step of the approach, i.e., it does not impact on the time taken for problem division. If only optimisation tasks are considered, the use of Parallel TSDC represents reducing execution times by 2.44%. The relatively small improvement can be explained by the characteristics of the benchmark. Generally, for each problem in the benchmark, there was one sub-instance that took much longer to solve than the remaining. When parallelisation was used, the time to solve each problem had a minimum bound given by the execution time of the longer sub-instances. Analysis of the execution times of the different sub-problems revealed that possible enhancements via parallelisation were limited to 2.89% from the start.

An Ultimate ASIL Allocation Problem (UAAP) was created to evaluate the possible benefits of using Parallel TSDC when tackling other, larger problems. The UAAP integrates every problem of the benchmark 50 times. In total, it counts 550 sub-instances of size greater than one and 7150 of size one. When Parallel TSDC was applied to the UAAP, it obtained performances that are very close to dividing the execution time of TSDC by the number of threads utilised. For reference, Parallel TSDC completed this huge problem in ≈0.4 seconds (using four threads in a processor that could handle up to four threads concurrently). Parallel TSDC performed better when:

1) Sub-problems were solved in a random order (i.e. accepting the order in which the problem division step produces them)
2) The maximum number of threads that could be solved concurrently was used

To enhance the usability of the optimisation method, two studies were conducted. One evaluated the possibility of utilising a same parameterisation for every problem tackled. A large number of parameterisations were evaluated, and the one obtaining best results across the benchmark was:

- $(updatePeriod_p; updatePeriod_q)$ = (10; 11)
- $limit_p$ = 3.5n;
- $limit_q$ = 0.5n

n stands for problem dimensionality.

In a second step, the possibility of programmatically and automatically setting the parameterisations as a function of characteristics of the problem to be solved was also considered. If such a method was available, this would completely eliminate the need for any parameterisation experimentation from the user. Furthermore, performance benefits have been found when parameterisations were optimised on a per sub-problem basis. Whilst this study did not derive any definitive conclusion, a method was theorised on the basis of some observable trends as a function of dimensionality. The method defined that *($updatePeriod_p$; $updatePeriod_q$)* should grow with dimensionality:

- Sub-problems of dimensionalities lower than 26
  - *($updatePeriod_p$; $updatePeriod_q$)* = (1; 2)
- Sub-problems of dimensionalities between 26 and 84
  - *($updatePeriod_p$; $updatePeriod_q$)* = (3; 4)

In regards to $limit_p$, it should grow with problem dimensionality following an increasing change rate. As for $limit_q$ it should be set to 0. There is room for more investigation in this topic, and this will be discussed in chapter 7.

4. **Apply the developed approach to case studies in order to validate its applicability and usefulness to the development of complex engineering systems.**

The optimisation methods developed were applied to a series of problems, which were built on top of four case studies. Three of these case studies were variants in complexity of a hybrid braking system for electrical vehicles. This system has been defined in the context of my Master's (Azevedo, 2012). The hybrid braking strategy is based on the work of de Castro *et al* (2011), and the system's structure is inspired by, for example, the works of Isermann *et al*. (2002) and Isermann (2002). The fourth case study, BSS, is a very large system of systems, combining braking and steering-by-wire features, and was provided by an industrial partner of HiP-HOPS. The final product of this thesis, Parallel TSDC, was able to find optimal solutions for all problems defined from the case studies, within very acceptable time spans. The technique was also successfully applied to an even larger problem, the UAAP. Although this problem does not necessarily represent a realistic case study, it further indicates promise to the approach.

The hybrid braking system was used in chapter 4 to demonstrate the usefulness of the automated ASIL allocation approach proposed in the thesis. The example illustrated the use of HiP-HOPS model-based approach to capture the system failure behaviour and obtain the all the information necessary to formulate ASIL allocation constraints. Then, in the face of a number of solutions that is clearly outside of realms of manual evaluation ($\approx 5 \times 10^{16}$), the optimisation algorithms were able to quickly retrieve ASIL algebra compliant and cost-optimal solutions. The example has also shown how the automated and efficient nature of the approach can facilitate negotiations with suppliers.

This thesis argued that ASIL allocation could be efficiently and effectively supported with a mixture of model-based safety analysis and metaheuristic optimisation algorithms. Whereas research focused on the optimisation techniques, it is also important to comment on the HiP-HOPS FTA performance. In his PhD thesis, Parker (2010) describes a series of enhancements to HiP-HOPS fault tree analysis and synthesis algorithms; the enhancements allowed for very promising performance when tackling large problems. HiP-HOPS has since then been equipped with these strongly performing algorithms. For every case study developed in this thesis, the tool returns FTA results within sub-second time frames. Parallel TSDC adds less than 0.1 seconds to the execution time of solving any of the problems coming from these case studies. It seems reasonable to assume that allocating a few hours to perform ASIL allocation does not put at risk the need to carry it out in the context of an iterative design process. The margin between sub-second time frames and hours leaves encouraging prospects for the use of HiP-HOPS and Parallel TSDC across even larger and more complex systems.

While HiP-HOPS has been used in the thesis to provide the FTA information required to formulate allocation constraints, it is important to highlight that Parallel TSDC can easily link with any other model-based FTA tools. In fact, it can work on top of any method that provides both:

a) The minimal cut sets for a system
b) The total ASILs that the failure modes of each minimal cut set must meet

5. Explore the impacts of different cost heuristics in the allocation of ASILs.

This objective was addressed using the pre-existing HiP-HOPS ASIL allocation algorithm. The allocation algorithm is of an exhaustive nature, meaning that when it is capable of finishing a problem, it will find its optimal solutions. Although the algorithm does not scale well, it is capable of completing the smallest case study defined in the thesis - HBSM1.

In chapter 4, the optimal solutions for the HBSM1 according to two illustrative ASIL cost heuristics were compared. The first observation was that each cost heuristic had a different number of optimal solutions associated with them – there was only one optimal solution according to one cost heuristic, whereas the other revealed four optimal solutions. The second observation is that the single optimal solution for the first cost heuristic was different from all four optimal solutions from the second.

Towards the end of chapter 5, a study with more refined cost models was conducted. Different cost weights were defined for categories of components and in a final step different, random, cost functions were defined for each component. The solutions obtained in these two steps were different between themselves, and different as well from the solutions obtained in the experiment of chapter 4.

The takeaway message here is that the use of different ASIL cost information can lead to very different allocations to be seen as optimal. This gives strength to the necessity for work to be undertaken in industry in order to obtain more accurate ASIL cost information. Approaches to inform decision making, such as the one proposed in this thesis, need this more precise data in order to find solutions which are closer to the real optimal allocations.

Whilst the above remains a topic to be addressed in the industrial context, the final product of this thesis has shown flexibility to effectively and efficiently find optimal solutions in the presence of multiple cost heuristics and cost models. Different system designers can in this way use the approach according to the best information that is available to them.

Range of applicability of the approach

The approach developed in this thesis is not constrained to any sub-domain of automotive safety-related electronic systems – it can be generically used to solve problems where the ASIL decomposition algebra applies.

Whereas the approach's implementation has been optimised to deal with ASILs, the concept is transferrable to other industry sectors where problems can be modelled using equivalent decomposition algebras. For the aerospace industry, the concept has already achieved some validation through the work from Sorokos *et al*. (2015a, 2015b). Influenced by the investigation from this thesis, the authors have developed a technique combining model-based safety analysis and Tabu Search. The implementation of the Tabu Search technique has been defined to deal with the allocation algebra from the aerospace industry. Case studies have been analysed with promising results.

6.2 Impact on external research

The thesis project is now drawing to an end, but it has already impacted external research. The Tabu Search technique used in the comparison with GA and PSO has been included in MAENAD (Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles). MAENAD was an European FP7 Project (MAENAD, 2015) that finished in the beginning of 2015. The project aimed at extending EAST-ADL (EAST-ADL Association, 2015), an automotive Architecture Description Language, and improving its tool support in order to address the new challenges of electrical vehicle development.

The same optimisation technique used in MAENAD has also been integrated in the research of André Oliveira, from the University of York, to address the ASIL allocation problem in the context of product line engineering. In product lines, multiple products are created following a strategy to maximise the number of components reused. Components are shared across different products, but different products originate different hazards with different risks and ASILs. It becomes important to find an allocation that simultaneously fulfils the ASIL requirements defined for each product - ideally one that imposes minimum costs. The results of this work have been published in de Oliveira *et al*. (2015).

As mentioned in the previous section, the research from this thesis has influenced work for the use of model-based techniques and metaheuristics in the allocation of SILs within the aerospace domain. This work is being carried out as part of Ioannis Sorokos PhD, here at the University of Hull. Research has already been outputted in Sorokos *et al*. (2015a, 2015b).

Towards the end of this project, a collaboration has been established with the Generative Software Development Lab from the University of Waterloo in Canada. The work consisted of investigating the feasibility of using off-the-shelf exact solvers to find all cost-optimal solutions for a series of ASIL allocation instances. Different approaches to formulate the ASIL allocation problem have been proposed and experiments were carried out with three versions of the hybrid braking system and four different cost heuristics - providing a total of 12 problem instances. For most cases, an Integer Linear Programming (ILP) and a Satisfiability Modulo Theories (SMT) solver, Z3 SMT (Microsoft Research, 2015) and CPLEX ILP (IBM, 2015) respectively, presented very efficient processing times; however, there were cases where several hours or even multiple days were needed to complete the optimisation process. The results of this work have been published in Murashkin *et al*. (2015).

Finally, Youcef Gheraibia, a PhD candidate from the University of Mohammed Cherif Messaadiahas (Algeria), has traveled to Hull to apply a metaheuristic of his authorship to the ASIL allocation optimisation problem: Penguins Search Optimisation Algorithm (Gheraibia & Moussaoui, 2013). This new nature-inspired metaheuristic is based on the collaborative hunting strategies of penguins. A paper based on the research has been accepted for publication on the IEEE Seventh International Conference on Intelligent Computing and Information Systems (Gheraibia *et al.*, 2015).

6.3 A note on the relationship with the ISO 26262 standard

As stated in the introduction chapter, this thesis addresses the generic problem of "safety integrity requirements allocation". Whereas the automotive industry was taken as an application example, specifically its ASIL decomposition algebra, the product of this thesis is not an attempt to fully implement ISO 262626 guidelines for ASIL allocation. For example, in the standard, ASILs are first allocated to safety requirements; then, the safety requirement and respective ASIL are allocated to components. This represents a different way of reasoning than in most safety standards. However, Blanquart *et al.* (2012) point out that there is no evidence that this approach leads to different allocations since the process still relies on failure propagation and end effects. In the method adopted in the thesis, the process of ASIL allocation starts from the point where a system architecture and respective safety-related functionality is already defined. ASILs are allocated and decomposed as a function of the dependability characteristics of the system. Another difference is that the standard does not allow for decomposition when redundant components are susceptible to cascading failures, i.e., in scenarios where errors of components elsewhere in the system cause the redundant elements to fail. Through FTA, the approach proposed here pinpoints what are the root causes of system failures, and assigns adequate integrity requirements to those root causes. If by failing, a component originates a system hazard posing the highest level of risk, it will be assigned with the highest integrity requirements. The architectural elements that receive inputs from this component will be assigned with integrity requirements that reflect solely their own impact in originating a system hazard.

The first edition of ISO 26262 was released in late 2011. As a result of different understanding and different needs, practitioners have been implementing it in different ways. A revised draft of the standard is being prepared. Possibly, the method proposed in this thesis can influence the evolution of the standard concerning ASIL allocation guidelines.

7. Limitations and suggestions for future work

This thesis has investigated automatic metaheuristic optimisation techniques, which combined with model-based safety analysis, provide effective and efficient support to ASIL allocation. There is room for further research to extend this work, and this chapter identifies limitations, areas for improvements and relevant paths for future work.

1) Identifying decomposition preferences and patterns

The optimisation approach developed in this thesis explores all decomposition options allowed by the SIL algebra. However, from a practical point of view, it can make sense to constrain the solution space further. The ISO 26262 standard says that when decomposition is to be applied to a component which is monitored by a safety mechanism, the safety mechanism should be allocated with a higher ASIL than the component it supervises (Int'l Organization for Standardization, 2011:Part 9). The rationale given is that safety mechanisms are usually less complex and are easier to develop to higher ASILs. Future work could consider the modelling of this, and possibly other, architectural patterns, and add extra constraints to the optimisation process.

2) Allocating ASILs to components

In the method proposed in the thesis, ASILs are allocated to components' FMs and the cost of a solution is evaluated on the basis of the ASILs allocated to each FM of a system. The approach provides means for a more refined allocation, which can be more cost-efficient over the course of a hierarchical allocation process. However, it might not go in meeting current industrial practice: ISO 26262 prescribes a process where ASILs are allocated to safety requirements and subsequently to components; also, it might be difficult to tailor costs as a function of the different FMs of a component. This is a limitation of the work presented in this thesis. A solution has been identified and it can be implemented as future work:

a) Variables are defined as the ASILs to allocate to each component of a system. As a result, ASIL dependent costs must be defined on a per component basis
b) From the MCSs of a system, the constraints associated with the FMs of a component must all be associated with the component itself

This approach would potentially result in a decrease of the search space to explore. The dimensionality of the problems to solve would often decrease (multiple variables would be merged into a single one – i.e. instead of allocating multiple ASILs to the different FMs of a component, it is only necessary to allocate an ASIL to the component), and there would be an increase to the number of constraints for each variable (the constraints associated with each FM of a component are all associated with the component). The algorithms formulated from this thesis would still be able to accept these problems as inputs, since variables and constraints remain of the same type.

3) Validating and enhancing the automatic parameterisation of Parallel TSDC

Discovering a method to automatically parameterise Parallel TSDC is important, as it can potentially lead to further performance benefits. Also, it eliminates the need for any parameterisation experimentation from the user. A method based on problem dimensionality has been presented in section 5.3.5. Whilst the approach obtained good results across the benchmark, further work needs to be conducted to validate it and possibly improve it. Furthermore, investigation hinted that other problem characteristics might need to be investigated in order to obtain the best performance. Two possible characteristics worth investigating are AACs size and AACs ASIL requirements. The automatic parameterisation of Parallel TSDC is not a trivial problem and will likely require the study of a wide number of problems.

4) Investigating ways of improving the parallelisation of TSDC

An initial effort has been presented in section 5.3.6 towards the parallelisation of TSDC. Promising performance benefits have been obtained, however there is room for further research. One path consists of investigating ways of minimising the sub-problem sorting overhead. It would also be interesting to understand the impacts of the operating system task scheduling mechanisms over which sub-problem sorting approach to adopt. OpenMP has been used as the framework to implement the parallelisation of TSDC. It could be beneficial to evaluate other frameworks that can guarantee the sub-problem order of execution. Finally, it could be useful to evaluate the performance of Parallel TSDC with processors with more cores and/or that make use of Hyper-threading technology.

5) Efficiency and near optimality

The optimisation techniques developed in this project are based on metaheuristics. The final product of the thesis, Parallel TSDC, was able to quickly find optimal solutions for problems with close to 200 variables and thousands of constraints, suggesting good scalability. There is, however, a limitation inherent to the use of metaheuristics optimisation techniques. When applying them to a new problem, there is no guarantee that optimal solutions will be retrieved. This is a limitation that can only be overcome with the use of exact techniques. Whereas for large instances it can be infeasible to apply exact techniques, a methodology could be developed to arrive at a solution that is, at least, guaranteed to be partially optimal. This methodology could be integrated with the divide and conquer approach from this thesis. Smaller interconnected instances could be solved with exact techniques, whereas very complex instances would be tackled with the efficient metaheuristic optimisation techniques proposed here. A first step towards such a methodology was already incorporated in the work from this thesis - interconnected instances of size 1 are solved to optimality with the Direct Allocation Algorithm.

Bibliography

Adachi, M., Papadopoulos, Y., Sharvia, S., Parker, D. & Tohdo, T. (2011) An approach to optimization of fault tolerant architectures using HiP-HOPS. *Software: Practice and Experience*, 41(11), 1 October, 1303–1327.

Affenzeller, M., Beham, A., Kofler, M., Kronberger, G., Wagner, S.A. & Winkler, S. (2009) Metaheuristic optimization. In Buchberger, B., Affenzeller, M., Ferscha, A., Haller, M., Jebelean, T., Klement, E.P., Paule, P., Pomberger, G., Schreiner, W., Stubenrauch, R., Wagner, R., Weiß, G. & Windsteiger, W. (eds) *Hagenberg Research*. Berlin: Springer, 103–155.

Allen, M. (2012) *Cost versus ASIL*. Available online: https://www.linkedin.com/grp/post/2308567-92692199 [Accessed 11/1/2013].

Artois Univeristy & CNRS (2015) *Sat4j - the boolean satisfaction and optimization library in Java*. Available online: http://www.sat4j.org/ [Accessed 17/2/2013].

Azevedo, L.S. (2012) *Hybrid braking system for electrical vehicles: functional safety*. MSc thesis. Faculdade de Engenharia da Universidade do Porto.

Azevedo, L.S., Parker, D., Papadopoulos, Y., Walker, M., Sorokos, I. & Araújo, R.E. (2014a) Exploring the impact of different cost heuristics in the allocation of safety integrity levels. In Ortmeier, F. & Rauzy, A. (eds) *Model-Based Safety and Assessment*. Lecture Notes in Computer Science 8822. Cham: Springer International Publishing, 70–81.

Azevedo, L.S., Parker, D., Walker, M., Papadopoulos, Y. & Araùjo, R.E. (2013) Automatic decomposition of safety integrity levels: optimization by tabu search. *The 2nd Workshop* *on Critical Automotive Applications: Robustness & Safety (CARS 2013) of the 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013)*. Toulouse, September 2013.

Azevedo, L.S., Parker, D., Walker, M., Papadopoulos, Y. & Esteves Araujo, R. (2014b) Assisted assignment of automotive safety requirements. *IEEE Software*, 31(1), January, 62–68.

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, May 2010.

Bieber, P., Bougnol, C., Castel, C., Kehren, J.-P.H.C., Metge, S. & Seguin, C. (2004) Safety assessment with Altarica. In Jacquart, R. (ed) *Building the Information Society*. International Federation for Information Processing 156. New York: Springer US, 505–510.

Bieber, P., Delmas, R. & Seguin, C. (2011) DALculus – theory and tool for development assurance level allocation. In Flammini, F., Bologna, S. & Vittorini, V. (eds) *Computer Safety, Reliability, and Security*. Lecture Notes in Computer Science 6894. Berlin: Springer, 43–56.

Blanquart, J.-P., Astruc, J.-M., Baufreton, P., Boulanger, J.-L., Delseny, H., Gassino, J., Ladier, G., Ledinot, E., Leeman, M., Machrouh, J., Quéré, P. & Ricque, B. (2012) Criticality categories across safety standards in different domains. *Embedded Real Time Software and Systems (ERTS 2012)*. Toulouse, February 2012, 1–3.

Blum, C. & Li, X. (2008) Swarm intelligence in optimization. In Blum, C. & Merkle, D. (eds) *Swarm Intelligence*. Natural Computing Series. Berlin: Springer, 43–85.

Blum, C., Puchinger, J., Raidl, G.R. & Roli, A. (2011) Hybrid metaheuristics in combinatorial optimization: a survey. *Applied Soft Computing*, 11(6), September, 4135–4151.

Blum, C. & Roli, A. (2003) Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Computing Surveys*, 35(3), September, 268–308.

Boussaïd, I., Lepagnot, J. & Siarry, P. (2013) A survey on optimization metaheuristics. *Information Sciences*, 237, 10 July, 82–117.

Bozzano, M. & Villafiorita, A. (2003) Improving system reliability via model checking: the FSAP/NuSMV-SA safety analysis platform. In Anderson, S., Felici, M. & Littlewood, B. (eds) *Computer Safety, Reliability, and Security*. Lecture Notes in Computer Science 2788. Berlin: Springer, 49–62.

Bozzano, M. & Villafiorita, A. (2010) Introduction. In *Design and safety assessment of critical systems*. Boston: Auerbach Publications.

de Castro, R., Araújo, R.E. & Freitas, D. (2011) Hybrid ABS with electric motor and friction brakes. *The 22nd International Symposium on Dynamics of Vehicles on Roads and Tracks (IAVSD 2011)*. Manchester, August 2011, 1–7.

Charette, R.N. (2009) This car runs on code. *IEEE Spectrum*, 46(3). Available online: http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code [Accessed 18/8/2013].

Chu, S.C. & Fang, H.L. (1999) Genetic algorithms vs. tabu search in timetable scheduling. *The Third International Conference on Knowledge-Based Intelligent Information Engineering Systems (KES 1999)*. Adelaide, December 1999. IEEE, 492–495.

Clerc, M. (1999) The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. *The 1999 Congress on Evolutionary Computation (CEC 99)*. Washington DC, July 1999. IEEE, 1951–1957.

Clerc, M. & Kennedy, J. (2002) The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1), February, 58–73.

Coello Coello, C.A. (2002) Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11–12), 4 January, 1245–1287.

Coit, D.W. & Smith, A.E. (1996a) Penalty guided genetic search for reliability design optimization. *Computers & Industrial Engineering*, 30(4), September, 895–904.

Coit, D.W. & Smith, A.E. (1996b) Reliability optimization of series-parallel systems using a genetic algorithm. *IEEE Transactions on Reliability*, 45(2), June, 254–260.

Dasgupta, S., Papadimitriou, C.H. & Vazirani, U. (2008) Divide and conquer algorithms. In *Algorithms*. New York: McGraw-Hill, Inc.

Dhouibi, M.S., Perquis, J.-M., Saintis, L. & Barreau, M. (2014) Automatic decomposition and allocation of safety integrity level using system of linear equations. *The Fourth*

International Conference on Performance, Safety and Robustness in Complex Systems and Applications (PESARO 2014). Nice, February 2014, 1–5.

Dorigo, M. (1992) *Optimization, learning and natural algorithms*. PhD thesis. Politecnico di Milano. Available online: http://ci.nii.ac.jp/naid/10016599043/ [Accessed 18/8/2015].

EAST-ADL Association (2015) *EAST-ADL*. Available online: http://www.east-adl.info/ [Accessed 14/10/2015].

Eberhart, R. & Kennedy, J. (1995) A new optimizer using particle swarm theory. *The Sixth International Symposium on Human Science (MHS '95)*. Nagoya, October 1995. IEEE, 39–43.

European Commission (2015) *Statistics - accidents data*. Available online: http://ec.europa.eu/transport/road_safety/specialist/statistics/index_en.htm [Accessed 12/1/2014].

European Committee for Eletrotechnical Standardization (1997) *EN 50126: Railway applications - the specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*. Brussels: CENELEC.

European Committee for Eletrotechnical Standardization (2001) *EN 50128: Railway application - communications, signaling and processing systems - software for railway control and protection systems*. Brussels: CENELEC.

European Committee for Eletrotechnical Standardization (2003) *EN 50129: Railway applications - communications, signaling and processing systems - safety related electronic systems for signaling*. Brussels: CENELEC.

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

Geem, Z.W., Kim, J.H. & Loganathan, G.V. (2001) A new heuristic optimization algorithm: harmony search. *Simulation*, 76(2), 60–68.

Gendreau, M. & Potvin, J.-Y. (2005a) Metaheuristics in combinatorial optimization. *Annals of Operations Research*, 140(1), November, 189–213.

Gendreau, M. & Potvin, J.-Y. (2005b) Tabu Search. In Burke, E.K. & Kendall, G. (eds) *Search methodologies: introductory tutorials in optimization and decision support techniques*. New York: Springer US, 165–186.

Gheraibia, Y. & Moussaoui, A. (2013) Penguins Search Optimization Algorithm (PeSOA). In Ali, M., Bosse, T., Hindriks, K.V., Hoogendoorn, M., Jonker, C.M. & Treur, J. (eds) *Recent Trends in Applied Artificial Intelligence*. Lecture Notes in Computer Science 7906. Berlin: Springer, 222–231.

Gheraibia, Y., Moussaoui, A., Azevedo, L.S., Parker, D., Papadopoulos, Y. & Walker, M. (2015) Can aquatic flightless birds allocate automotive safety requirements? *The Seventh International Conference on Intelligent Computing and Information Systems (ICICIS 2015)*. Cairo, December 2015. IEEE [accepted].

Glover, F. (1986) Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5), 533–549.

Glover, F. (1989) Tabu search - part I. *ORSA Journal on Computing*, 1(3), 1 August, 190–206.

Glover, F. (1990) Tabu search - part II. *ORSA Journal on Computing*, 2(1), 1 February, 4–32.

Grunske, L. & Kaiser, B. (2005) An automated dependability analysis method for COTS-based systems. In Franch, X. & Port, D. (eds) *COTS-Based Software Systems*. Lecture Notes in Computer Science 3412. Berlin: Springer, 178–190.

Hansen, P. & Jaumard, B. (1990) Algorithms for the maximum satisfiability problem. *Computing*, 44(4), 279–303.

Hansen, P. & Lih, K.-W. (1996) Heuristic reliability optimization by tabu search. *Annals of Operations Research*, 63(2), April, 321–336.

Holland, J.H. (1975) *Adaption in natural and artificial systems*. Ann Arbor: The University of Michigan Press.

IBM (2015) *CPLEX optimizer*. Available online: http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer [Accessed 11/9/2014].

IEEE (2015) *IEEE Xplore digital library*. Available online: http://ieeexplore.ieee.org/Xplore/home.jsp [Accessed 3/9/2015].

Intel Corporation (2015) *Intel® ARK (product specifications)*. Available online: http://ark.intel.com [Accessed 26/6/2015].

Intel Corporation (2003) *Intel®hyper-threading technology - technical user's guide*. Santa Clara: Intel.

Int'l Electrotechnical Comission (2010) *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety related systems*, 2nd edition. Geneva: IEC.

Int'l Electrotechnical Commission (1998) *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety related systems*. Geneva: IEC.

Int'l Electrotechnical Commission (2003) *IEC 61511: Functional safety - Safety instrumented systems for the process industry sector*. Geneva: IEC.

Int'l Electrotechnical Commission (2011) *IEC 61513: Nuclear power plants - instrumentation and control important to safety - general requirements for systems*. Geneva: IEC.

Int'l Organization for Standardization (2011) *ISO 26262: Road vehicles - functional safety*. Geneva: ISO.

Isermann, R., Schwarz, R. & Stolzl, S. (2002) Fault-tolerant drive-by-wire systems. *IEEE Control Systems*, 22(5), October, 64–81.

Isograph Ltd (2015) *FaultTree+*. Available online: http://www.isograph.com/software/reliability-workbench/fault-tree-analysis/ [Accessed 10/7/2015].

ITI Gmbh (2015) *SimulationX*. Available online: http://www.simulationx.com/ [Accessed 8/6/2015].

Kaiser, B., Liggesmeyer, P. & Mäckel, O. (2003) A new component concept for fault trees. *The 8th Australian Workshop on Safety Critical Systems and Software (SCS 2003)*. Canberra, October 2003. Australian Computer Society, Inc., 37–46.

Kennedy, J. & Eberhart, R. (1995) Particle swarm optimization. *The 1995 International Conference on Neural Networks*. Perth, November 1995. IEEE, 1942–1948.

Kennedy, J., Kennedy, J.F., Eberhart, R.C. & Shi, Y. (2001) *Swarm intelligence*. San Francisco: Morgan Kaufmann.

Kirkpatrick, S., Gelatt, C.D. & Vecchi, M.P. (1983) Optimization by simulated annealing. *Science*, 220(4598), 671–680.

Kulturel-Konak, S., Smith, A.E. & Coit, D.W. (2003) Efficiently solving the redundancy allocation problem using tabu search. *IIE Transactions*, 35(6), 1 June, 515–526.

Kuo, W. (2001) Metaheuristic algorithms for optimization in reliability systems. In *Optimal Reliability Design: Fundamentals and Applications*. Cambridge: Cambridge University Press.

Levitin, G., Hu, X. & Dai, Y.-S. (2007) Particle swarm optimization in reliability engineering. In Levitin, D.G. (ed) *Computational Intelligence in Reliability Engineering*. Studies in Computational Intelligence 40. Berlin: Springer, 83–112.

Li, C., Yang, S. & Korejo, I. (2008) An adaptive mutation operator for particle swarm optimization. *The 2008 UK Workshop on Computational Intelligence (UKCI '08)*. Leicester, September 2008. MIC 2008, 165–170.

Lin, M.-H., Tsai, J.-F. & Yu, C.-S. (2012) A review of deterministic optimization methods in engineering and management. *Mathematical Problems in Engineering*, 2012, 20 June. Available online: http://www.hindawi.com/journals/mpe/2012/756023/abs/ [Accessed 6/2/2013].

Li, X., Cui, J., Qi, J. & Yang, S. (2007) Energy transmission modes based on Tabu search and particle swarm hybrid optimization algorithm. *Journal of Central South University of Technology*, 14(1), January, 144–148.

Mader, R., Armengaud, E., Leitner, A. & Steger, C. (2012) Automatic and optimal allocation of safety integrity levels. *The 2012 Reliability and Maintainability Symposium (RAMS 2012)*. Reno, January 2012. IEEE, 1–6.

MAENAD (2015) *MAENAD*. Available online: http://www.maenad.eu/ [Accessed 15/2/2014].

Mathworks, Inc (2015) *Simulink - simulation and model-based design*. Available online: http://uk.mathworks.com/products/simulink/ [Accessed 8/5/2015].

McDermott, R., Mikulak, R.J. & Beauregard, M. (1996) *The basics of FMEA*, 2nd edition. New York: CRC Press.

Microsoft (2015) *Visual Studio*. Available online: https://www.visualstudio.com/ [Accessed 17/7/2015].

Microsoft Research (2015) *Z3 solver*. Available online: http://z3.codeplex.com [Accessed 16/9/2014].

Murashkin, A., Azevedo, L.S., Guo, J., Zulkoski, E., Liang, J.H., Czarnecki, K. & Parker, D. (2015) Automated decomposition and allocation of automotive safety integrity levels using exact solvers. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 8(1), 14 April. Available online: http://papers.sae.org/2015-01-0156/ [Accessed 30/4/2015].

Nakano, S., Ishigame, A. & Yasuda, K. (2010) Consideration of particle swarm optimization combined with tabu search. *Electrical Engineering in Japan*, 172(4), 1 September, 31–37.

Nakano, S., Ishigame, A. & Yasuda, K. (2007) Particle swarm optimization based on the concept of tabu search. *The 2007 Congress on Evolutionary Computation (CEC 2007)*. Singapore, September 2007. IEEE, 3258–3263.

Nggada, S.H., Parker, D.J. & Papadopoulos, Y.I. (2010) Dynamic effect of perfect preventive maintenance on system reliability and cost using HiP-HOPS. *The 5th Conference on Management and Control of Production and Logistics (IFAC-MCPL 2010)*. Coimbra, September 2010. IFAC, 204–209.

de Oliveira, A.L., Papadopoulos, Y., Azevedo, L.S., Parker, D., Braga, R.T.V., Masiero, P.C., Habli, I. & Kelly, T. (2015) Automatic allocation of safety requirements to components of a software product line. *IFAC-PapersOnLine*, 48(21). Available online: http://www.sciencedirect.com/science/article/pii/S2405896315018352 [Accessed 15/11/2015].

OpenMP ARB (2015) *The OpenMP® API specification for parallel programming*. Available online: http://openmp.org/wp/ [Accessed 17/5/2015].

Osman, I.H. & Laporte, G. (1996) Metaheuristics: a bibliography. *Annals of operations research*, 63(5), 511–623.

Pande, P.K., Spector, M.E. & Chatterjee, P. (1975) *Computerized fault tree analysis: TREEL and MICSUP*. Berkeley: University Of California Operations Research Centre.

Papadimitriou, C.H. & Steiglitz, K. (1982) *Combinatorial optimization: algorithms and complexity*. Upper Saddle River: Prentice-Hall, Inc.

Papadopoulos, Y. & McDermid, J.A. (1999) Hierarchically performed hazard origin and propagation studies. In Felici, M. & Kanoun, K. (eds) *Computer Safety, Reliability and Security*. Lecture Notes in Computer Science 1698. Berlin: Springer, 139–152.

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), March, 590–608.

Papadopoulos, Y., Walker, M., Reiser, M.-O., Weber, M., Chen, D., Törngren, M., Servat, D., Abele, A., Stappert, F., Lonn, H. & others (2010) Automatic allocation of safety integrity levels. *The 1st Workshop on Critical Automotive Applications: Robustness & Safety (CARS 2010)*. Valencia, April 2010. ACM, 7–10.

Parker, D.J. (2010) *Multi-objective optimisation of safety-critical hierarchical systems*. PhD thesis. The University of Hull. Available online: http://hydra.hull.ac.uk/resources/hull:3465 [Accessed 5/1/2013].

Parker, D., Walker, M., Azevedo, L.S., Papadopoulos, Y. & Araújo, R.E. (2013) Automatic decomposition and allocation of safety integrity levels using a penalty-based genetic algorithm. In Ali, M., Bosse, T., Hindriks, K.V., Hoogendoorn, M., Jonker, C.M. & Treur, J. (eds) *Recent Trends in Applied Artificial Intelligence*. Lecture Notes in Computer Science 7906. Berlin: Springer, 449–459.

Poli, R. (2008) Analysis of the publications on the applications of particle swarm optimisation. *Journal of Artificial Evolution and Applications*, 2008, 1–10.

Poli, R., Kennedy, J. & Blackwell, T. (2007) Particle swarm optimization - an overview. *Swarm Intelligence*, 1(1), 1 August, 33–57.

Puchinger, J. & Raidl, G.R. (2005) Combining metaheuristics and exact algorithms in combinatorial optimization: a survey and classification. In Mira, J. & Álvarez, J.R. (eds) *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*. Lecture Notes in Computer Science 3562. Berlin: Springer, 41–53.

Society of Automotive Engineers (2010) *ARP4754-A: Guidelines for development of civil aircraft and systems*. Warrendale: SAE International.

Sorokos, I., Papadopoulos, Y., Azevedo, L., Parker, D. & Walker, M. (2015a) Automating allocation of development assurance levels: an extension to HiP-HOPS. *IFAC-PapersOnLine*, 48(7). Available online: http://www.sciencedirect.com/science/article/pii/S2405896315007053 [Accessed 15/11/2015].

Sorokos, I., Papadopoulos, Y., Walker, M., Azevedo, L.S. & Parker, D. (2015b) Driving design refinement: how to optimise allocation of software development assurance or integrity requirements. In Mistrik, I., Soley, R.M., Ali, N., Grundy, J. & Tekinerdogan, B.

(eds) *Software Quality Assurance: In Large Scale and Complex Software-intensive Systems*. Burlington: Morgan Kaufmann, 237–250.

Stacey, A., Jancic, M. & Grundy, I. (2003) Particle swarm optimization with mutation. *The 2003 Congress on Evolutionary Computation (CEC '03)*. Camberra, December 2003. IEEE, 1425–1430 Vol.2.

Storey, N.R. (1996) *Safety critical computer systems*. Boston: Addison-Wesley Longman Publishing Co., Inc.

Syswerda, G. (1989) Uniform crossover in genetic algorithms. *The 3rd International Conference on Genetic Algorithms*. Fairfax, June 1989. Morgan Kaufmann, 2–9.

Talbi, E.-G. (2009) *Metaheuristics: from design to implementation*. Hoboken: John Wiley & Sons.

Tate, D.M. & Smith, A.E. (1995) A genetic approach to the quadratic assignment problem. *Computers & Operations Research*, 22(1), 73–83.

The Eclipse Foundation (2015) *Papyrus*. Available online: http://www.eclipse.org/papyrus/ [Accessed 7/5/2015].

Trelea, I.C. (2003) The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information Processing Letters*, 85(6), 31 March, 317–325.

U.S. Department of Defense (1949) *MIL–P–1629: Procedures for performing a failure mode, effects and criticality analysis*. Washington DC: U.S. Department of Defense.

Walker, M. & Papadopoulos, Y. (2008) Synthesis and analysis of temporal fault trees with PANDORA1: the time of Priority AND gates. *Nonlinear Analysis: Hybrid Systems*, 2(2), June, 368–382.

Wang, Y. & Li, L. (2012) Heterogeneous redundancy allocation for series-parallel multi-state systems using hybrid particle swarm optimization and local search. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 42(2), March, 464–474.

Watson, H.A. (1961) Method of inadvertent launch control analysis. In *Launch control safety study*. Murray Hill: Bell Labs.

Wolforth, I., Walker, M., Papadopoulos, Y. & Grunske, L. (2010) Capture and reuse of composable failure patterns. *International Journal of Critical Computer-Based Systems*, 1(1-3), 128–147.

Wolpert, D.H. & Macready, W.G. (1997) No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1), April, 67–82.

Wolsey, L.A. & Nemhauser, G.L. (1988) *Integer and combinatorial optimization*. New York: John Wiley & Sons.

Woon, S.F. (2009) *Global algorithms for nonlinear discrete optimization and discrete-valued optimal control problems*. PhD thesis. Curtin University of Technology. Available online:
http://espace.library.curtin.edu.au/webclient/StreamGate?folder_id=0&dvs=1439856178067~249&usePid1=true&usePid2=true [Accessed 18/8/2013].

Xiang, W., Richardson, P.C., Zhao, C. & Mohammad, S. (2008) Automobile brake-by-wire control system design and analysis. *IEEE Transactions on Vehicular Technology*, 57(1), January, 138–145.

Yang, X.-S. (2010) Introduction. In *Nature-inspired metaheuristic algorithms*, 2nd edition. Frome: Luniver Press.

Appendix I – HBSM1 failure logic and cut sets

HBSM1 failure expressions

Table 47 – HBSM1 failure expressions

Component	Failure expressions
Braking Pedal	O-Out1 = OFailure1
	V-Out1 = OFailure1
Electronic Pedal	O-Out1 = OFailure1 OR O-In1
	O-Out2 = OFailure2 OR O-In1
	V-Out1 = VFailure1 OR V-In1
	V-Out2 = VFailure2 OR V-In1
Bus1	O-Out1 = O-In1 AND O-In2 OR OFailure1
Bus2	O-Out1 = O-In1 AND O-In2 OR OFailure1
WNC	O-Out1 = O-In1 AND O-In2 OR OFailure1
	O-Out2 = O-In1 AND O-In2 OR OFailure2
	V-Out1 = O-In1 AND V-In2 OR VFailure1
	V-Out2 = O-In1 AND V-In2 OR VFailure2
Auxiliary Battery	O-Out1 = OFailure1
	V-Out1 = OFailure1
Powertrain Battery	O-Out1 = OFailure1 OR O-In1
	V-Out1 = VFailure1 OR V-In1
EMB Power Converter	O-Out1 = OFailure1 OR O-In1
	V-Out1 = VFailure1 OR V-In1
IWM Power Converter	O-Out1 = OFailure1 OR O-In1
	V-Out1 = VFailure1 OR V-In1
EMB	O-Out1 = OFailure1 OR O-In1
	V-Out1 = VFailure1 OR V-In1
IWM	O-Out1 = OFailure1 OR O-In1
	V-Out1 = VFailure1 OR V-In1

HBSM1 cut sets

Table 48 – HSBM1 cut sets

No braking after command	Wrong value braking
Braking Pedal.OFailure1	Braking Pedal.VFailure1
Electronic Pedal.OFailure1 Electronic Pedal.OFailure2	Braking Pedal.OFailure1 Electronic Pedal.VFailure2
Bus 1.OFailure1 Bus 2.OFailure1	Electronic Pedal.VFailure1
WNC.OFailure1 WNC OFailure2	Electronic Pedal.OFailure1 Electronic Pedal.VFailure2
WNC.OFailure1 IWM Power Converter.OFailure1	WNC.VFailure1
WNC.OFailure1 IWM.OFailure1	WNC.VFailure2
WNC.OFailure1 Powertrain Battery.OFailure1	EMB Power Converter.VFailure1
EMB Power Converter.OFailure1 WNC.OFailure2	IWM Power Converter.VFailure1
EMB Power Converter.OFailure1 IWM Power Converter.OFailure1	EMB.VFailure1
EMB Power Converter.OFailure1 IWM.OFailure1	IWM.VFailure1
EMB Power Converter.OFailure1 Powertrain Battery.OFailure1	Powertrain Battery.VFailure1
EMB.OFailure1 WNC.OFailure2	Auxiliary Battery.VFailure1
EMB.OFailure1 IWM Power Converter.OFailure1	
EMB.OFailure1 IWM.OFailure1	
EMB.OFailure1 Powertrain Battery.OFailure1	
Auxiliary Battery.OFailure1 WNC.OFailure2	
Auxiliary Battery.OFailure1 IWM Power Converter.OFailure1	
Auxiliary Battery.OFailure1 IWM.OFailure1	
Auxiliary Battery.OFailure1 Powertrain Battery.OFailure1	

Appendix II – Random ASIL cost functions for HBSM1

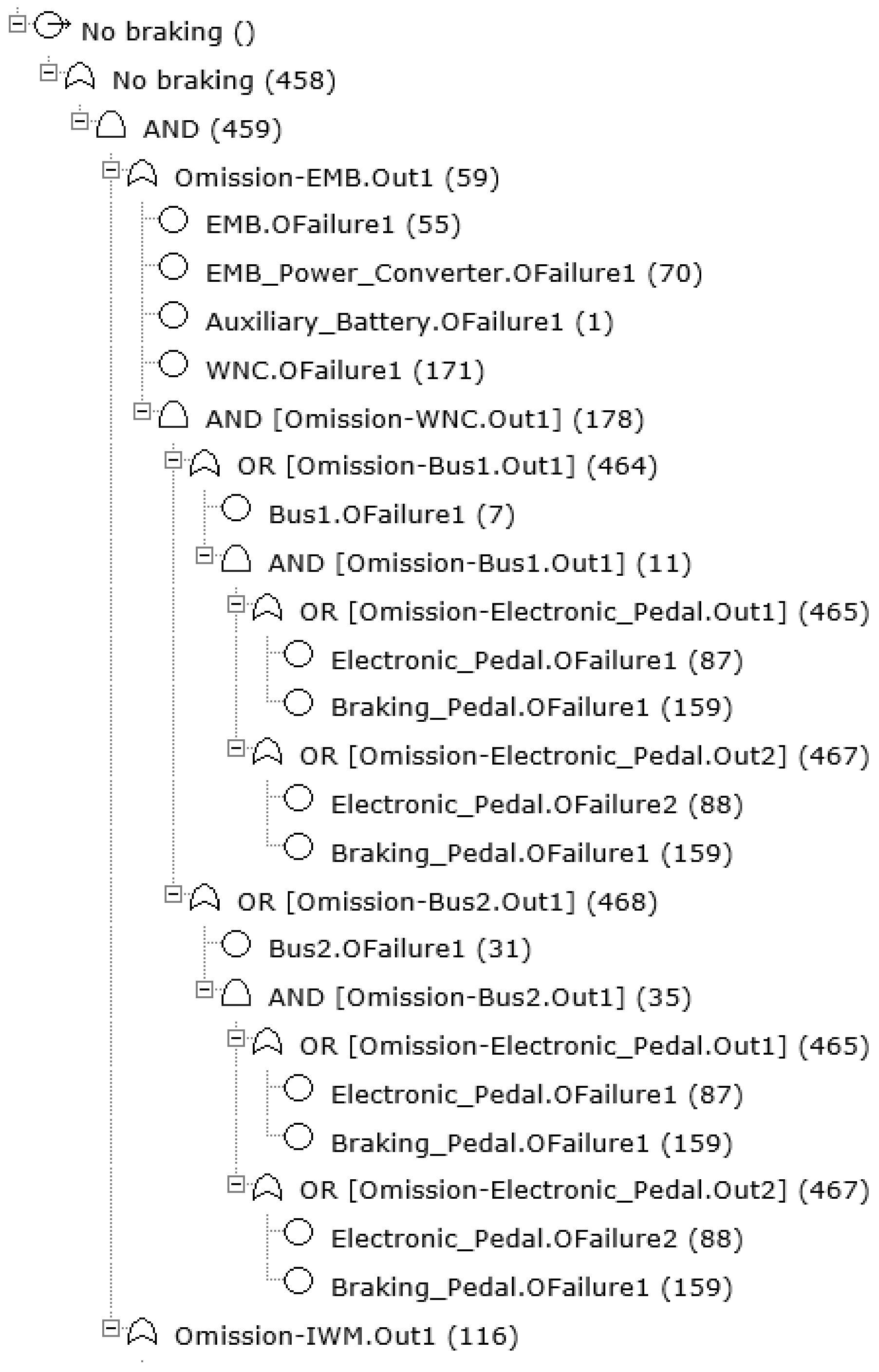
Table 49 – Random ASIL cost functions for the different FMs of HBSM1.

FM	ASIL QM	ASIL A	ASIL B	ASIL C	ASIL D
Braking Pedal.OFailure1	0	38	154	178	189
Braking Pedal.VFailure1	0	22	140	161	239
Electronic Pedal.OFailure1	0	2	48	164	238
Electronic Pedal.OFailure2	0	37	99	231	249
Electronic Pedal.VFailure1	0	41	161	165	174
Electronic Pedal.VFailure2	0	28	122	130	156
Bus1.OFailure1	0	67	114	157	166
Bus2.OFailure1	0	66	101	111	222
WNC.OFailure1	0	13	17	90	153
WNC.OFailure2	0	14	53	69	134
WNC.VFailure1	0	11	43	72	170
WNC.VFailure2	0	44	58	202	207
Auxiliary Battery.OFailure1	0	70	82	157	159
Auxiliary Battery.VFailure1	0	30	41	103	202
Powertrain Battery.OFailure1	0	80	138	239	241
Powertrain Battery.VFailure1	0	26	72	148	188
EMB Power Converter.OFailure1	0	87	88	150	187
EMB Power Converter.VFailure1	0	76	121	241	246
IWM Power Converter.OFailure1	0	122	143	181	236
IWM Power Converter.VFailure1	0	6	16	107	183
EMB.OFailure1	0	16	68	187	232
EMB.VFailure1	0	66	71	80	98
IWM.OFailure1	0	30	108	160	206
IWM.VFailure1	0	129	137	200	237

Appendix III – Case studies' fault trees

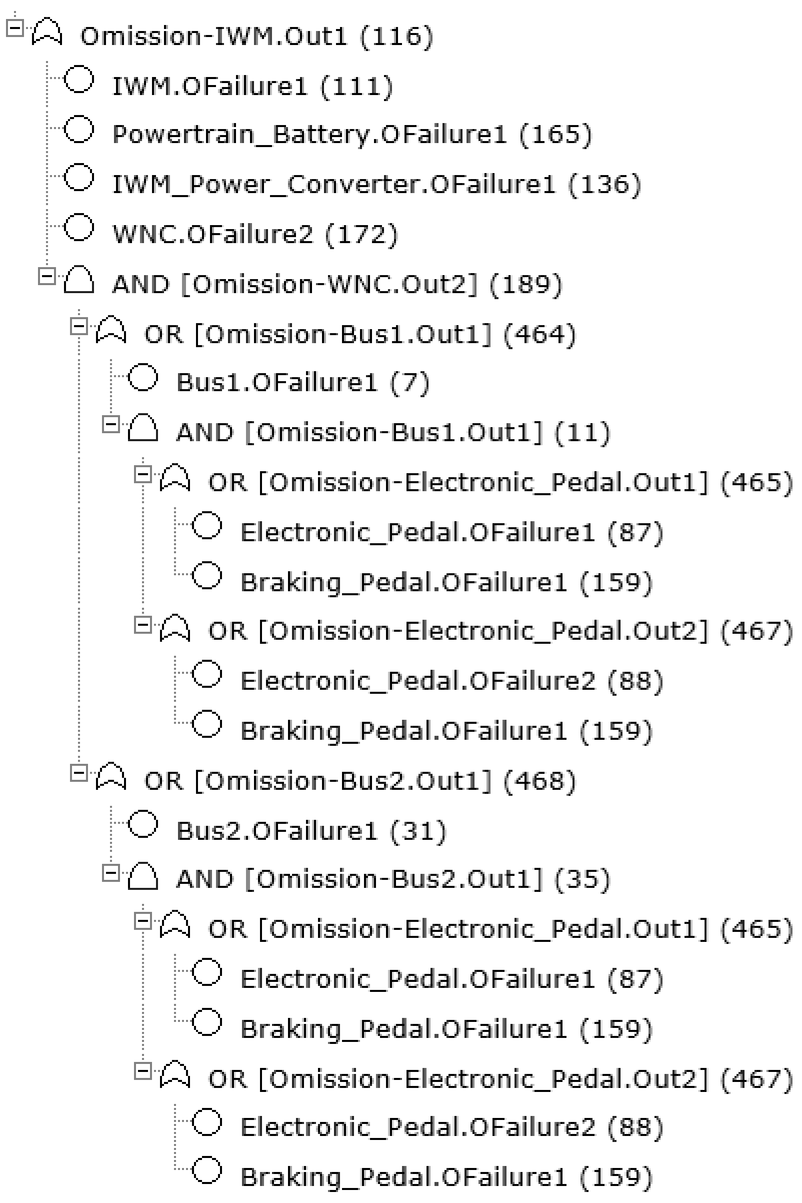
HBSM1 fault trees

No braking



```
No braking ()
  No braking (458)
    AND (459)
      Omission-EMB.Out1 (59)
        EMB.OFailure1 (55)
        EMB_Power_Converter.OFailure1 (70)
        Auxiliary_Battery.OFailure1 (1)
        WNC.OFailure1 (171)
        AND [Omission-WNC.Out1] (178)
          OR [Omission-Bus1.Out1] (464)
            Bus1.OFailure1 (7)
            AND [Omission-Bus1.Out1] (11)
              OR [Omission-Electronic_Pedal.Out1] (465)
                Electronic_Pedal.OFailure1 (87)
                Braking_Pedal.OFailure1 (159)
              OR [Omission-Electronic_Pedal.Out2] (467)
                Electronic_Pedal.OFailure2 (88)
                Braking_Pedal.OFailure1 (159)
          OR [Omission-Bus2.Out1] (468)
            Bus2.OFailure1 (31)
            AND [Omission-Bus2.Out1] (35)
              OR [Omission-Electronic_Pedal.Out1] (465)
                Electronic_Pedal.OFailure1 (87)
                Braking_Pedal.OFailure1 (159)
              OR [Omission-Electronic_Pedal.Out2] (467)
                Electronic_Pedal.OFailure2 (88)
                Braking_Pedal.OFailure1 (159)
      Omission-IWM.Out1 (116)
```

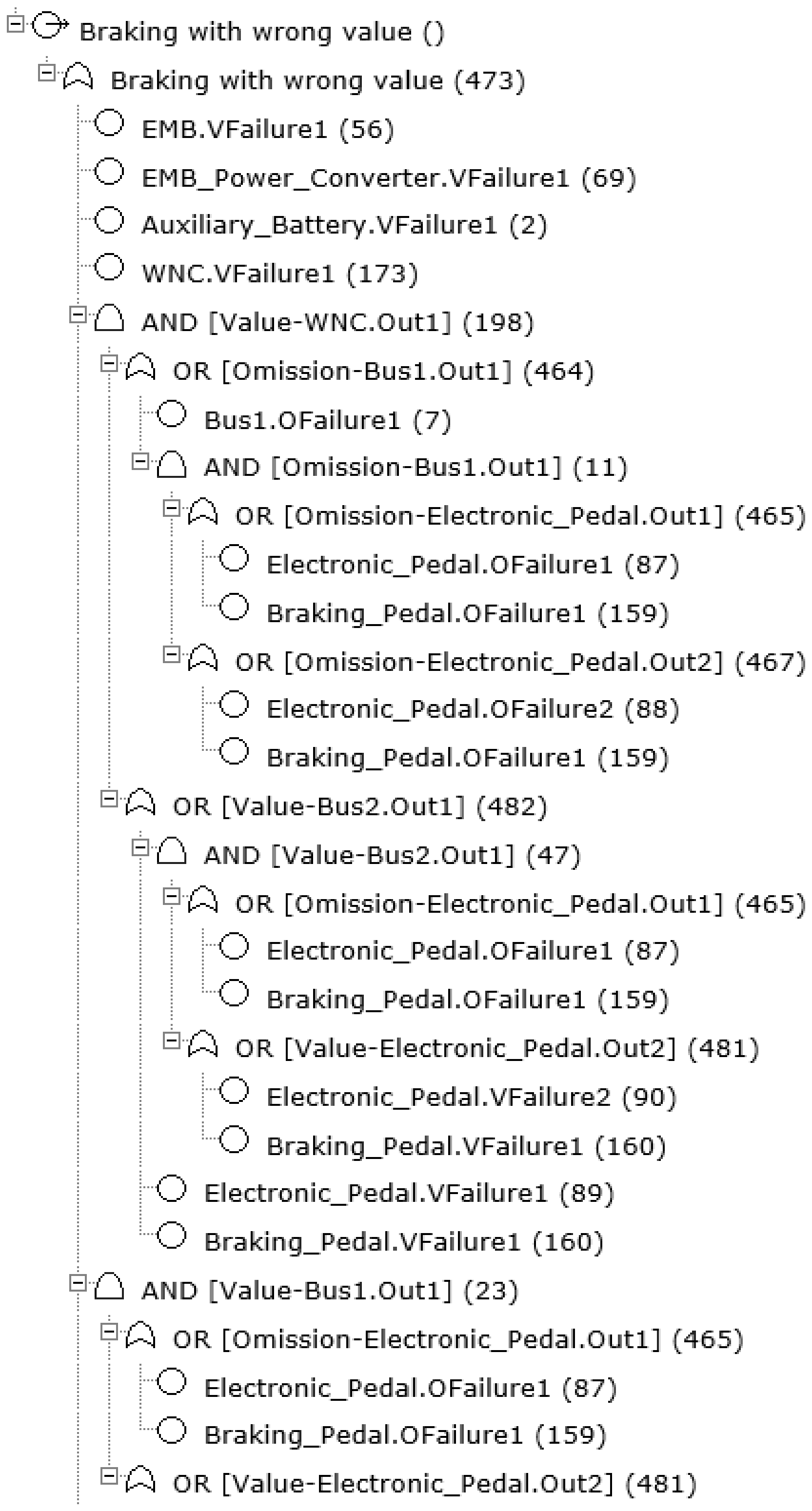
Figure 66 – HBSM1 fault trees: "No braking" hazard – part 1 of 2.



```
Omission-IWM.Out1 (116)
    IWM.OFailure1 (111)
    Powertrain_Battery.OFailure1 (165)
    IWM_Power_Converter.OFailure1 (136)
    WNC.OFailure2 (172)
    AND [Omission-WNC.Out2] (189)
        OR [Omission-Bus1.Out1] (464)
            Bus1.OFailure1 (7)
            AND [Omission-Bus1.Out1] (11)
                OR [Omission-Electronic_Pedal.Out1] (465)
                    Electronic_Pedal.OFailure1 (87)
                    Braking_Pedal.OFailure1 (159)
                OR [Omission-Electronic_Pedal.Out2] (467)
                    Electronic_Pedal.OFailure2 (88)
                    Braking_Pedal.OFailure1 (159)
        OR [Omission-Bus2.Out1] (468)
            Bus2.OFailure1 (31)
            AND [Omission-Bus2.Out1] (35)
                OR [Omission-Electronic_Pedal.Out1] (465)
                    Electronic_Pedal.OFailure1 (87)
                    Braking_Pedal.OFailure1 (159)
                OR [Omission-Electronic_Pedal.Out2] (467)
                    Electronic_Pedal.OFailure2 (88)
                    Braking_Pedal.OFailure1 (159)
```

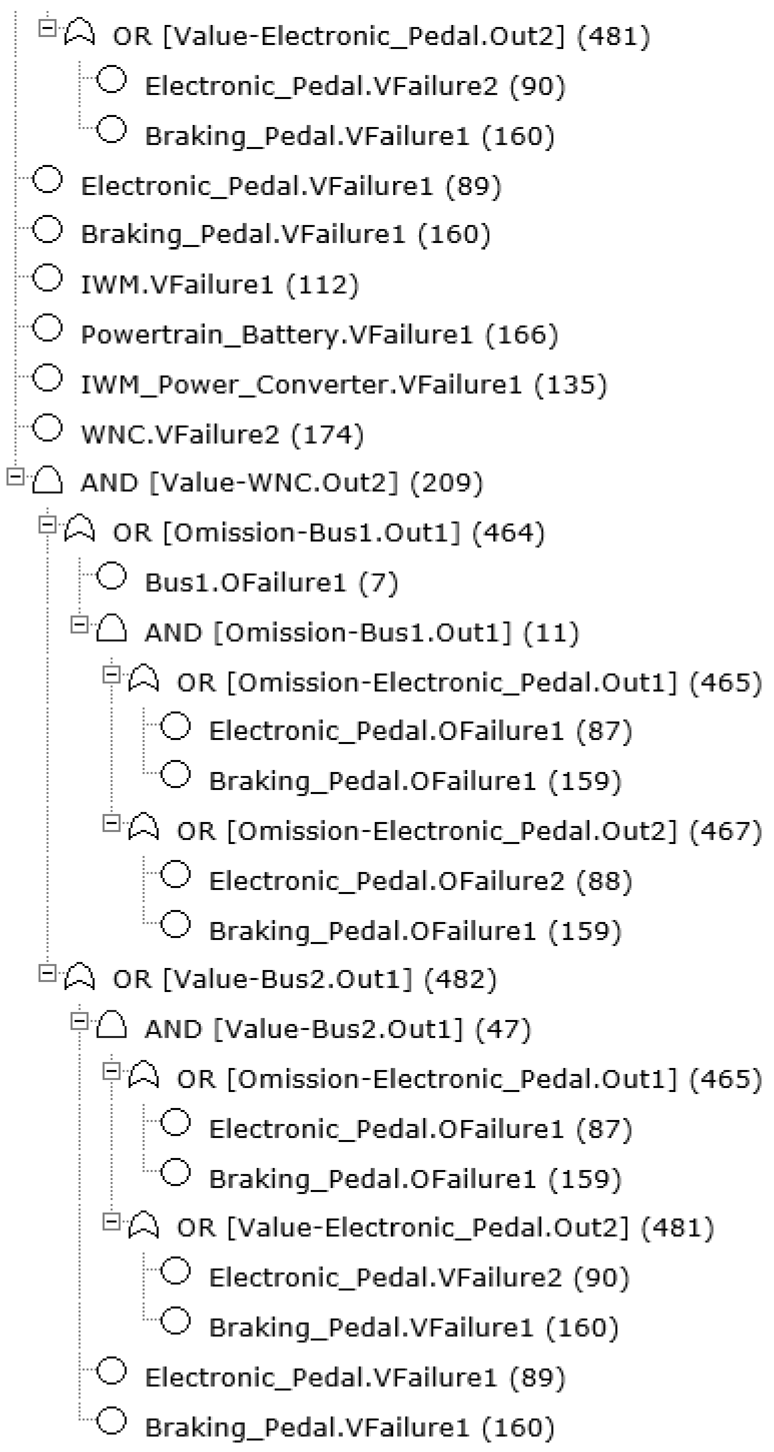
Figure 67 - HBSM1 fault trees: “No braking” hazard – part 2 of 2.

Braking with wrong value



```
Braking with wrong value ()
  Braking with wrong value (473)
    EMB.VFailure1 (56)
    EMB_Power_Converter.VFailure1 (69)
    Auxiliary_Battery.VFailure1 (2)
    WNC.VFailure1 (173)
    AND [Value-WNC.Out1] (198)
      OR [Omission-Bus1.Out1] (464)
        Bus1.OFailure1 (7)
        AND [Omission-Bus1.Out1] (11)
          OR [Omission-Electronic_Pedal.Out1] (465)
            Electronic_Pedal.OFailure1 (87)
            Braking_Pedal.OFailure1 (159)
          OR [Omission-Electronic_Pedal.Out2] (467)
            Electronic_Pedal.OFailure2 (88)
            Braking_Pedal.OFailure1 (159)
      OR [Value-Bus2.Out1] (482)
        AND [Value-Bus2.Out1] (47)
          OR [Omission-Electronic_Pedal.Out1] (465)
            Electronic_Pedal.OFailure1 (87)
            Braking_Pedal.OFailure1 (159)
          OR [Value-Electronic_Pedal.Out2] (481)
            Electronic_Pedal.VFailure2 (90)
            Braking_Pedal.VFailure1 (160)
        Electronic_Pedal.VFailure1 (89)
        Braking_Pedal.VFailure1 (160)
    AND [Value-Bus1.Out1] (23)
      OR [Omission-Electronic_Pedal.Out1] (465)
        Electronic_Pedal.OFailure1 (87)
        Braking_Pedal.OFailure1 (159)
      OR [Value-Electronic_Pedal.Out2] (481)
```

Figure 68 - HBSM1 fault trees: "Braking with wrong value" hazard – part 1 of 2.

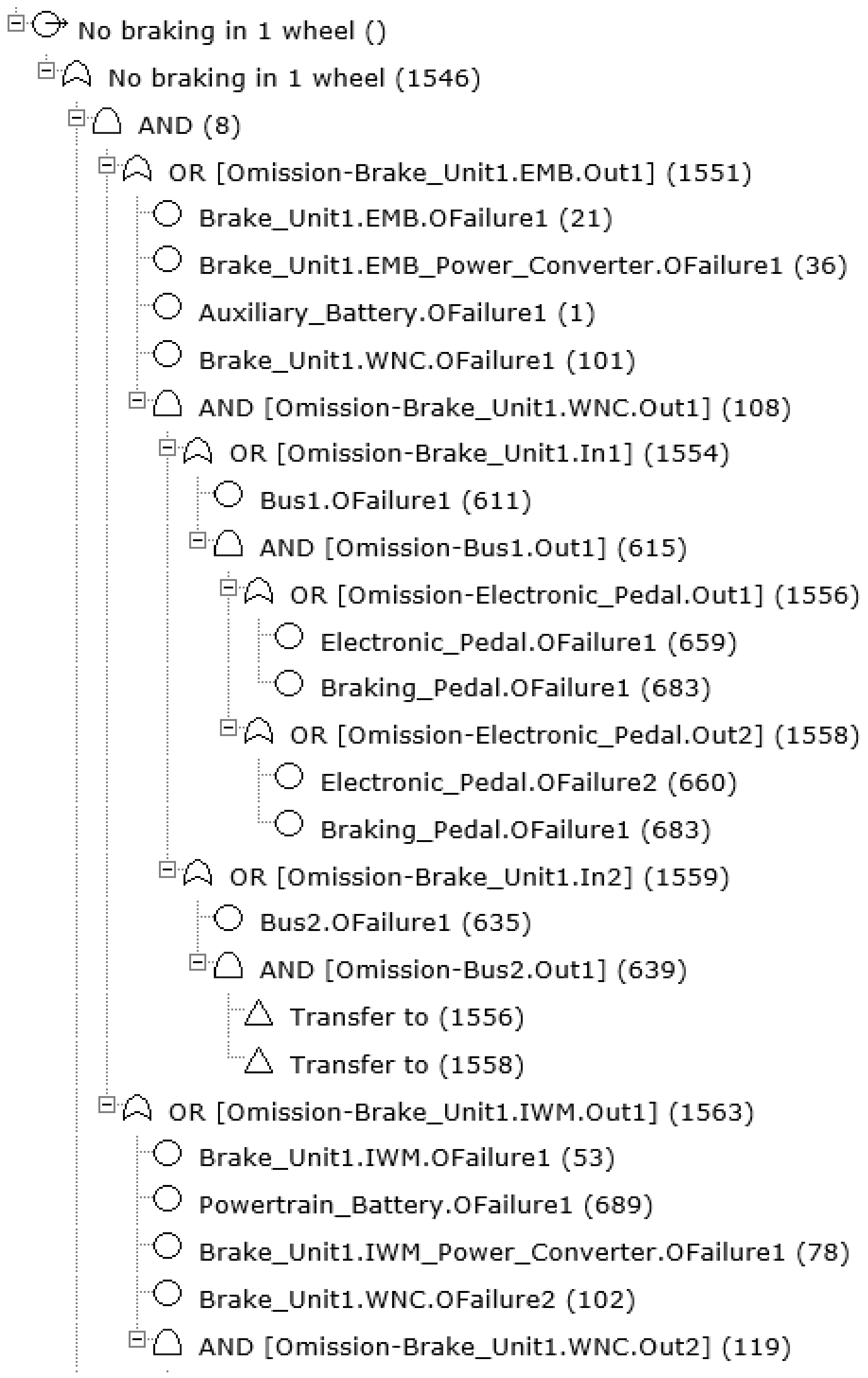


```
  OR [Value-Electronic_Pedal.Out2] (481)
    Electronic_Pedal.VFailure2 (90)
    Braking_Pedal.VFailure1 (160)
  Electronic_Pedal.VFailure1 (89)
  Braking_Pedal.VFailure1 (160)
  IWM.VFailure1 (112)
  Powertrain_Battery.VFailure1 (166)
  IWM_Power_Converter.VFailure1 (135)
  WNC.VFailure2 (174)
AND [Value-WNC.Out2] (209)
  OR [Omission-Bus1.Out1] (464)
    Bus1.OFailure1 (7)
    AND [Omission-Bus1.Out1] (11)
      OR [Omission-Electronic_Pedal.Out1] (465)
        Electronic_Pedal.OFailure1 (87)
        Braking_Pedal.OFailure1 (159)
      OR [Omission-Electronic_Pedal.Out2] (467)
        Electronic_Pedal.OFailure2 (88)
        Braking_Pedal.OFailure1 (159)
  OR [Value-Bus2.Out1] (482)
    AND [Value-Bus2.Out1] (47)
      OR [Omission-Electronic_Pedal.Out1] (465)
        Electronic_Pedal.OFailure1 (87)
        Braking_Pedal.OFailure1 (159)
      OR [Value-Electronic_Pedal.Out2] (481)
        Electronic_Pedal.VFailure2 (90)
        Braking_Pedal.VFailure1 (160)
    Electronic_Pedal.VFailure1 (89)
    Braking_Pedal.VFailure1 (160)
```

Figure 69 - HBSM1 fault trees: "Braking with wrong value" hazard – part 2 of 2.

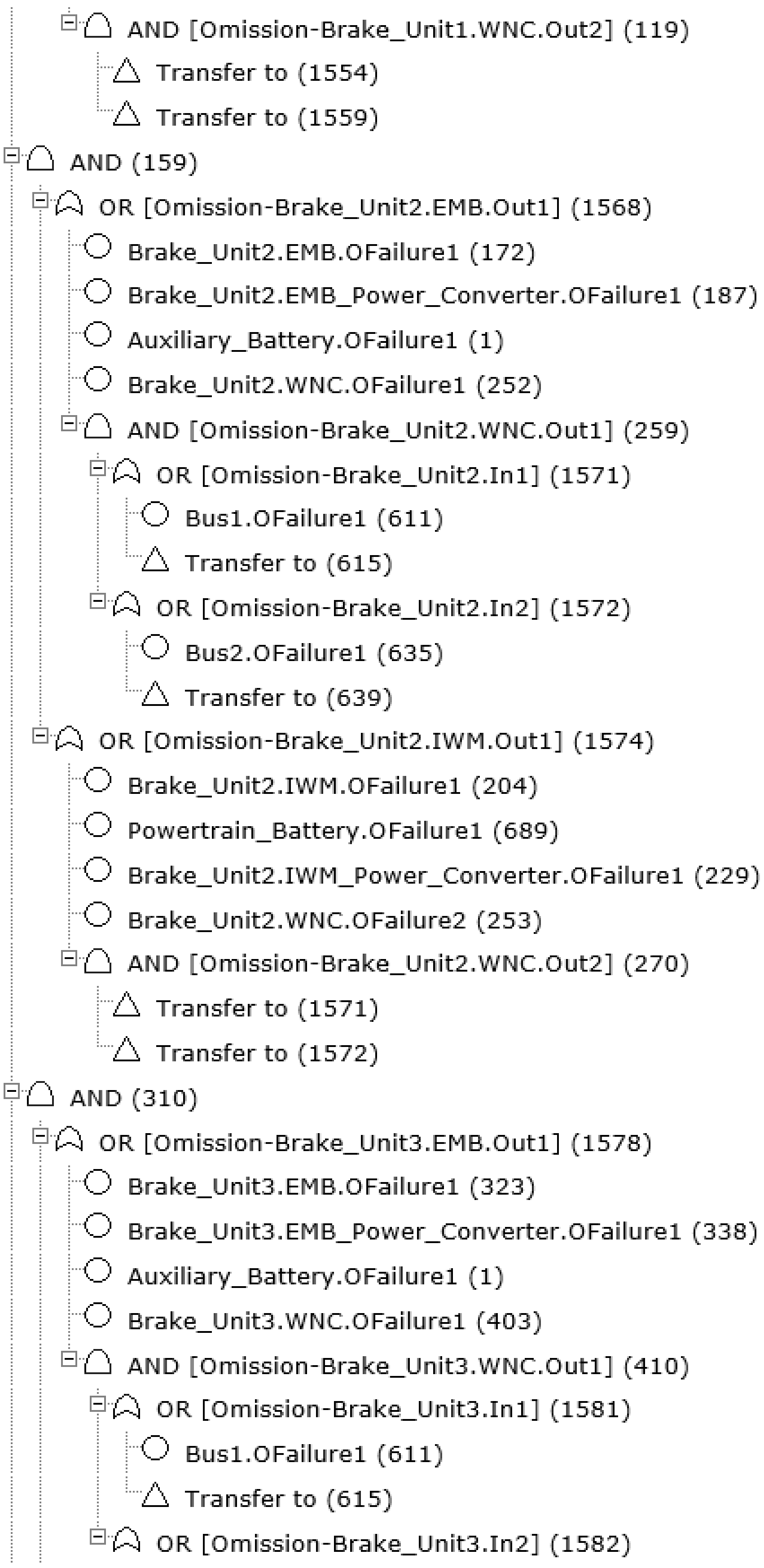
HBSM2 fault trees

No braking in 1 wheel



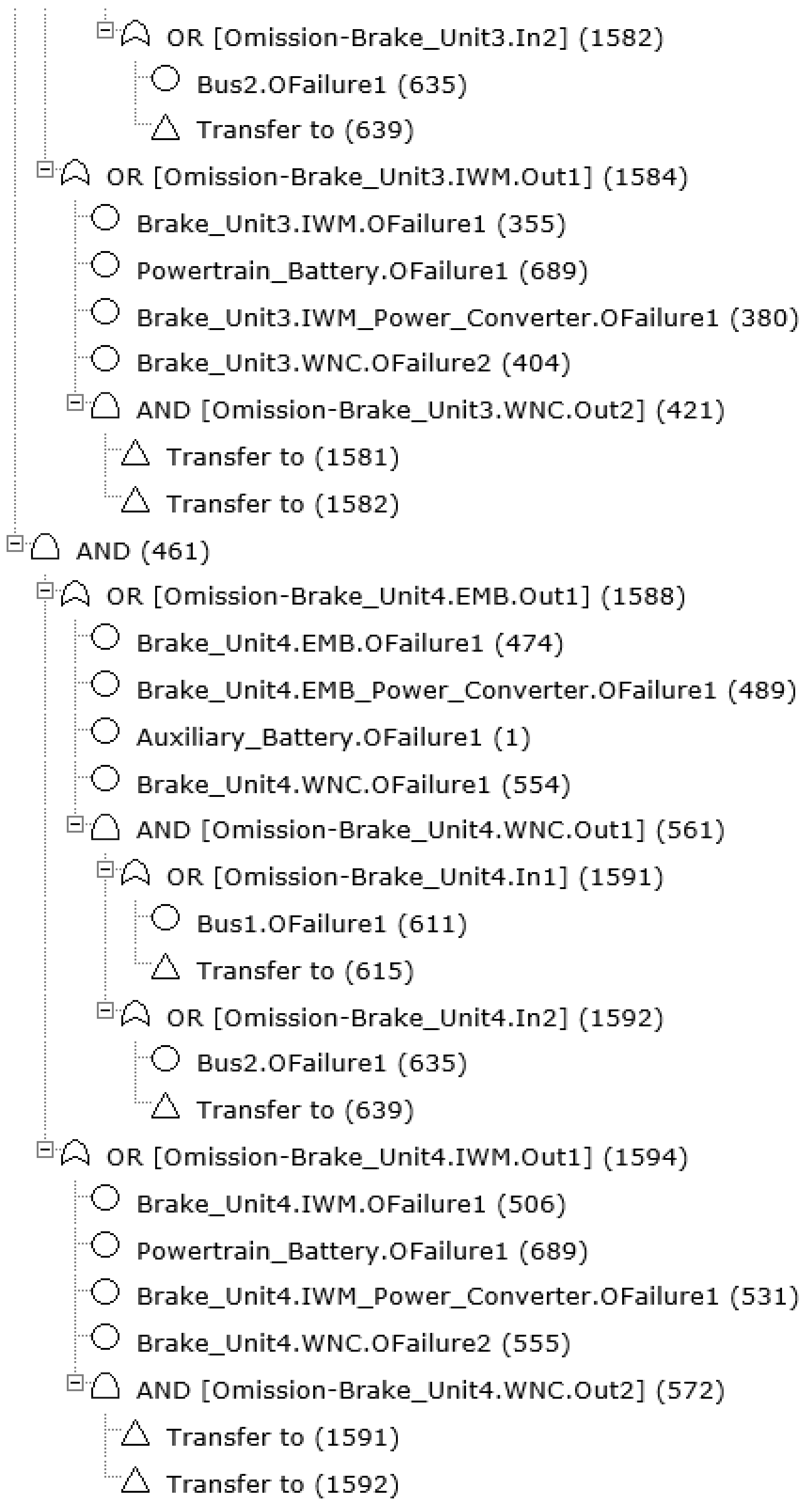
```
No braking in 1 wheel ()
  No braking in 1 wheel (1546)
    AND (8)
      OR [Omission-Brake_Unit1.EMB.Out1] (1551)
        Brake_Unit1.EMB.OFailure1 (21)
        Brake_Unit1.EMB_Power_Converter.OFailure1 (36)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit1.WNC.OFailure1 (101)
        AND [Omission-Brake_Unit1.WNC.Out1] (108)
          OR [Omission-Brake_Unit1.In1] (1554)
            Bus1.OFailure1 (611)
            AND [Omission-Bus1.Out1] (615)
              OR [Omission-Electronic_Pedal.Out1] (1556)
                Electronic_Pedal.OFailure1 (659)
                Braking_Pedal.OFailure1 (683)
              OR [Omission-Electronic_Pedal.Out2] (1558)
                Electronic_Pedal.OFailure2 (660)
                Braking_Pedal.OFailure1 (683)
          OR [Omission-Brake_Unit1.In2] (1559)
            Bus2.OFailure1 (635)
            AND [Omission-Bus2.Out1] (639)
              Transfer to (1556)
              Transfer to (1558)
      OR [Omission-Brake_Unit1.IWM.Out1] (1563)
        Brake_Unit1.IWM.OFailure1 (53)
        Powertrain_Battery.OFailure1 (689)
        Brake_Unit1.IWM_Power_Converter.OFailure1 (78)
        Brake_Unit1.WNC.OFailure2 (102)
        AND [Omission-Brake_Unit1.WNC.Out2] (119)
```

Figure 70 – HBSM2 fault trees: “No braking in one wheel” hazard – part 1 of 3.



```
    AND [Omission-Brake_Unit1.WNC.Out2] (119)
      Transfer to (1554)
      Transfer to (1559)
AND (159)
  OR [Omission-Brake_Unit2.EMB.Out1] (1568)
    Brake_Unit2.EMB.OFailure1 (172)
    Brake_Unit2.EMB_Power_Converter.OFailure1 (187)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit2.WNC.OFailure1 (252)
    AND [Omission-Brake_Unit2.WNC.Out1] (259)
      OR [Omission-Brake_Unit2.In1] (1571)
        Bus1.OFailure1 (611)
        Transfer to (615)
      OR [Omission-Brake_Unit2.In2] (1572)
        Bus2.OFailure1 (635)
        Transfer to (639)
  OR [Omission-Brake_Unit2.IWM.Out1] (1574)
    Brake_Unit2.IWM.OFailure1 (204)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit2.IWM_Power_Converter.OFailure1 (229)
    Brake_Unit2.WNC.OFailure2 (253)
    AND [Omission-Brake_Unit2.WNC.Out2] (270)
      Transfer to (1571)
      Transfer to (1572)
AND (310)
  OR [Omission-Brake_Unit3.EMB.Out1] (1578)
    Brake_Unit3.EMB.OFailure1 (323)
    Brake_Unit3.EMB_Power_Converter.OFailure1 (338)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit3.WNC.OFailure1 (403)
    AND [Omission-Brake_Unit3.WNC.Out1] (410)
      OR [Omission-Brake_Unit3.In1] (1581)
        Bus1.OFailure1 (611)
        Transfer to (615)
      OR [Omission-Brake_Unit3.In2] (1582)
```

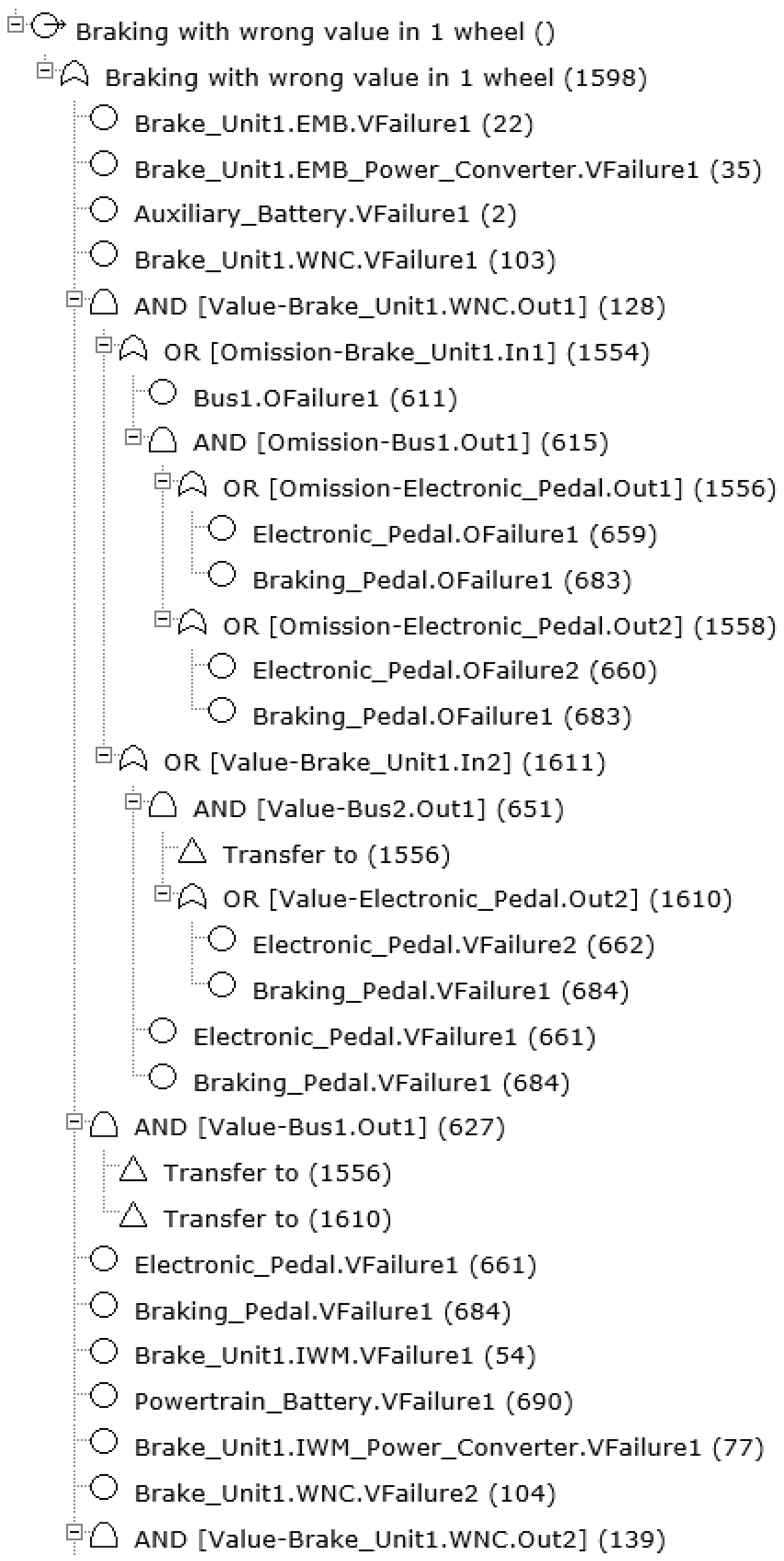
Figure 71 - HBSM2 fault trees: “No braking in one wheel” hazard – part 2 of 3.



```
      OR [Omission-Brake_Unit3.In2] (1582)
        Bus2.OFailure1 (635)
        Transfer to (639)
    OR [Omission-Brake_Unit3.IWM.Out1] (1584)
      Brake_Unit3.IWM.OFailure1 (355)
      Powertrain_Battery.OFailure1 (689)
      Brake_Unit3.IWM_Power_Converter.OFailure1 (380)
      Brake_Unit3.WNC.OFailure2 (404)
      AND [Omission-Brake_Unit3.WNC.Out2] (421)
        Transfer to (1581)
        Transfer to (1582)
  AND (461)
    OR [Omission-Brake_Unit4.EMB.Out1] (1588)
      Brake_Unit4.EMB.OFailure1 (474)
      Brake_Unit4.EMB_Power_Converter.OFailure1 (489)
      Auxiliary_Battery.OFailure1 (1)
      Brake_Unit4.WNC.OFailure1 (554)
      AND [Omission-Brake_Unit4.WNC.Out1] (561)
        OR [Omission-Brake_Unit4.In1] (1591)
          Bus1.OFailure1 (611)
          Transfer to (615)
        OR [Omission-Brake_Unit4.In2] (1592)
          Bus2.OFailure1 (635)
          Transfer to (639)
    OR [Omission-Brake_Unit4.IWM.Out1] (1594)
      Brake_Unit4.IWM.OFailure1 (506)
      Powertrain_Battery.OFailure1 (689)
      Brake_Unit4.IWM_Power_Converter.OFailure1 (531)
      Brake_Unit4.WNC.OFailure2 (555)
      AND [Omission-Brake_Unit4.WNC.Out2] (572)
        Transfer to (1591)
        Transfer to (1592)
```

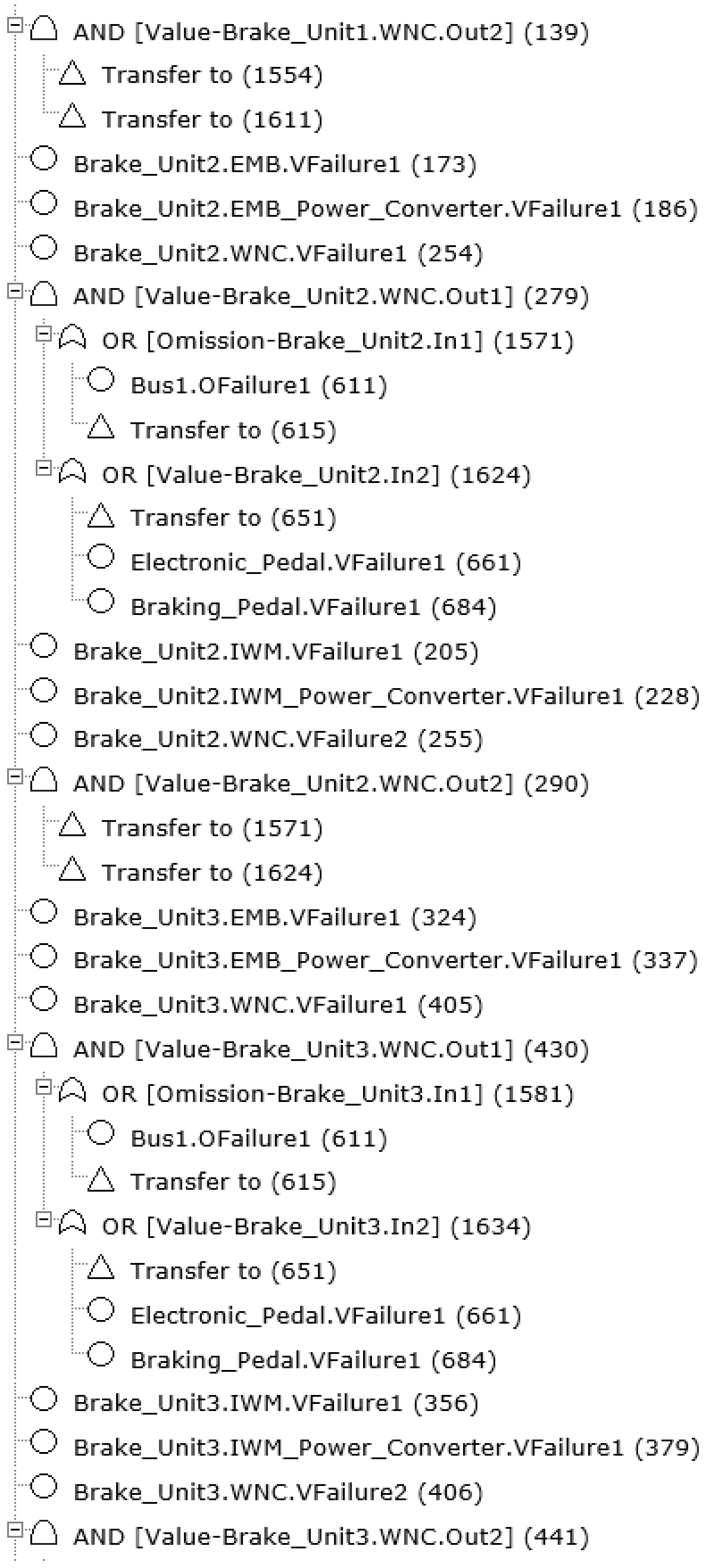
Figure 72 - HBSM2 fault trees: “No braking in one wheel” hazard – part 3 of 3.

Braking with wrong value in 1 wheel



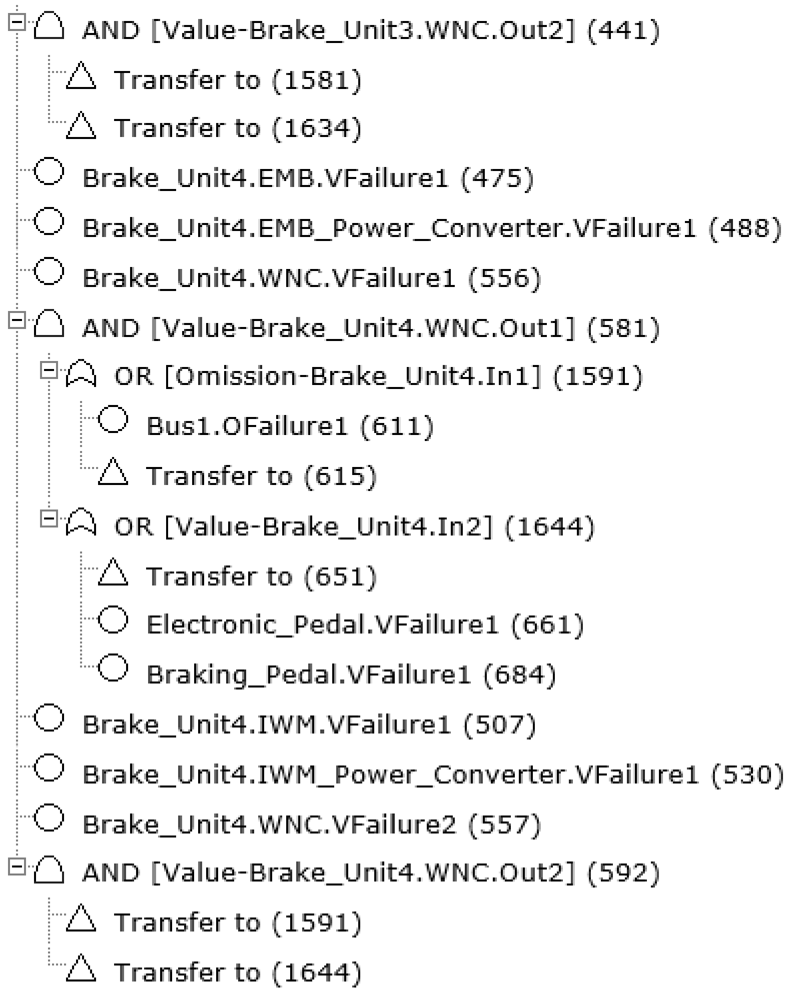
```
Braking with wrong value in 1 wheel ()
  Braking with wrong value in 1 wheel (1598)
    Brake_Unit1.EMB.VFailure1 (22)
    Brake_Unit1.EMB_Power_Converter.VFailure1 (35)
    Auxiliary_Battery.VFailure1 (2)
    Brake_Unit1.WNC.VFailure1 (103)
    AND [Value-Brake_Unit1.WNC.Out1] (128)
      OR [Omission-Brake_Unit1.In1] (1554)
        Bus1.OFailure1 (611)
        AND [Omission-Bus1.Out1] (615)
          OR [Omission-Electronic_Pedal.Out1] (1556)
            Electronic_Pedal.OFailure1 (659)
            Braking_Pedal.OFailure1 (683)
          OR [Omission-Electronic_Pedal.Out2] (1558)
            Electronic_Pedal.OFailure2 (660)
            Braking_Pedal.OFailure1 (683)
      OR [Value-Brake_Unit1.In2] (1611)
        AND [Value-Bus2.Out1] (651)
          Transfer to (1556)
          OR [Value-Electronic_Pedal.Out2] (1610)
            Electronic_Pedal.VFailure2 (662)
            Braking_Pedal.VFailure1 (684)
        Electronic_Pedal.VFailure1 (661)
        Braking_Pedal.VFailure1 (684)
    AND [Value-Bus1.Out1] (627)
      Transfer to (1556)
      Transfer to (1610)
    Electronic_Pedal.VFailure1 (661)
    Braking_Pedal.VFailure1 (684)
    Brake_Unit1.IWM.VFailure1 (54)
    Powertrain_Battery.VFailure1 (690)
    Brake_Unit1.IWM_Power_Converter.VFailure1 (77)
    Brake_Unit1.WNC.VFailure2 (104)
    AND [Value-Brake_Unit1.WNC.Out2] (139)
```

Figure 73 – HBSM2 fault trees: “Braking with wrong value in 1 wheel” hazard – part 1 of 3.



```
AND [Value-Brake_Unit1.WNC.Out2] (139)
    Transfer to (1554)
    Transfer to (1611)
Brake_Unit2.EMB.VFailure1 (173)
Brake_Unit2.EMB_Power_Converter.VFailure1 (186)
Brake_Unit2.WNC.VFailure1 (254)
AND [Value-Brake_Unit2.WNC.Out1] (279)
    OR [Omission-Brake_Unit2.In1] (1571)
        Bus1.OFailure1 (611)
        Transfer to (615)
    OR [Value-Brake_Unit2.In2] (1624)
        Transfer to (651)
        Electronic_Pedal.VFailure1 (661)
        Braking_Pedal.VFailure1 (684)
Brake_Unit2.IWM.VFailure1 (205)
Brake_Unit2.IWM_Power_Converter.VFailure1 (228)
Brake_Unit2.WNC.VFailure2 (255)
AND [Value-Brake_Unit2.WNC.Out2] (290)
    Transfer to (1571)
    Transfer to (1624)
Brake_Unit3.EMB.VFailure1 (324)
Brake_Unit3.EMB_Power_Converter.VFailure1 (337)
Brake_Unit3.WNC.VFailure1 (405)
AND [Value-Brake_Unit3.WNC.Out1] (430)
    OR [Omission-Brake_Unit3.In1] (1581)
        Bus1.OFailure1 (611)
        Transfer to (615)
    OR [Value-Brake_Unit3.In2] (1634)
        Transfer to (651)
        Electronic_Pedal.VFailure1 (661)
        Braking_Pedal.VFailure1 (684)
Brake_Unit3.IWM.VFailure1 (356)
Brake_Unit3.IWM_Power_Converter.VFailure1 (379)
Brake_Unit3.WNC.VFailure2 (406)
AND [Value-Brake_Unit3.WNC.Out2] (441)
```

Figure 74 - HBSM2 fault trees: "Braking with wrong value in 1 wheel" hazard – part 2 of 3.

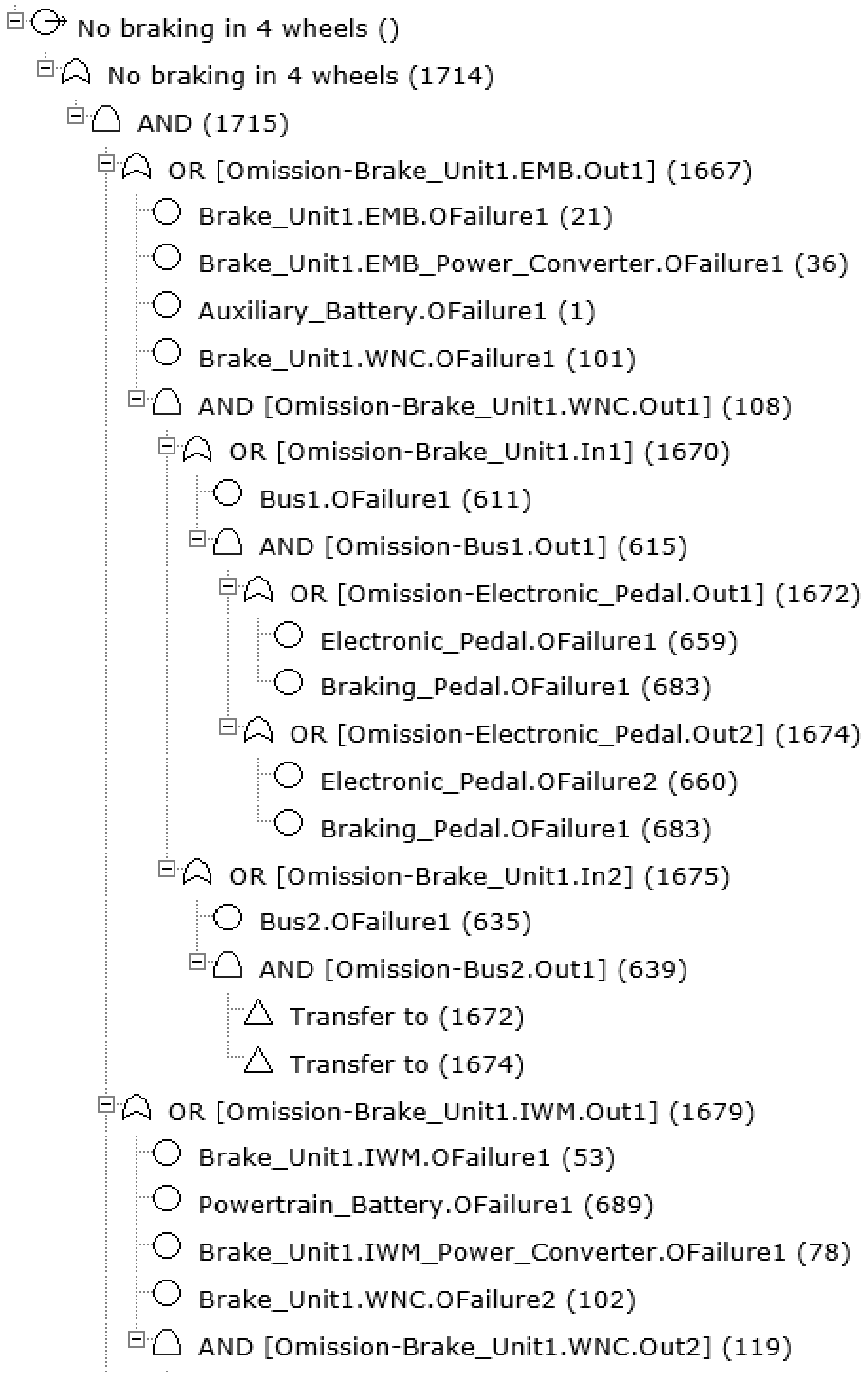


```
AND [Value-Brake_Unit3.WNC.Out2] (441)
    Transfer to (1581)
    Transfer to (1634)
Brake_Unit4.EMB.VFailure1 (475)
Brake_Unit4.EMB_Power_Converter.VFailure1 (488)
Brake_Unit4.WNC.VFailure1 (556)
AND [Value-Brake_Unit4.WNC.Out1] (581)
    OR [Omission-Brake_Unit4.In1] (1591)
        Bus1.OFailure1 (611)
        Transfer to (615)
    OR [Value-Brake_Unit4.In2] (1644)
        Transfer to (651)
        Electronic_Pedal.VFailure1 (661)
        Braking_Pedal.VFailure1 (684)
Brake_Unit4.IWM.VFailure1 (507)
Brake_Unit4.IWM_Power_Converter.VFailure1 (530)
Brake_Unit4.WNC.VFailure2 (557)
AND [Value-Brake_Unit4.WNC.Out2] (592)
    Transfer to (1591)
    Transfer to (1644)
```

Figure 75 - HBSM2 fault trees: "Braking with wrong value in 1 wheel" hazard – part 3 of 3.

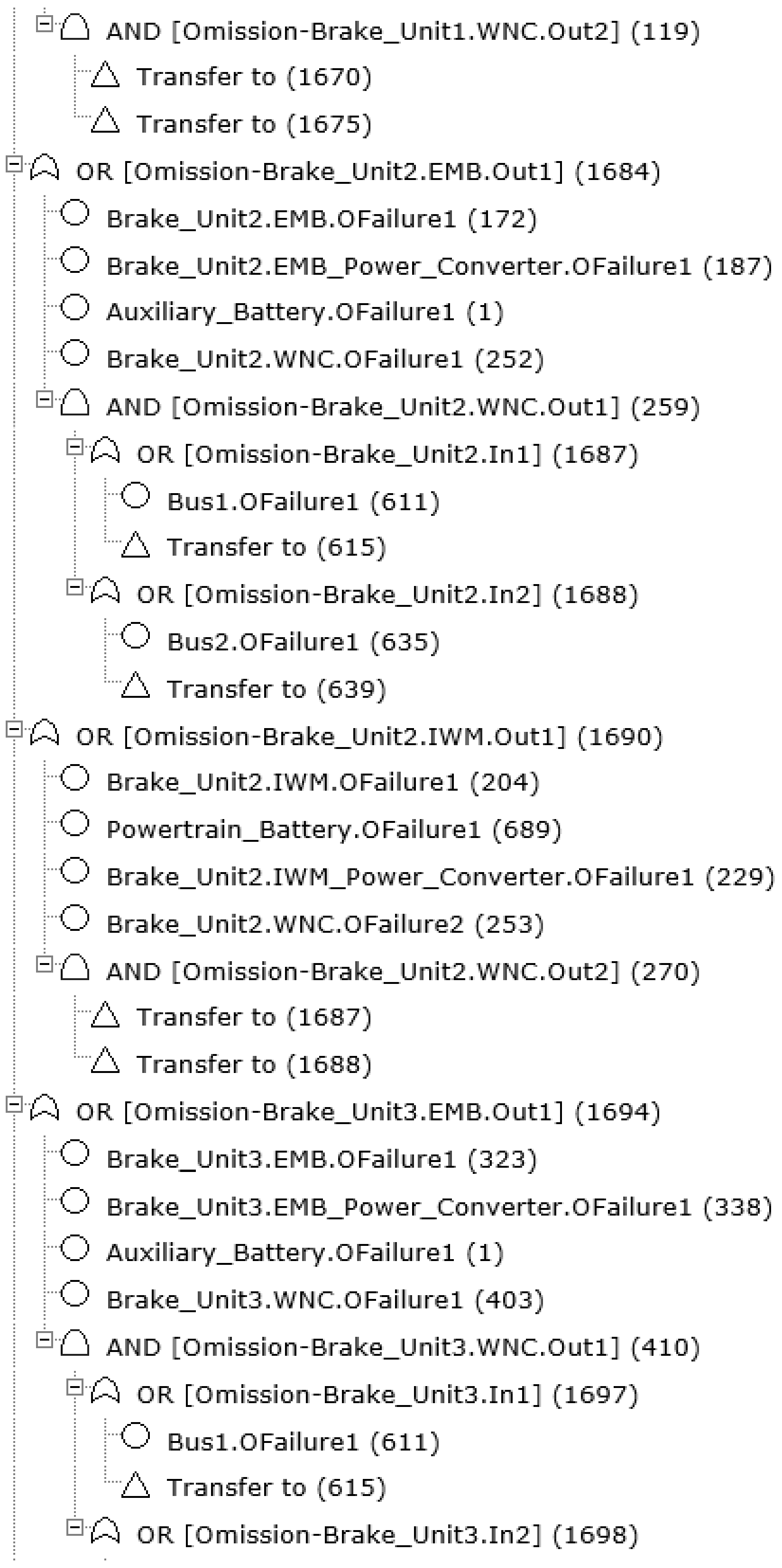
HBSM3 fault trees

No braking in 4 wheels



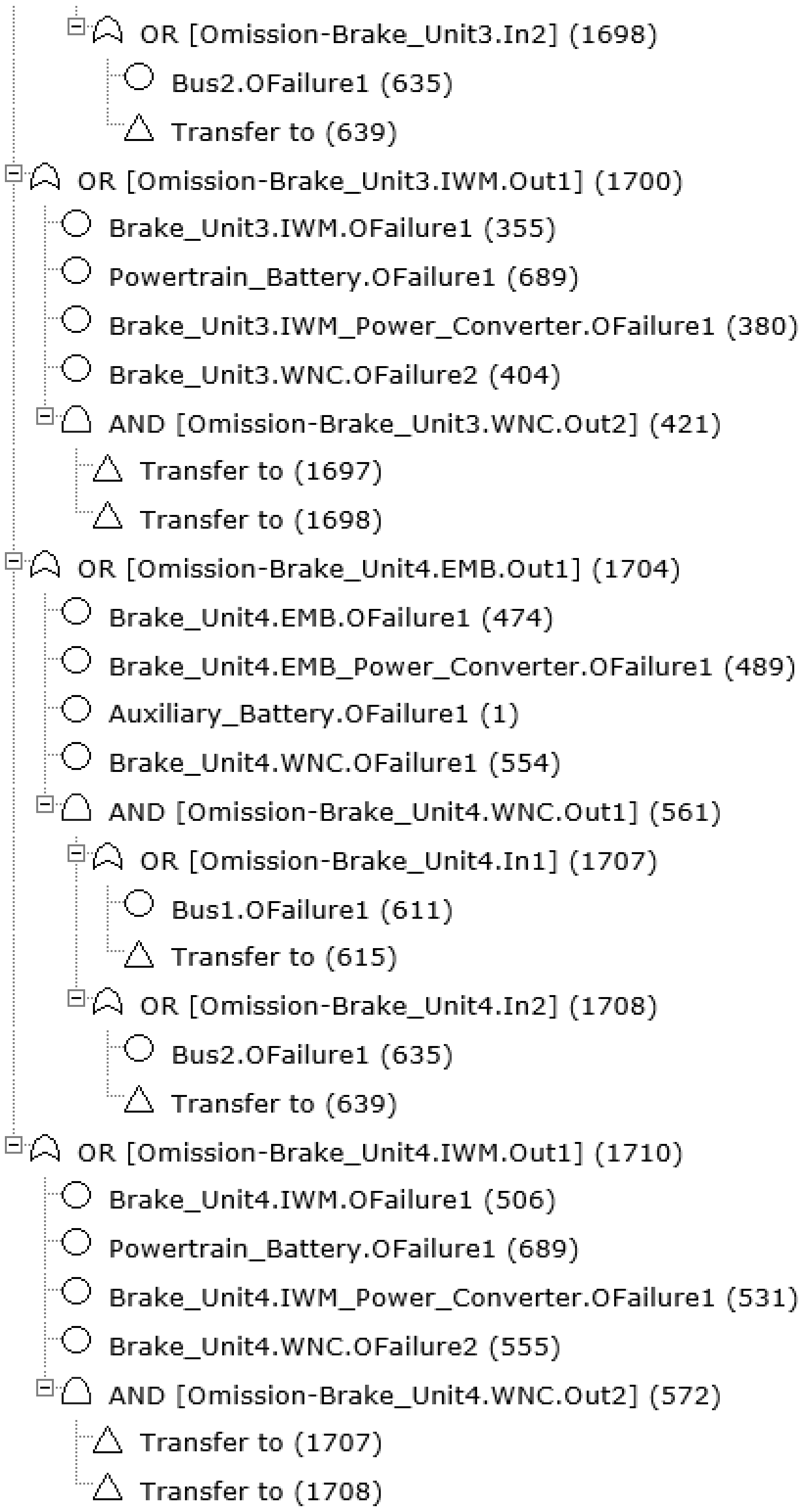
```
No braking in 4 wheels ()
  No braking in 4 wheels (1714)
    AND (1715)
      OR [Omission-Brake_Unit1.EMB.Out1] (1667)
        Brake_Unit1.EMB.OFailure1 (21)
        Brake_Unit1.EMB_Power_Converter.OFailure1 (36)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit1.WNC.OFailure1 (101)
        AND [Omission-Brake_Unit1.WNC.Out1] (108)
          OR [Omission-Brake_Unit1.In1] (1670)
            Bus1.OFailure1 (611)
            AND [Omission-Bus1.Out1] (615)
              OR [Omission-Electronic_Pedal.Out1] (1672)
                Electronic_Pedal.OFailure1 (659)
                Braking_Pedal.OFailure1 (683)
              OR [Omission-Electronic_Pedal.Out2] (1674)
                Electronic_Pedal.OFailure2 (660)
                Braking_Pedal.OFailure1 (683)
          OR [Omission-Brake_Unit1.In2] (1675)
            Bus2.OFailure1 (635)
            AND [Omission-Bus2.Out1] (639)
              Transfer to (1672)
              Transfer to (1674)
      OR [Omission-Brake_Unit1.IWM.Out1] (1679)
        Brake_Unit1.IWM.OFailure1 (53)
        Powertrain_Battery.OFailure1 (689)
        Brake_Unit1.IWM_Power_Converter.OFailure1 (78)
        Brake_Unit1.WNC.OFailure2 (102)
        AND [Omission-Brake_Unit1.WNC.Out2] (119)
```

Figure 76 – HBSM3 fault trees: “No braking in 4 wheels” hazard – part 1 of 3.



```
    AND [Omission-Brake_Unit1.WNC.Out2] (119)
        Transfer to (1670)
        Transfer to (1675)
OR [Omission-Brake_Unit2.EMB.Out1] (1684)
    Brake_Unit2.EMB.OFailure1 (172)
    Brake_Unit2.EMB_Power_Converter.OFailure1 (187)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit2.WNC.OFailure1 (252)
    AND [Omission-Brake_Unit2.WNC.Out1] (259)
        OR [Omission-Brake_Unit2.In1] (1687)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit2.In2] (1688)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit2.IWM.Out1] (1690)
    Brake_Unit2.IWM.OFailure1 (204)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit2.IWM_Power_Converter.OFailure1 (229)
    Brake_Unit2.WNC.OFailure2 (253)
    AND [Omission-Brake_Unit2.WNC.Out2] (270)
        Transfer to (1687)
        Transfer to (1688)
OR [Omission-Brake_Unit3.EMB.Out1] (1694)
    Brake_Unit3.EMB.OFailure1 (323)
    Brake_Unit3.EMB_Power_Converter.OFailure1 (338)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit3.WNC.OFailure1 (403)
    AND [Omission-Brake_Unit3.WNC.Out1] (410)
        OR [Omission-Brake_Unit3.In1] (1697)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit3.In2] (1698)
```

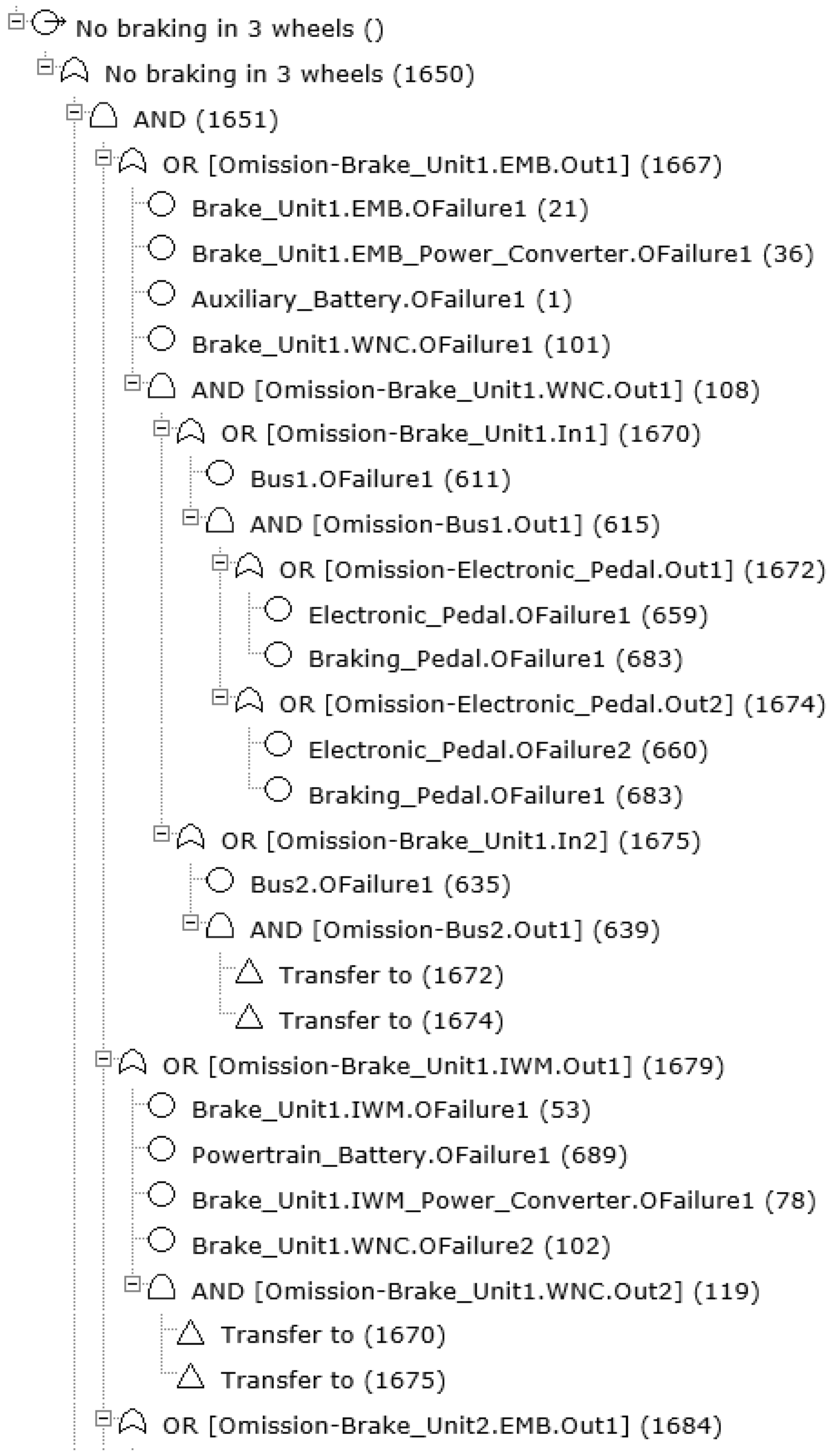
Figure 77 - HBSM3 fault trees: “No braking in 4 wheels” hazard – part 2 of 3



```
        OR [Omission-Brake_Unit3.In2] (1698)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit3.IWM.Out1] (1700)
    Brake_Unit3.IWM.OFailure1 (355)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit3.IWM_Power_Converter.OFailure1 (380)
    Brake_Unit3.WNC.OFailure2 (404)
    AND [Omission-Brake_Unit3.WNC.Out2] (421)
        Transfer to (1697)
        Transfer to (1698)
OR [Omission-Brake_Unit4.EMB.Out1] (1704)
    Brake_Unit4.EMB.OFailure1 (474)
    Brake_Unit4.EMB_Power_Converter.OFailure1 (489)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit4.WNC.OFailure1 (554)
    AND [Omission-Brake_Unit4.WNC.Out1] (561)
        OR [Omission-Brake_Unit4.In1] (1707)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit4.In2] (1708)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit4.IWM.Out1] (1710)
    Brake_Unit4.IWM.OFailure1 (506)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit4.IWM_Power_Converter.OFailure1 (531)
    Brake_Unit4.WNC.OFailure2 (555)
    AND [Omission-Brake_Unit4.WNC.Out2] (572)
        Transfer to (1707)
        Transfer to (1708)
```

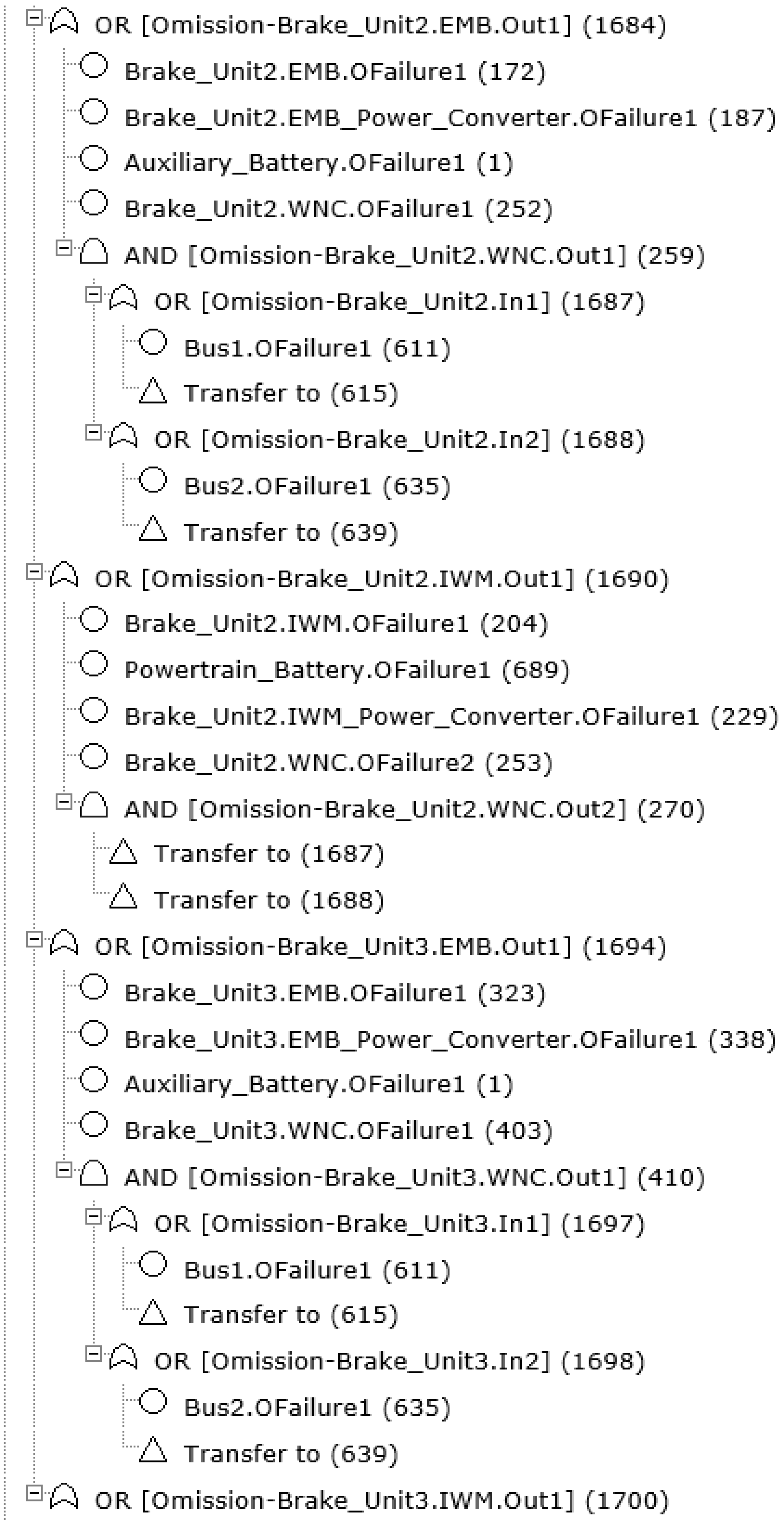
Figure 78 - HBSM3 fault trees: “No braking in 4 wheels” hazard – part 3 of 3.

No braking in 3 wheels



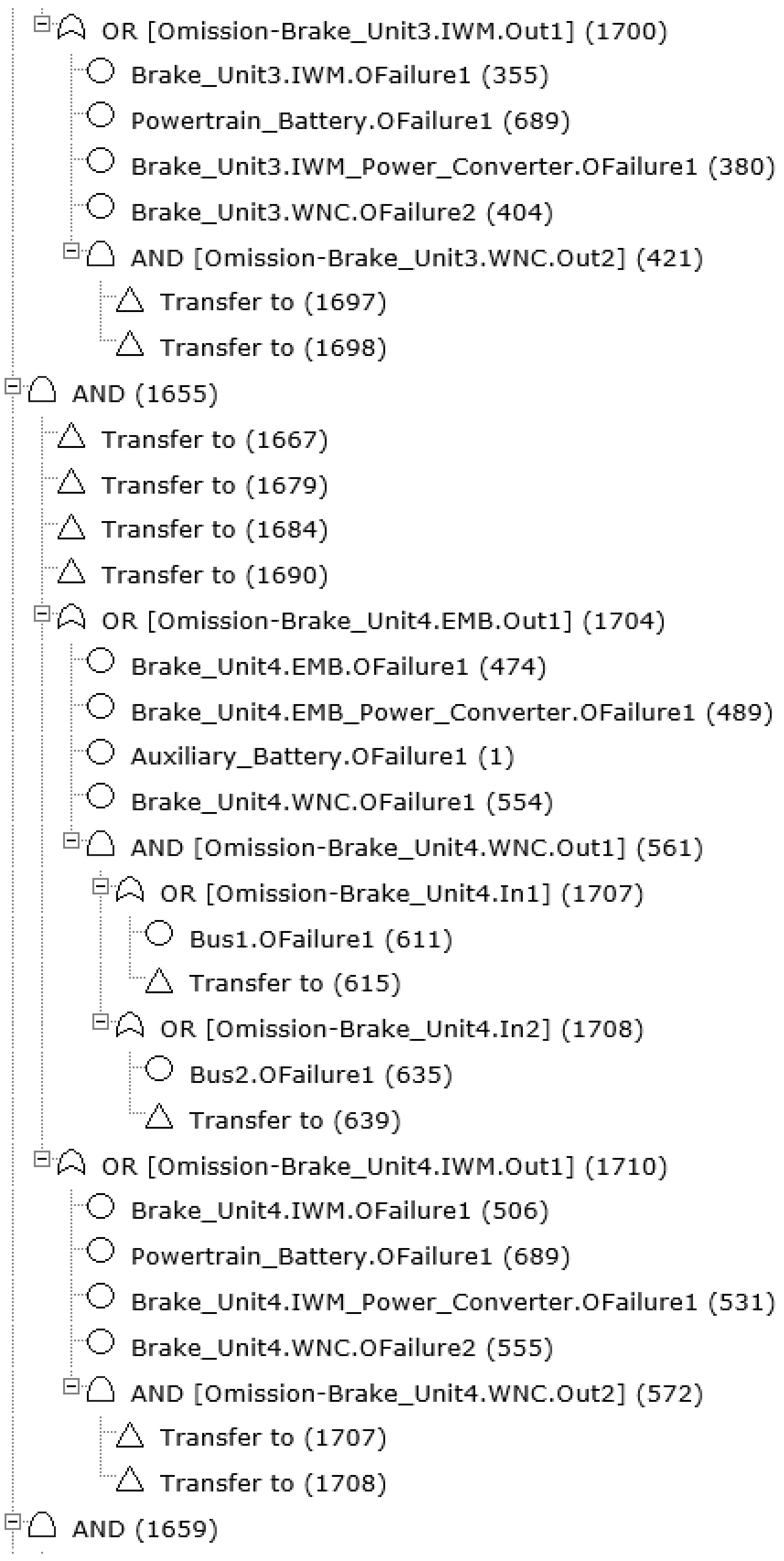
```
No braking in 3 wheels ()
  No braking in 3 wheels (1650)
    AND (1651)
      OR [Omission-Brake_Unit1.EMB.Out1] (1667)
        Brake_Unit1.EMB.OFailure1 (21)
        Brake_Unit1.EMB_Power_Converter.OFailure1 (36)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit1.WNC.OFailure1 (101)
        AND [Omission-Brake_Unit1.WNC.Out1] (108)
          OR [Omission-Brake_Unit1.In1] (1670)
            Bus1.OFailure1 (611)
            AND [Omission-Bus1.Out1] (615)
              OR [Omission-Electronic_Pedal.Out1] (1672)
                Electronic_Pedal.OFailure1 (659)
                Braking_Pedal.OFailure1 (683)
              OR [Omission-Electronic_Pedal.Out2] (1674)
                Electronic_Pedal.OFailure2 (660)
                Braking_Pedal.OFailure1 (683)
          OR [Omission-Brake_Unit1.In2] (1675)
            Bus2.OFailure1 (635)
            AND [Omission-Bus2.Out1] (639)
              Transfer to (1672)
              Transfer to (1674)
      OR [Omission-Brake_Unit1.IWM.Out1] (1679)
        Brake_Unit1.IWM.OFailure1 (53)
        Powertrain_Battery.OFailure1 (689)
        Brake_Unit1.IWM_Power_Converter.OFailure1 (78)
        Brake_Unit1.WNC.OFailure2 (102)
        AND [Omission-Brake_Unit1.WNC.Out2] (119)
          Transfer to (1670)
          Transfer to (1675)
      OR [Omission-Brake_Unit2.EMB.Out1] (1684)
```

Figure 79 - HBSM3 fault trees: “No braking in 3 wheels” hazard – part 1 of 4.



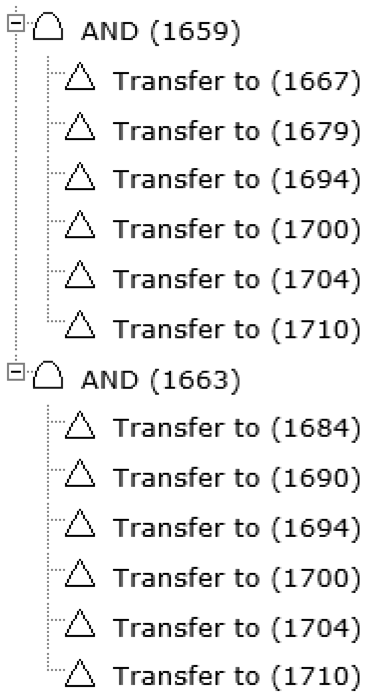
```
OR [Omission-Brake_Unit2.EMB.Out1] (1684)
    Brake_Unit2.EMB.OFailure1 (172)
    Brake_Unit2.EMB_Power_Converter.OFailure1 (187)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit2.WNC.OFailure1 (252)
    AND [Omission-Brake_Unit2.WNC.Out1] (259)
        OR [Omission-Brake_Unit2.In1] (1687)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit2.In2] (1688)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit2.IWM.Out1] (1690)
    Brake_Unit2.IWM.OFailure1 (204)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit2.IWM_Power_Converter.OFailure1 (229)
    Brake_Unit2.WNC.OFailure2 (253)
    AND [Omission-Brake_Unit2.WNC.Out2] (270)
        Transfer to (1687)
        Transfer to (1688)
OR [Omission-Brake_Unit3.EMB.Out1] (1694)
    Brake_Unit3.EMB.OFailure1 (323)
    Brake_Unit3.EMB_Power_Converter.OFailure1 (338)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit3.WNC.OFailure1 (403)
    AND [Omission-Brake_Unit3.WNC.Out1] (410)
        OR [Omission-Brake_Unit3.In1] (1697)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit3.In2] (1698)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit3.IWM.Out1] (1700)
```

Figure 80 – HBSM3 fault trees: "No braking in 3 wheels" hazard – part 2 of 4.



```
  OR [Omission-Brake_Unit3.IWM.Out1] (1700)
    Brake_Unit3.IWM.OFailure1 (355)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit3.IWM_Power_Converter.OFailure1 (380)
    Brake_Unit3.WNC.OFailure2 (404)
    AND [Omission-Brake_Unit3.WNC.Out2] (421)
      Transfer to (1697)
      Transfer to (1698)
AND (1655)
  Transfer to (1667)
  Transfer to (1679)
  Transfer to (1684)
  Transfer to (1690)
  OR [Omission-Brake_Unit4.EMB.Out1] (1704)
    Brake_Unit4.EMB.OFailure1 (474)
    Brake_Unit4.EMB_Power_Converter.OFailure1 (489)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit4.WNC.OFailure1 (554)
    AND [Omission-Brake_Unit4.WNC.Out1] (561)
      OR [Omission-Brake_Unit4.In1] (1707)
        Bus1.OFailure1 (611)
        Transfer to (615)
      OR [Omission-Brake_Unit4.In2] (1708)
        Bus2.OFailure1 (635)
        Transfer to (639)
  OR [Omission-Brake_Unit4.IWM.Out1] (1710)
    Brake_Unit4.IWM.OFailure1 (506)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit4.IWM_Power_Converter.OFailure1 (531)
    Brake_Unit4.WNC.OFailure2 (555)
    AND [Omission-Brake_Unit4.WNC.Out2] (572)
      Transfer to (1707)
      Transfer to (1708)
AND (1659)
```

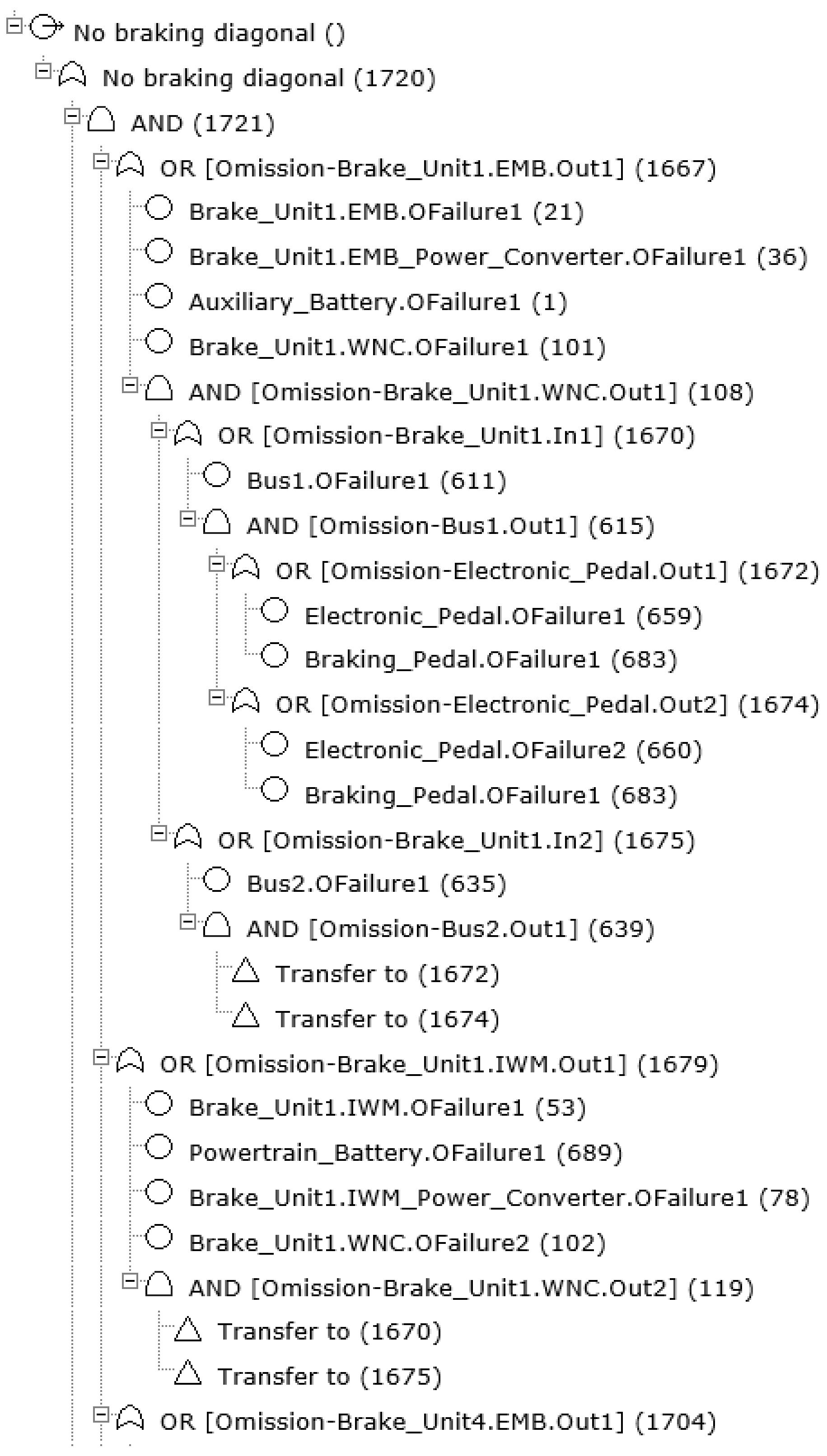
Figure 81 – HBSM3 fault trees: "No braking in 3 wheels" hazard – part 3 of 4.



```
AND (1659)
    Transfer to (1667)
    Transfer to (1679)
    Transfer to (1694)
    Transfer to (1700)
    Transfer to (1704)
    Transfer to (1710)
AND (1663)
    Transfer to (1684)
    Transfer to (1690)
    Transfer to (1694)
    Transfer to (1700)
    Transfer to (1704)
    Transfer to (1710)
```

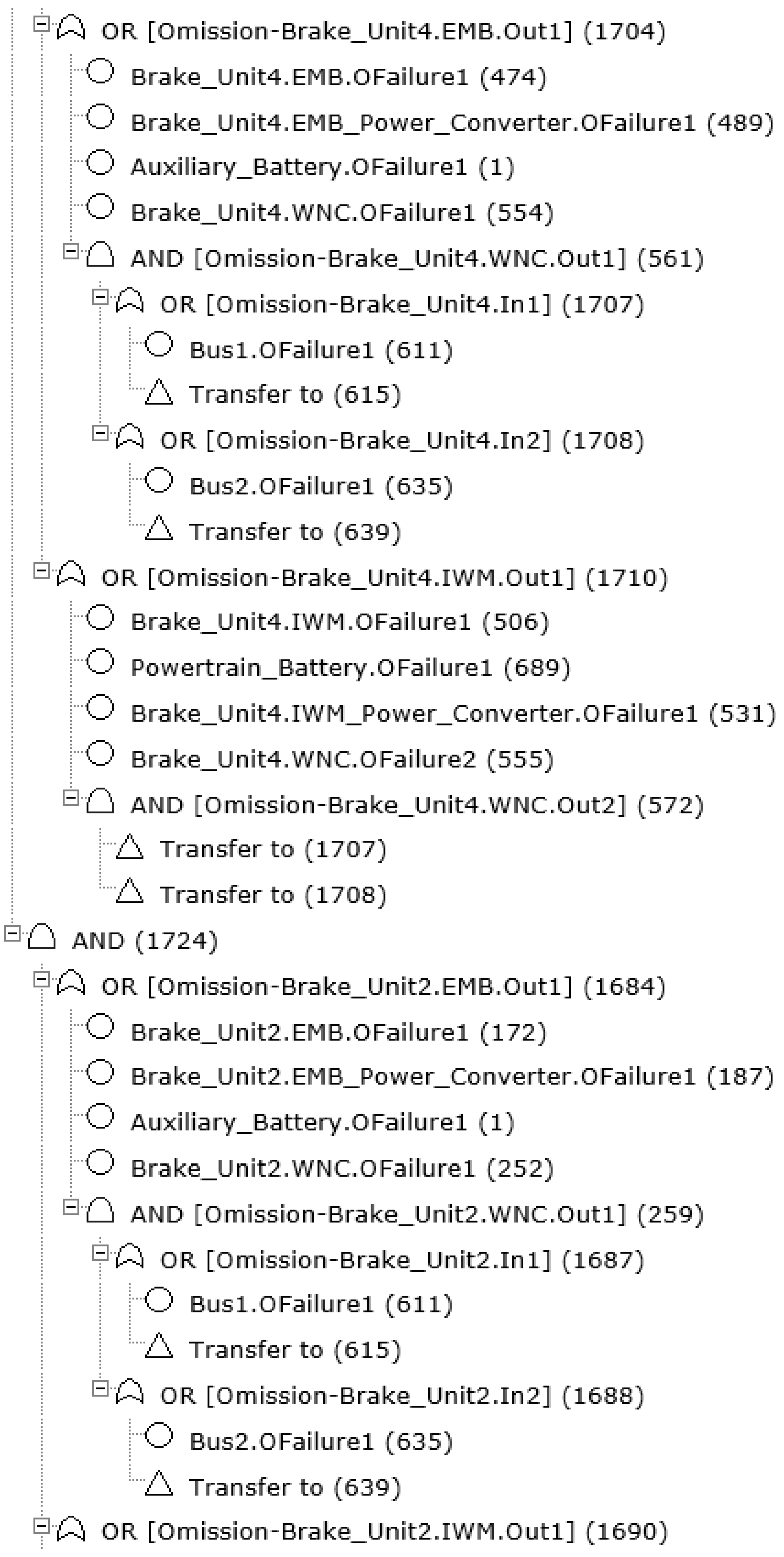
Figure 82 – HBSM3 fault trees: “No braking in 3 wheels” hazard – part 4 of 4.

No braking diagonal



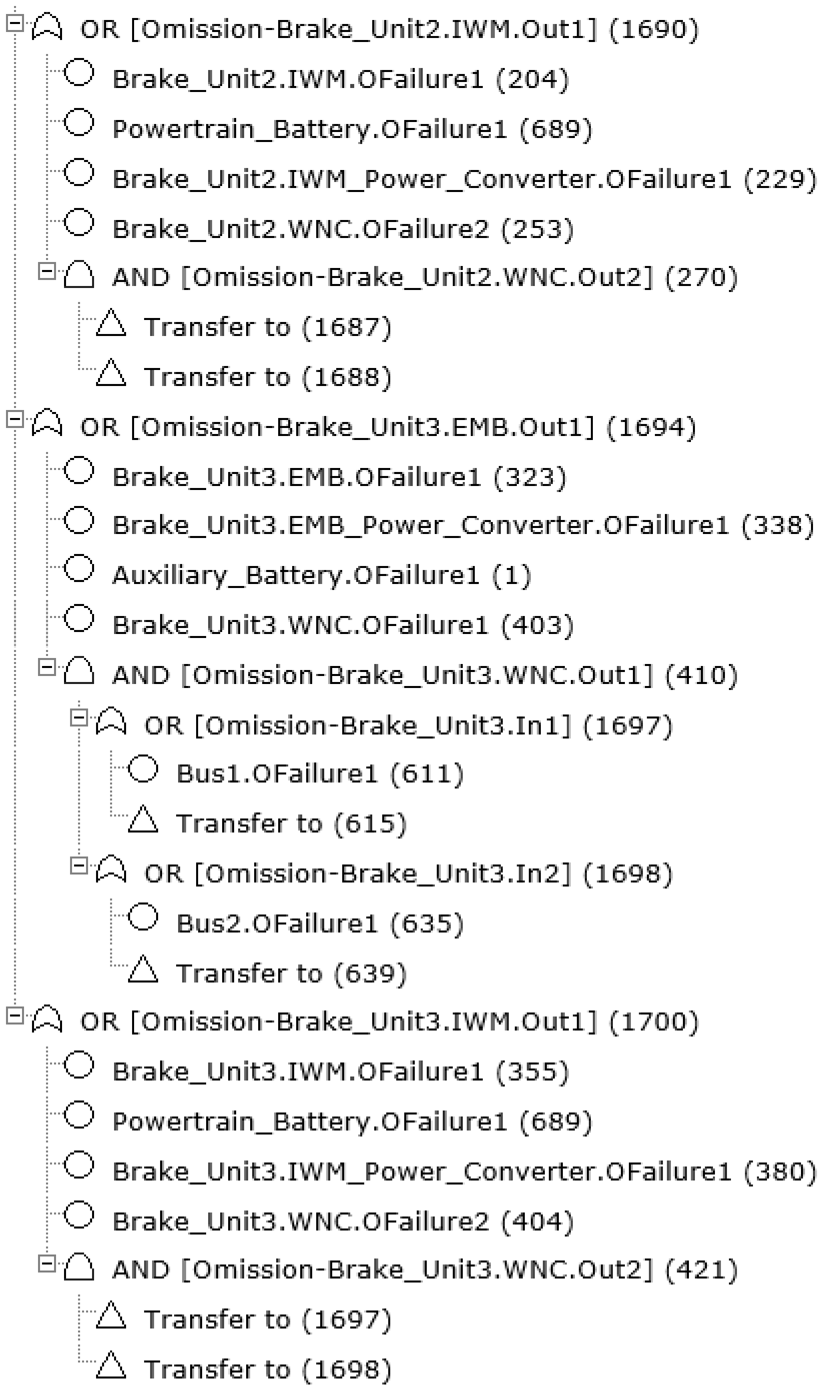
```
No braking diagonal ()
  No braking diagonal (1720)
    AND (1721)
      OR [Omission-Brake_Unit1.EMB.Out1] (1667)
        Brake_Unit1.EMB.OFailure1 (21)
        Brake_Unit1.EMB_Power_Converter.OFailure1 (36)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit1.WNC.OFailure1 (101)
        AND [Omission-Brake_Unit1.WNC.Out1] (108)
          OR [Omission-Brake_Unit1.In1] (1670)
            Bus1.OFailure1 (611)
            AND [Omission-Bus1.Out1] (615)
              OR [Omission-Electronic_Pedal.Out1] (1672)
                Electronic_Pedal.OFailure1 (659)
                Braking_Pedal.OFailure1 (683)
              OR [Omission-Electronic_Pedal.Out2] (1674)
                Electronic_Pedal.OFailure2 (660)
                Braking_Pedal.OFailure1 (683)
          OR [Omission-Brake_Unit1.In2] (1675)
            Bus2.OFailure1 (635)
            AND [Omission-Bus2.Out1] (639)
              Transfer to (1672)
              Transfer to (1674)
      OR [Omission-Brake_Unit1.IWM.Out1] (1679)
        Brake_Unit1.IWM.OFailure1 (53)
        Powertrain_Battery.OFailure1 (689)
        Brake_Unit1.IWM_Power_Converter.OFailure1 (78)
        Brake_Unit1.WNC.OFailure2 (102)
        AND [Omission-Brake_Unit1.WNC.Out2] (119)
          Transfer to (1670)
          Transfer to (1675)
      OR [Omission-Brake_Unit4.EMB.Out1] (1704)
```

Figure 83 - HBSM3 fault trees: “No braking diagonal” hazard – part 1 of 3.



```
    OR [Omission-Brake_Unit4.EMB.Out1] (1704)
        Brake_Unit4.EMB.OFailure1 (474)
        Brake_Unit4.EMB_Power_Converter.OFailure1 (489)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit4.WNC.OFailure1 (554)
        AND [Omission-Brake_Unit4.WNC.Out1] (561)
            OR [Omission-Brake_Unit4.In1] (1707)
                Bus1.OFailure1 (611)
                Transfer to (615)
            OR [Omission-Brake_Unit4.In2] (1708)
                Bus2.OFailure1 (635)
                Transfer to (639)
    OR [Omission-Brake_Unit4.IWM.Out1] (1710)
        Brake_Unit4.IWM.OFailure1 (506)
        Powertrain_Battery.OFailure1 (689)
        Brake_Unit4.IWM_Power_Converter.OFailure1 (531)
        Brake_Unit4.WNC.OFailure2 (555)
        AND [Omission-Brake_Unit4.WNC.Out2] (572)
            Transfer to (1707)
            Transfer to (1708)
AND (1724)
    OR [Omission-Brake_Unit2.EMB.Out1] (1684)
        Brake_Unit2.EMB.OFailure1 (172)
        Brake_Unit2.EMB_Power_Converter.OFailure1 (187)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit2.WNC.OFailure1 (252)
        AND [Omission-Brake_Unit2.WNC.Out1] (259)
            OR [Omission-Brake_Unit2.In1] (1687)
                Bus1.OFailure1 (611)
                Transfer to (615)
            OR [Omission-Brake_Unit2.In2] (1688)
                Bus2.OFailure1 (635)
                Transfer to (639)
    OR [Omission-Brake_Unit2.IWM.Out1] (1690)
```

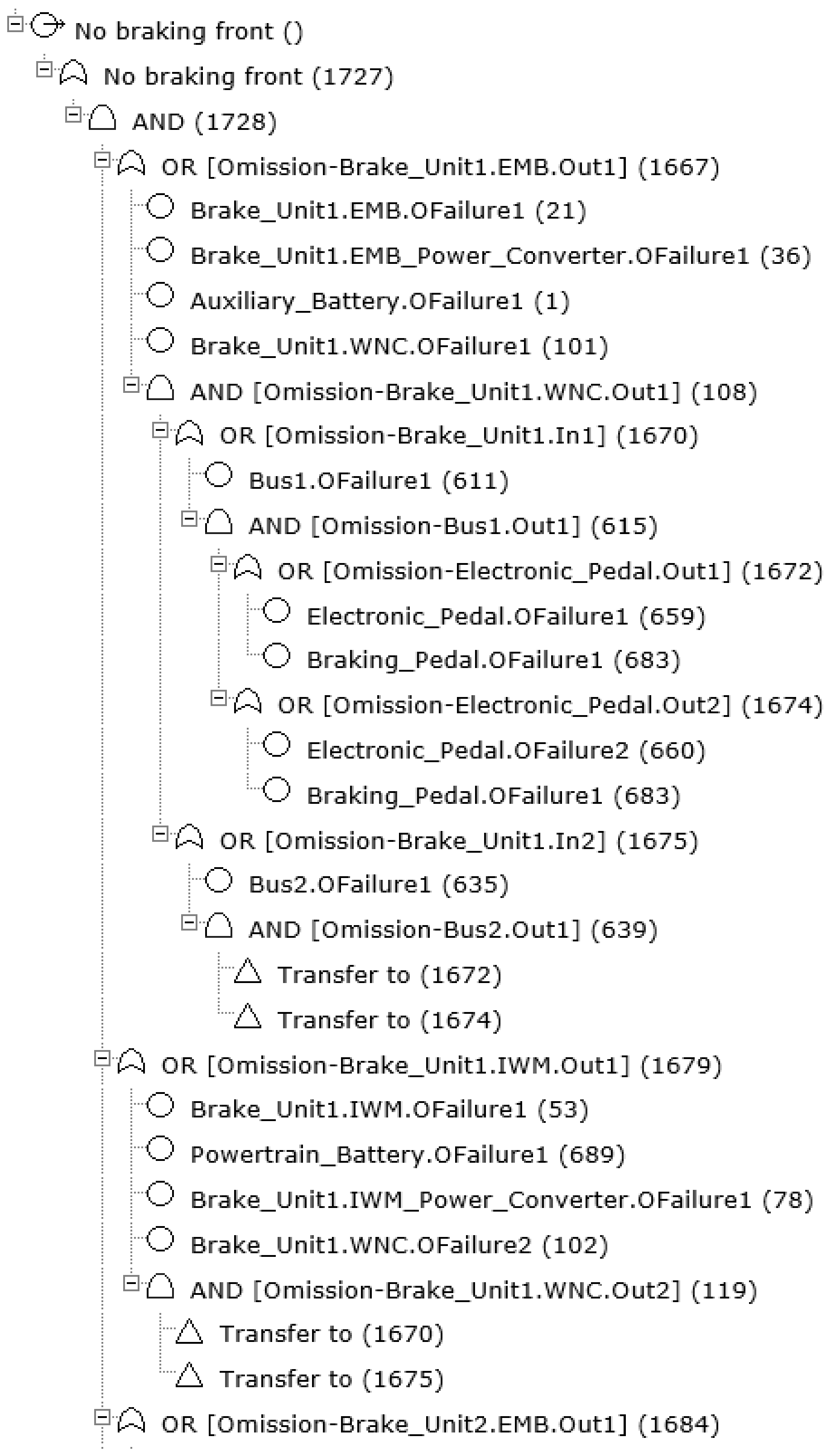
Figure 84 – HBSM3 fault trees: "No braking diagonal" hazard – part 2 of 3.



```
OR [Omission-Brake_Unit2.IWM.Out1] (1690)
    Brake_Unit2.IWM.OFailure1 (204)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit2.IWM_Power_Converter.OFailure1 (229)
    Brake_Unit2.WNC.OFailure2 (253)
    AND [Omission-Brake_Unit2.WNC.Out2] (270)
        Transfer to (1687)
        Transfer to (1688)
OR [Omission-Brake_Unit3.EMB.Out1] (1694)
    Brake_Unit3.EMB.OFailure1 (323)
    Brake_Unit3.EMB_Power_Converter.OFailure1 (338)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit3.WNC.OFailure1 (403)
    AND [Omission-Brake_Unit3.WNC.Out1] (410)
        OR [Omission-Brake_Unit3.In1] (1697)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit3.In2] (1698)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit3.IWM.Out1] (1700)
    Brake_Unit3.IWM.OFailure1 (355)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit3.IWM_Power_Converter.OFailure1 (380)
    Brake_Unit3.WNC.OFailure2 (404)
    AND [Omission-Brake_Unit3.WNC.Out2] (421)
        Transfer to (1697)
        Transfer to (1698)
```

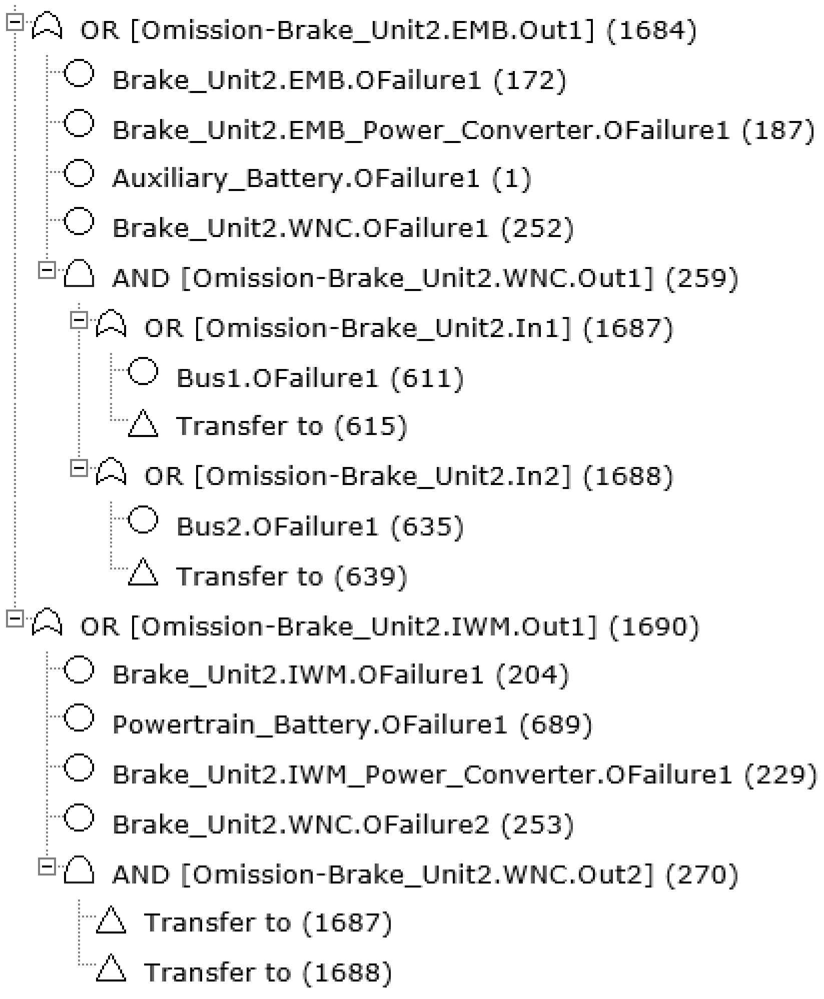
Figure 85 – HBSM3 fault trees: “No braking diagonal” hazard – part 3 of 3.

No braking front



```
No braking front ()
  No braking front (1727)
    AND (1728)
      OR [Omission-Brake_Unit1.EMB.Out1] (1667)
        Brake_Unit1.EMB.OFailure1 (21)
        Brake_Unit1.EMB_Power_Converter.OFailure1 (36)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit1.WNC.OFailure1 (101)
        AND [Omission-Brake_Unit1.WNC.Out1] (108)
          OR [Omission-Brake_Unit1.In1] (1670)
            Bus1.OFailure1 (611)
            AND [Omission-Bus1.Out1] (615)
              OR [Omission-Electronic_Pedal.Out1] (1672)
                Electronic_Pedal.OFailure1 (659)
                Braking_Pedal.OFailure1 (683)
              OR [Omission-Electronic_Pedal.Out2] (1674)
                Electronic_Pedal.OFailure2 (660)
                Braking_Pedal.OFailure1 (683)
          OR [Omission-Brake_Unit1.In2] (1675)
            Bus2.OFailure1 (635)
            AND [Omission-Bus2.Out1] (639)
              Transfer to (1672)
              Transfer to (1674)
      OR [Omission-Brake_Unit1.IWM.Out1] (1679)
        Brake_Unit1.IWM.OFailure1 (53)
        Powertrain_Battery.OFailure1 (689)
        Brake_Unit1.IWM_Power_Converter.OFailure1 (78)
        Brake_Unit1.WNC.OFailure2 (102)
        AND [Omission-Brake_Unit1.WNC.Out2] (119)
          Transfer to (1670)
          Transfer to (1675)
      OR [Omission-Brake_Unit2.EMB.Out1] (1684)
```

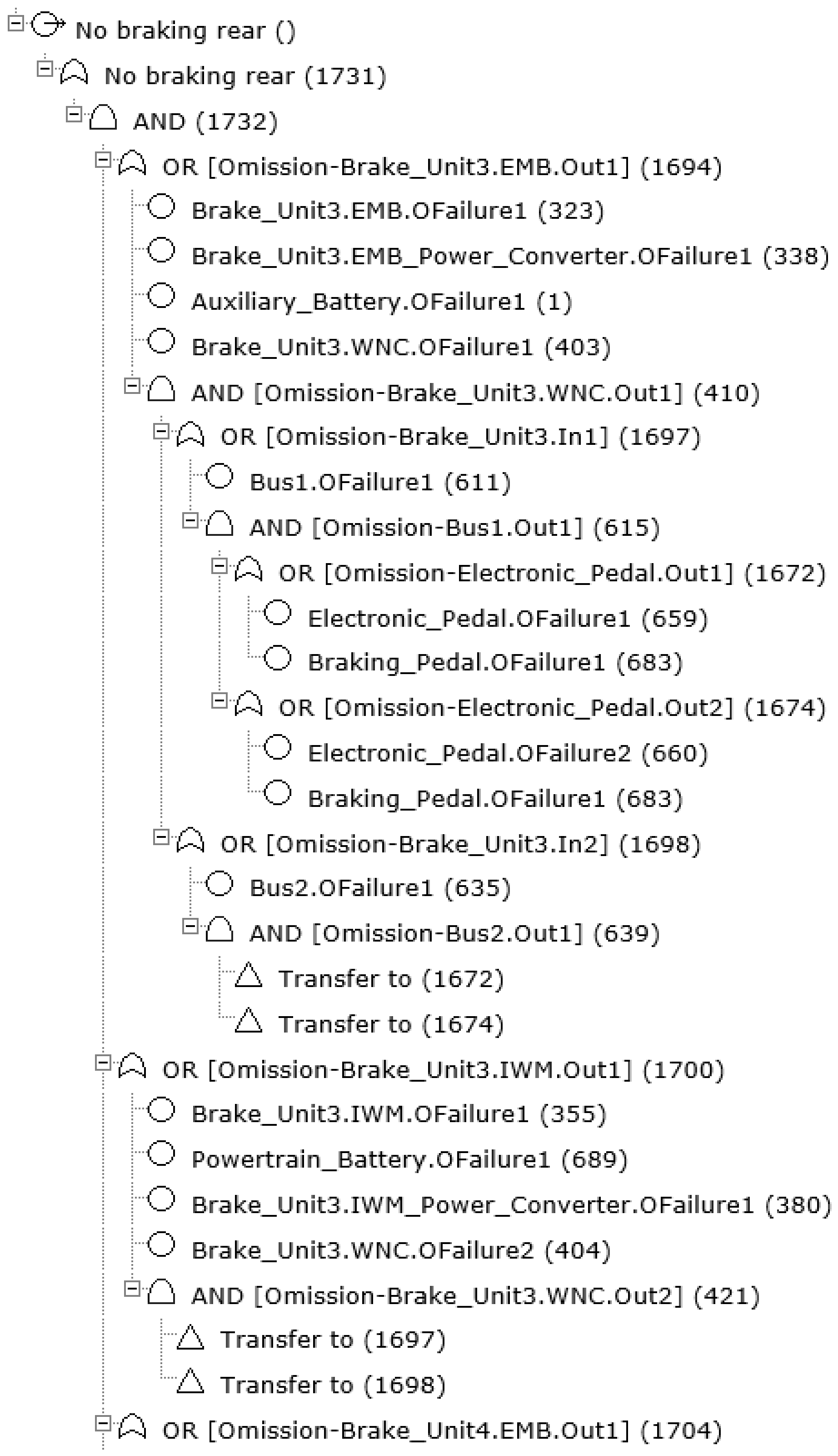
Figure 86 - HBSM3 fault trees: “No braking front” hazard – part 1 of 2.



```
OR [Omission-Brake_Unit2.EMB.Out1] (1684)
    Brake_Unit2.EMB.OFailure1 (172)
    Brake_Unit2.EMB_Power_Converter.OFailure1 (187)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit2.WNC.OFailure1 (252)
    AND [Omission-Brake_Unit2.WNC.Out1] (259)
        OR [Omission-Brake_Unit2.In1] (1687)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit2.In2] (1688)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit2.IWM.Out1] (1690)
    Brake_Unit2.IWM.OFailure1 (204)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit2.IWM_Power_Converter.OFailure1 (229)
    Brake_Unit2.WNC.OFailure2 (253)
    AND [Omission-Brake_Unit2.WNC.Out2] (270)
        Transfer to (1687)
        Transfer to (1688)
```

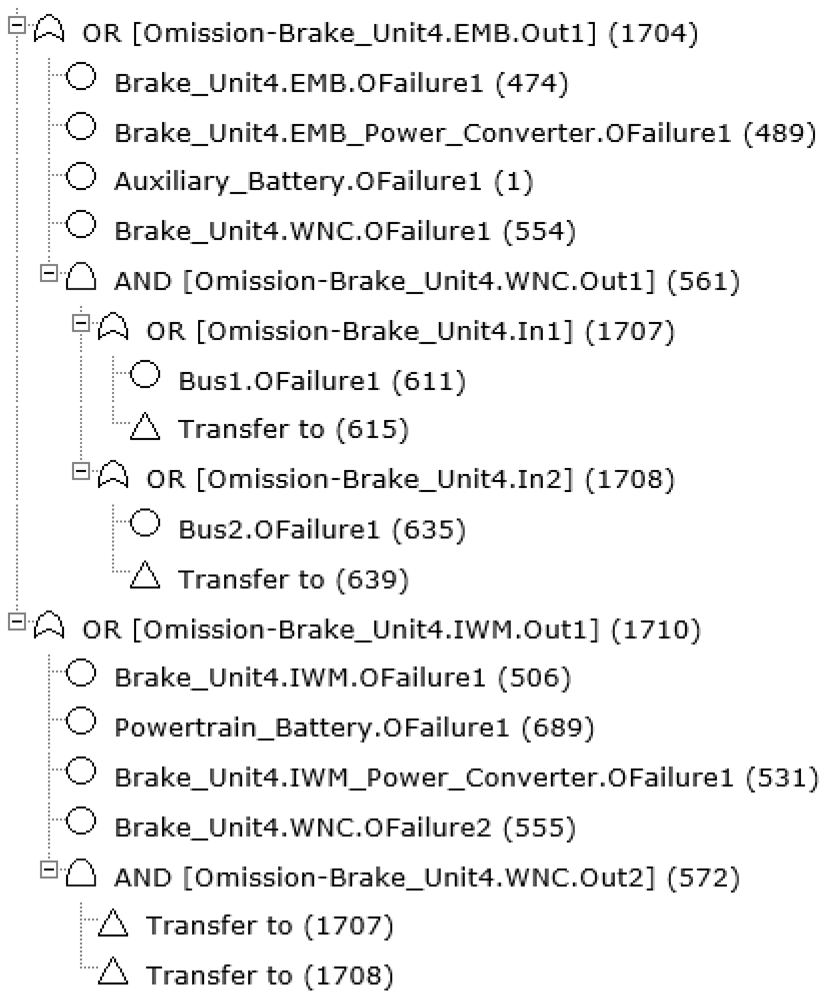
Figure 87 - HBSM3 fault trees: “No braking front” hazard – part 2 of 2.

No braking rear



```
No braking rear ()
  No braking rear (1731)
    AND (1732)
      OR [Omission-Brake_Unit3.EMB.Out1] (1694)
        Brake_Unit3.EMB.OFailure1 (323)
        Brake_Unit3.EMB_Power_Converter.OFailure1 (338)
        Auxiliary_Battery.OFailure1 (1)
        Brake_Unit3.WNC.OFailure1 (403)
        AND [Omission-Brake_Unit3.WNC.Out1] (410)
          OR [Omission-Brake_Unit3.In1] (1697)
            Bus1.OFailure1 (611)
            AND [Omission-Bus1.Out1] (615)
              OR [Omission-Electronic_Pedal.Out1] (1672)
                Electronic_Pedal.OFailure1 (659)
                Braking_Pedal.OFailure1 (683)
              OR [Omission-Electronic_Pedal.Out2] (1674)
                Electronic_Pedal.OFailure2 (660)
                Braking_Pedal.OFailure1 (683)
          OR [Omission-Brake_Unit3.In2] (1698)
            Bus2.OFailure1 (635)
            AND [Omission-Bus2.Out1] (639)
              Transfer to (1672)
              Transfer to (1674)
      OR [Omission-Brake_Unit3.IWM.Out1] (1700)
        Brake_Unit3.IWM.OFailure1 (355)
        Powertrain_Battery.OFailure1 (689)
        Brake_Unit3.IWM_Power_Converter.OFailure1 (380)
        Brake_Unit3.WNC.OFailure2 (404)
        AND [Omission-Brake_Unit3.WNC.Out2] (421)
          Transfer to (1697)
          Transfer to (1698)
      OR [Omission-Brake_Unit4.EMB.Out1] (1704)
```

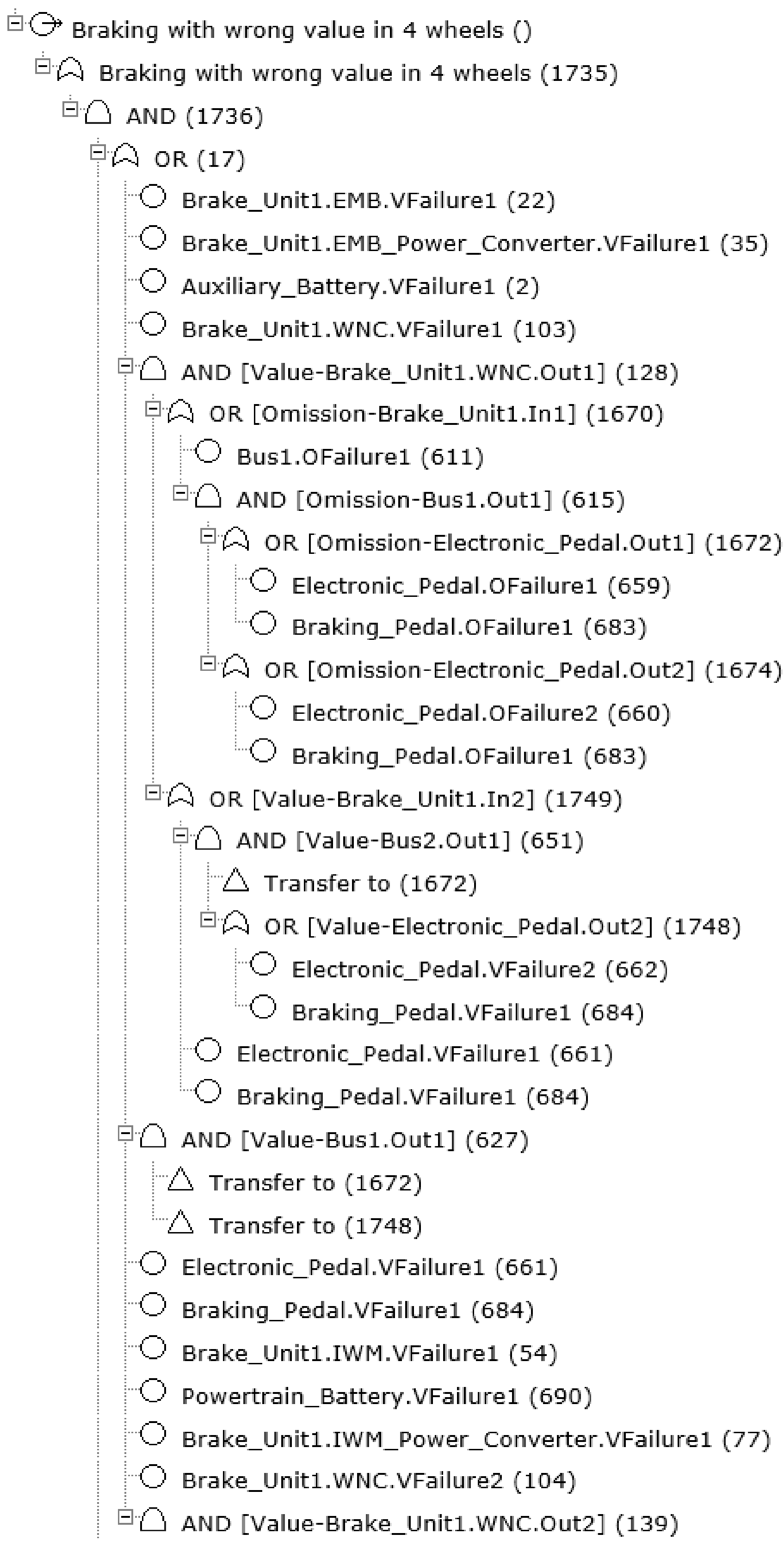
Figure 88 - HBSM3 fault trees: "No braking rear" hazard – part 1 of 2.



```
OR [Omission-Brake_Unit4.EMB.Out1] (1704)
    Brake_Unit4.EMB.OFailure1 (474)
    Brake_Unit4.EMB_Power_Converter.OFailure1 (489)
    Auxiliary_Battery.OFailure1 (1)
    Brake_Unit4.WNC.OFailure1 (554)
    AND [Omission-Brake_Unit4.WNC.Out1] (561)
        OR [Omission-Brake_Unit4.In1] (1707)
            Bus1.OFailure1 (611)
            Transfer to (615)
        OR [Omission-Brake_Unit4.In2] (1708)
            Bus2.OFailure1 (635)
            Transfer to (639)
OR [Omission-Brake_Unit4.IWM.Out1] (1710)
    Brake_Unit4.IWM.OFailure1 (506)
    Powertrain_Battery.OFailure1 (689)
    Brake_Unit4.IWM_Power_Converter.OFailure1 (531)
    Brake_Unit4.WNC.OFailure2 (555)
    AND [Omission-Brake_Unit4.WNC.Out2] (572)
        Transfer to (1707)
        Transfer to (1708)
```

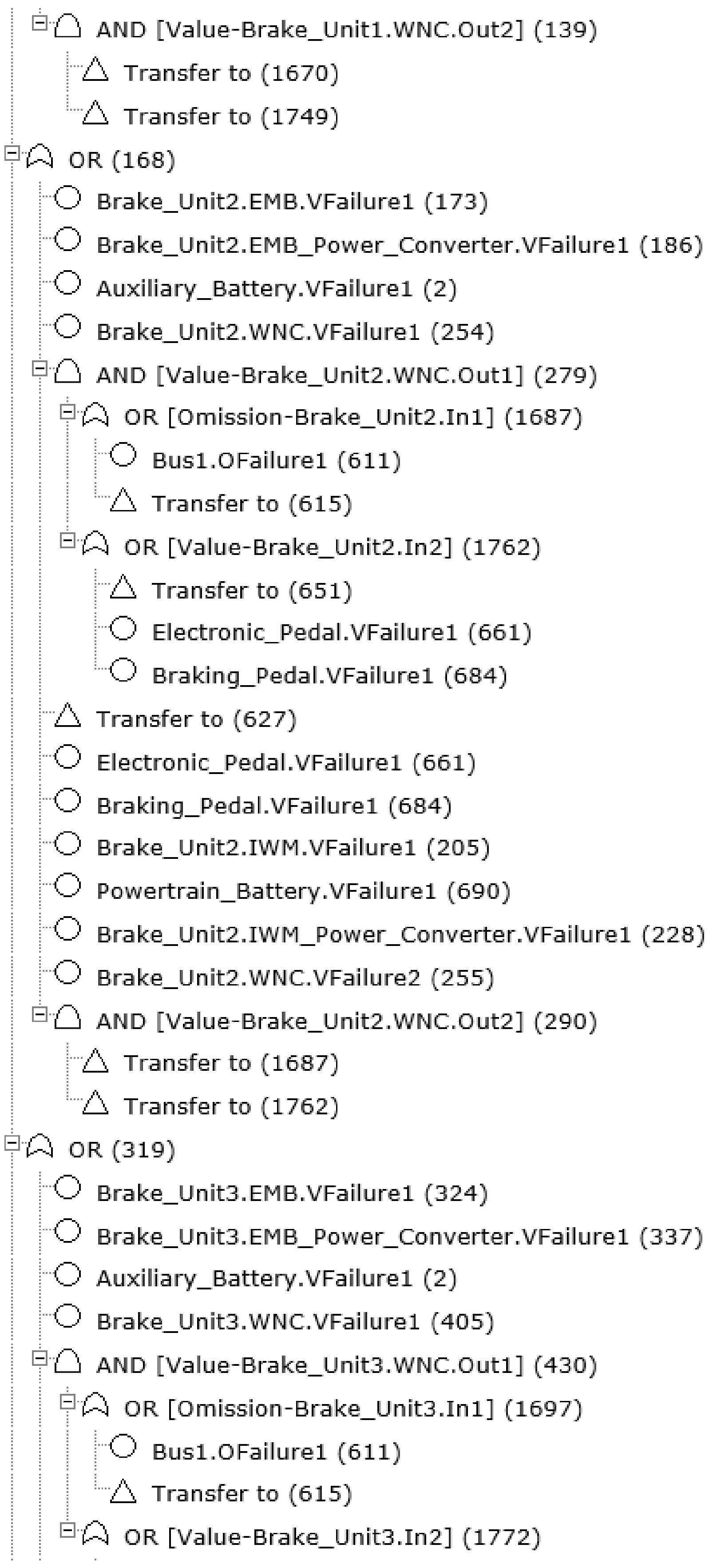
Figure 89 - HBSM3 fault trees: “No braking rear” hazard – part 2 of 2.

Braking with wrong value in 4 wheels



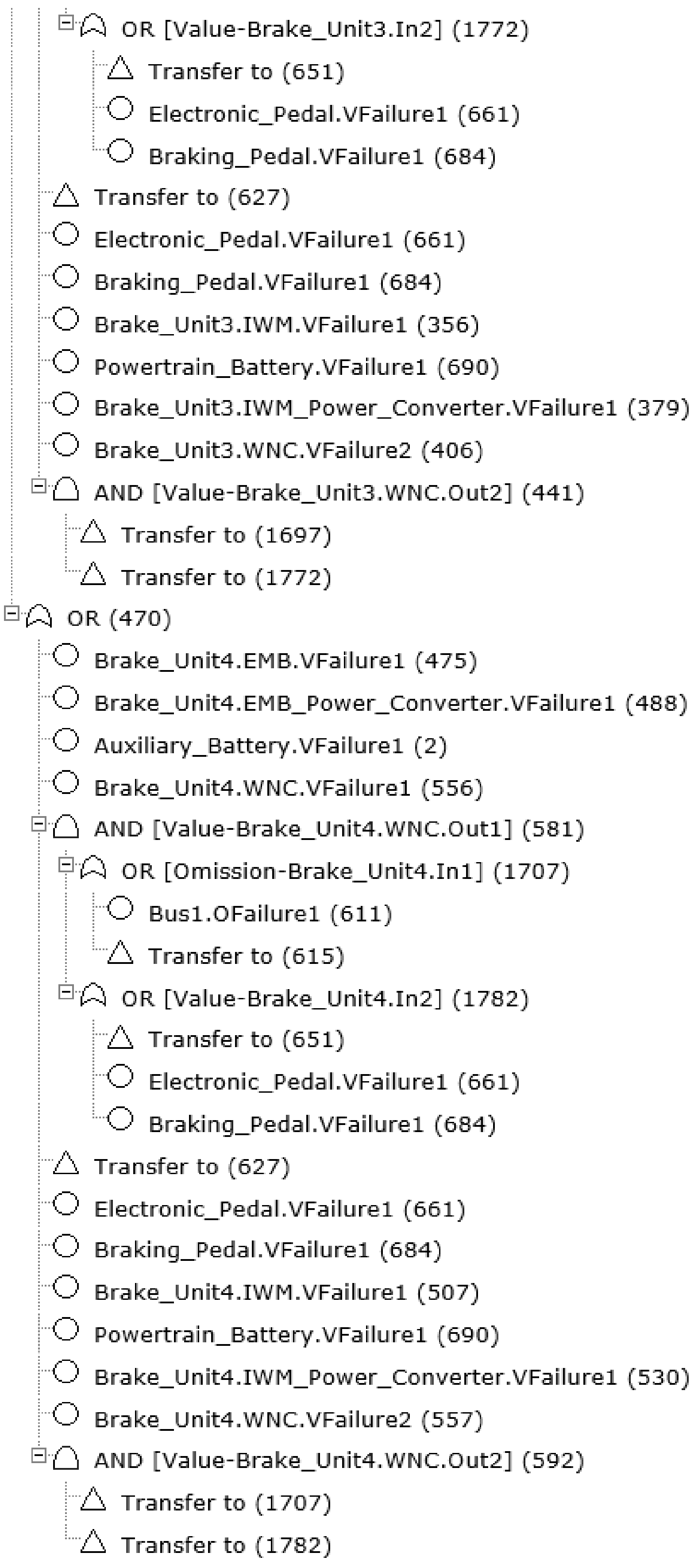
```
Braking with wrong value in 4 wheels ()
  Braking with wrong value in 4 wheels (1735)
    AND (1736)
      OR (17)
        Brake_Unit1.EMB.VFailure1 (22)
        Brake_Unit1.EMB_Power_Converter.VFailure1 (35)
        Auxiliary_Battery.VFailure1 (2)
        Brake_Unit1.WNC.VFailure1 (103)
        AND [Value-Brake_Unit1.WNC.Out1] (128)
          OR [Omission-Brake_Unit1.In1] (1670)
            Bus1.OFailure1 (611)
            AND [Omission-Bus1.Out1] (615)
              OR [Omission-Electronic_Pedal.Out1] (1672)
                Electronic_Pedal.OFailure1 (659)
                Braking_Pedal.OFailure1 (683)
              OR [Omission-Electronic_Pedal.Out2] (1674)
                Electronic_Pedal.OFailure2 (660)
                Braking_Pedal.OFailure1 (683)
          OR [Value-Brake_Unit1.In2] (1749)
            AND [Value-Bus2.Out1] (651)
              Transfer to (1672)
              OR [Value-Electronic_Pedal.Out2] (1748)
                Electronic_Pedal.VFailure2 (662)
                Braking_Pedal.VFailure1 (684)
            Electronic_Pedal.VFailure1 (661)
            Braking_Pedal.VFailure1 (684)
        AND [Value-Bus1.Out1] (627)
          Transfer to (1672)
          Transfer to (1748)
        Electronic_Pedal.VFailure1 (661)
        Braking_Pedal.VFailure1 (684)
        Brake_Unit1.IWM.VFailure1 (54)
        Powertrain_Battery.VFailure1 (690)
        Brake_Unit1.IWM_Power_Converter.VFailure1 (77)
        Brake_Unit1.WNC.VFailure2 (104)
        AND [Value-Brake_Unit1.WNC.Out2] (139)
```

Figure 90 - HBSM3 fault trees: “Braking with wrong value in 4 wheels” hazard – part 1 of 3.



```
  AND [Value-Brake_Unit1.WNC.Out2] (139)
    Transfer to (1670)
    Transfer to (1749)
OR (168)
  Brake_Unit2.EMB.VFailure1 (173)
  Brake_Unit2.EMB_Power_Converter.VFailure1 (186)
  Auxiliary_Battery.VFailure1 (2)
  Brake_Unit2.WNC.VFailure1 (254)
  AND [Value-Brake_Unit2.WNC.Out1] (279)
    OR [Omission-Brake_Unit2.In1] (1687)
      Bus1.OFailure1 (611)
      Transfer to (615)
    OR [Value-Brake_Unit2.In2] (1762)
      Transfer to (651)
      Electronic_Pedal.VFailure1 (661)
      Braking_Pedal.VFailure1 (684)
  Transfer to (627)
  Electronic_Pedal.VFailure1 (661)
  Braking_Pedal.VFailure1 (684)
  Brake_Unit2.IWM.VFailure1 (205)
  Powertrain_Battery.VFailure1 (690)
  Brake_Unit2.IWM_Power_Converter.VFailure1 (228)
  Brake_Unit2.WNC.VFailure2 (255)
  AND [Value-Brake_Unit2.WNC.Out2] (290)
    Transfer to (1687)
    Transfer to (1762)
OR (319)
  Brake_Unit3.EMB.VFailure1 (324)
  Brake_Unit3.EMB_Power_Converter.VFailure1 (337)
  Auxiliary_Battery.VFailure1 (2)
  Brake_Unit3.WNC.VFailure1 (405)
  AND [Value-Brake_Unit3.WNC.Out1] (430)
    OR [Omission-Brake_Unit3.In1] (1697)
      Bus1.OFailure1 (611)
      Transfer to (615)
    OR [Value-Brake_Unit3.In2] (1772)
```

Figure 91 – HBSM3 fault trees: "Braking with wrong value in 4 wheels" hazard – part 2 of 3.



```
    OR [Value-Brake_Unit3.In2] (1772)
      Transfer to (651)
      Electronic_Pedal.VFailure1 (661)
      Braking_Pedal.VFailure1 (684)
  Transfer to (627)
  Electronic_Pedal.VFailure1 (661)
  Braking_Pedal.VFailure1 (684)
  Brake_Unit3.IWM.VFailure1 (356)
  Powertrain_Battery.VFailure1 (690)
  Brake_Unit3.IWM_Power_Converter.VFailure1 (379)
  Brake_Unit3.WNC.VFailure2 (406)
  AND [Value-Brake_Unit3.WNC.Out2] (441)
    Transfer to (1697)
    Transfer to (1772)
OR (470)
  Brake_Unit4.EMB.VFailure1 (475)
  Brake_Unit4.EMB_Power_Converter.VFailure1 (488)
  Auxiliary_Battery.VFailure1 (2)
  Brake_Unit4.WNC.VFailure1 (556)
  AND [Value-Brake_Unit4.WNC.Out1] (581)
    OR [Omission-Brake_Unit4.In1] (1707)
      Bus1.OFailure1 (611)
      Transfer to (615)
    OR [Value-Brake_Unit4.In2] (1782)
      Transfer to (651)
      Electronic_Pedal.VFailure1 (661)
      Braking_Pedal.VFailure1 (684)
  Transfer to (627)
  Electronic_Pedal.VFailure1 (661)
  Braking_Pedal.VFailure1 (684)
  Brake_Unit4.IWM.VFailure1 (507)
  Powertrain_Battery.VFailure1 (690)
  Brake_Unit4.IWM_Power_Converter.VFailure1 (530)
  Brake_Unit4.WNC.VFailure2 (557)
  AND [Value-Brake_Unit4.WNC.Out2] (592)
    Transfer to (1707)
    Transfer to (1782)
```

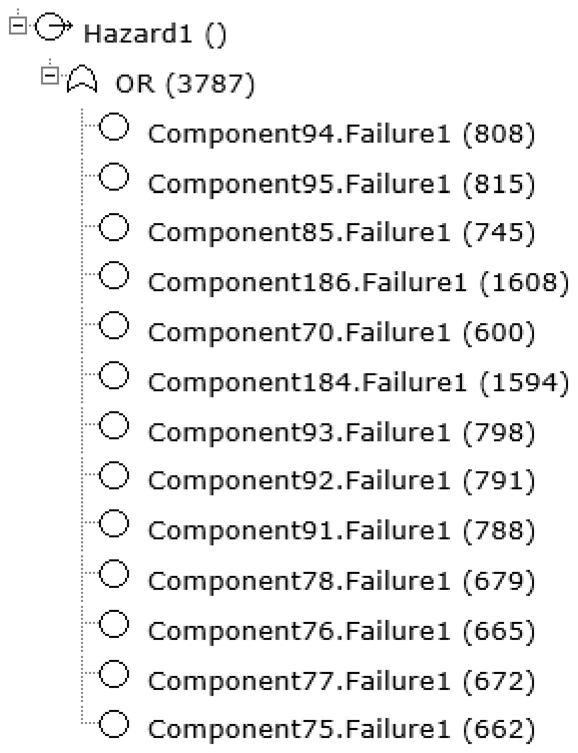
Figure 92 – HBSM3 fault trees: "Braking with wrong value in 4 wheels" hazard – part 3 of 3.

BSS fault trees

The BSS model was provided by an industrial partner and for confidentiality reasons the real names of its hazards, components and failure modes will not be shown. Generic names are used instead as follows: *HazardX*, *ComponentY* and *FailureZ*. "X" varies from 1 and the number of hazards identified for the BSS (10); "Y" varies from 1 and the number of components in the system (185); finally, "Z" ranges between 1 and the number of failure modes identified for a given component.

There are a total of 190 failure modes in the BSS. Only 185 are part of the ASIL allocation optimisation process because five failure modes are *safe*, i.e., they do not contribute to originating a system hazard.

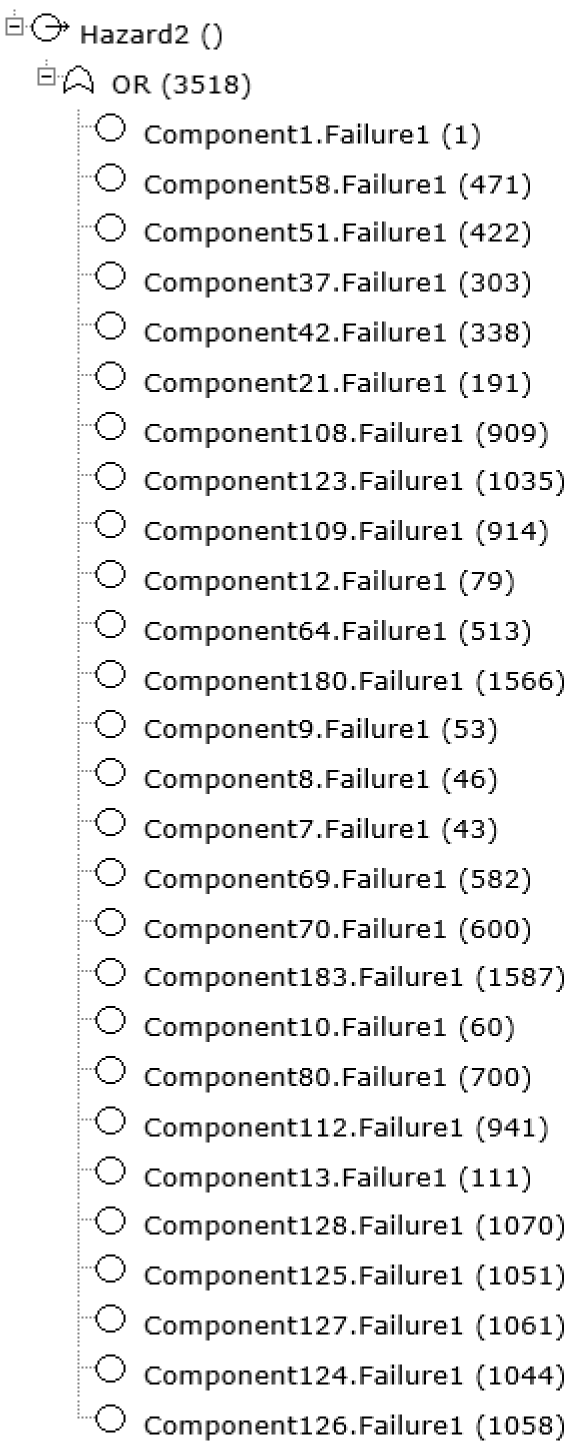
Hazard1



```
Hazard1 ()
  OR (3787)
    Component94.Failure1 (808)
    Component95.Failure1 (815)
    Component85.Failure1 (745)
    Component186.Failure1 (1608)
    Component70.Failure1 (600)
    Component184.Failure1 (1594)
    Component93.Failure1 (798)
    Component92.Failure1 (791)
    Component91.Failure1 (788)
    Component78.Failure1 (679)
    Component76.Failure1 (665)
    Component77.Failure1 (672)
    Component75.Failure1 (662)
```

Figure 93 - BSS fault trees: "Hazard1" hazard – part 1 of 1.

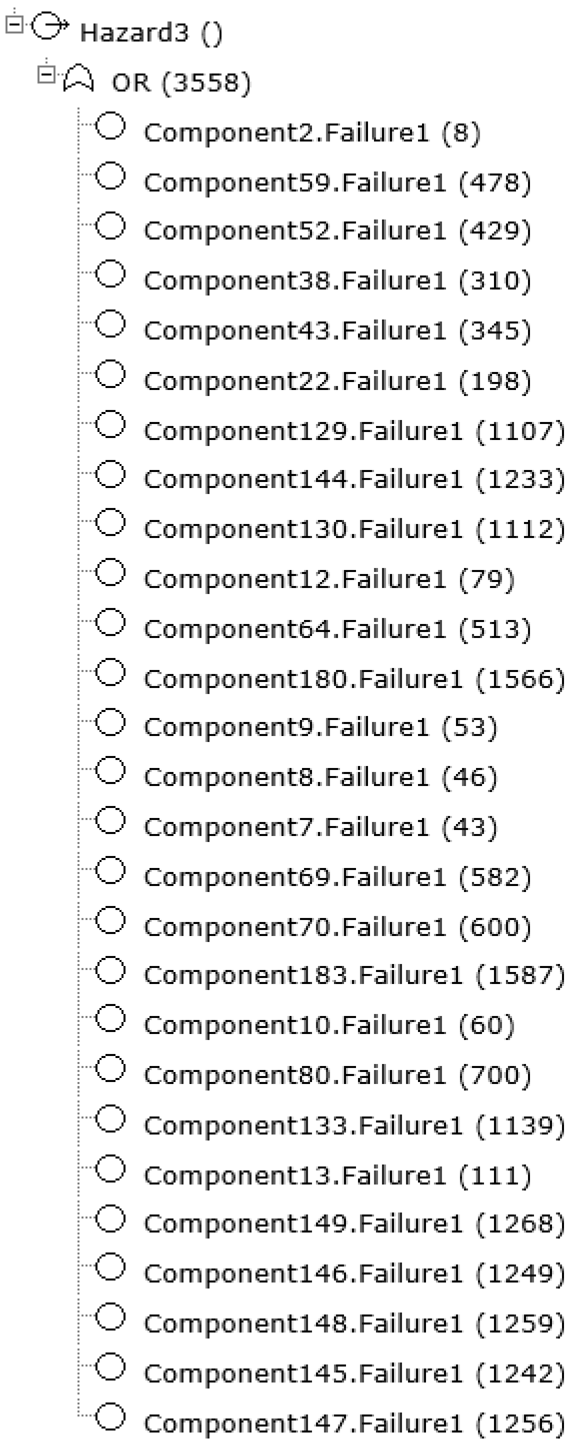
Hazard2



```
Hazard2 ()
  OR (3518)
    Component1.Failure1 (1)
    Component58.Failure1 (471)
    Component51.Failure1 (422)
    Component37.Failure1 (303)
    Component42.Failure1 (338)
    Component21.Failure1 (191)
    Component108.Failure1 (909)
    Component123.Failure1 (1035)
    Component109.Failure1 (914)
    Component12.Failure1 (79)
    Component64.Failure1 (513)
    Component180.Failure1 (1566)
    Component9.Failure1 (53)
    Component8.Failure1 (46)
    Component7.Failure1 (43)
    Component69.Failure1 (582)
    Component70.Failure1 (600)
    Component183.Failure1 (1587)
    Component10.Failure1 (60)
    Component80.Failure1 (700)
    Component112.Failure1 (941)
    Component13.Failure1 (111)
    Component128.Failure1 (1070)
    Component125.Failure1 (1051)
    Component127.Failure1 (1061)
    Component124.Failure1 (1044)
    Component126.Failure1 (1058)
```

Figure 94 - BSS fault trees: “Hazard2” hazard – part 1 of 1.

Hazard3



```
Hazard3 ()
  OR (3558)
    Component2.Failure1 (8)
    Component59.Failure1 (478)
    Component52.Failure1 (429)
    Component38.Failure1 (310)
    Component43.Failure1 (345)
    Component22.Failure1 (198)
    Component129.Failure1 (1107)
    Component144.Failure1 (1233)
    Component130.Failure1 (1112)
    Component12.Failure1 (79)
    Component64.Failure1 (513)
    Component180.Failure1 (1566)
    Component9.Failure1 (53)
    Component8.Failure1 (46)
    Component7.Failure1 (43)
    Component69.Failure1 (582)
    Component70.Failure1 (600)
    Component183.Failure1 (1587)
    Component10.Failure1 (60)
    Component80.Failure1 (700)
    Component133.Failure1 (1139)
    Component13.Failure1 (111)
    Component149.Failure1 (1268)
    Component146.Failure1 (1249)
    Component148.Failure1 (1259)
    Component145.Failure1 (1242)
    Component147.Failure1 (1256)
```

Figure 95 - BSS fault trees: “Hazard3” hazard – part 1 of 1.

Hazard4

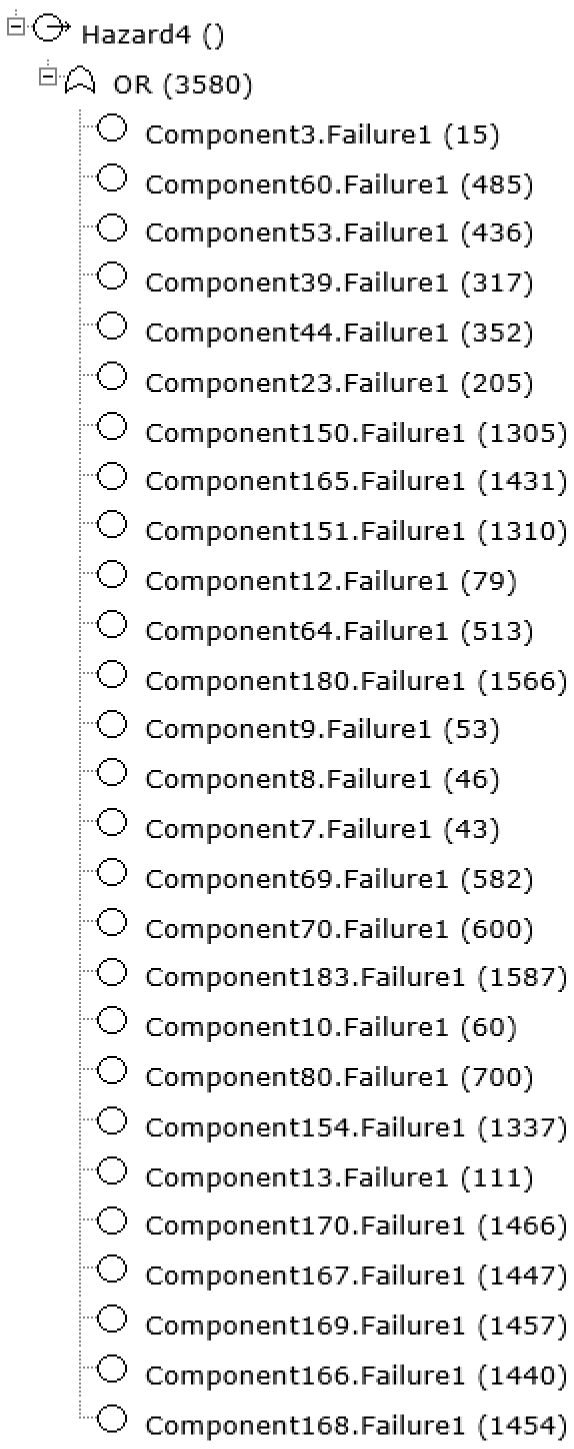
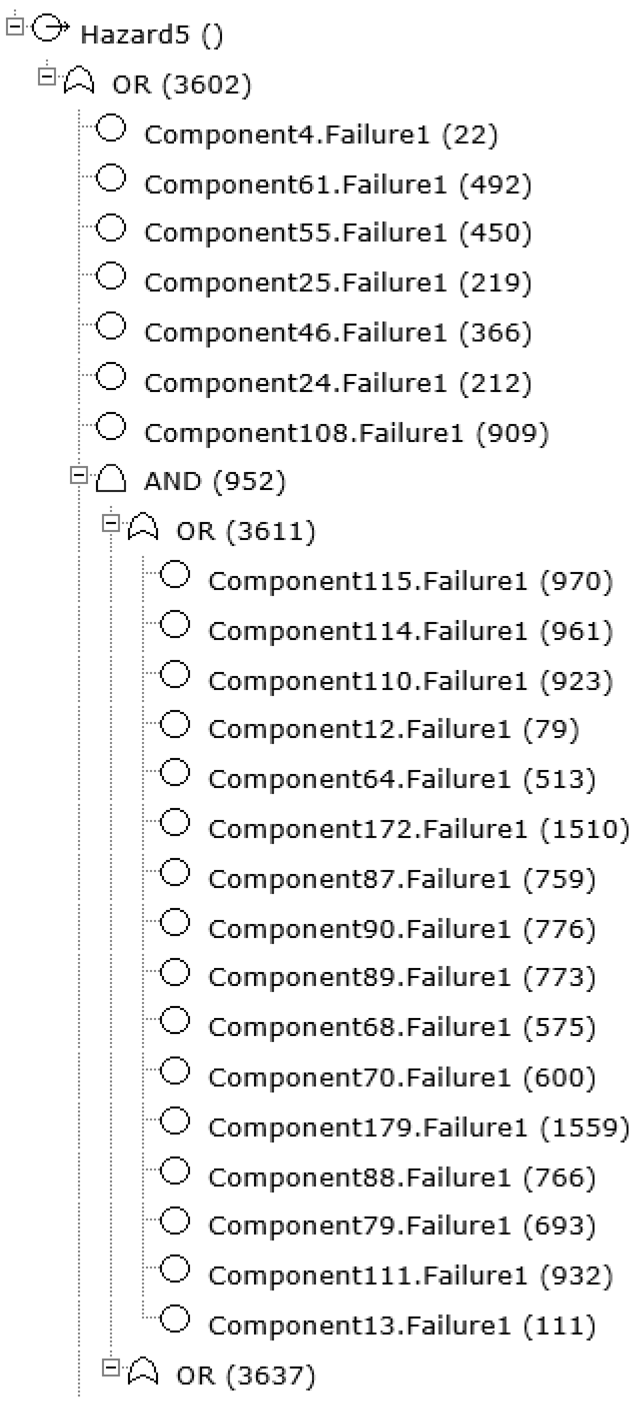

Hazard4 ()
OR (3580)
Component3.Failure1 (15)
Component60.Failure1 (485)
Component53.Failure1 (436)
Component39.Failure1 (317)
Component44.Failure1 (352)
Component23.Failure1 (205)
Component150.Failure1 (1305)
Component165.Failure1 (1431)
Component151.Failure1 (1310)
Component12.Failure1 (79)
Component64.Failure1 (513)
Component180.Failure1 (1566)
Component9.Failure1 (53)
Component8.Failure1 (46)
Component7.Failure1 (43)
Component69.Failure1 (582)
Component70.Failure1 (600)
Component183.Failure1 (1587)
Component10.Failure1 (60)
Component80.Failure1 (700)
Component154.Failure1 (1337)
Component13.Failure1 (111)
Component170.Failure1 (1466)
Component167.Failure1 (1447)
Component169.Failure1 (1457)
Component166.Failure1 (1440)
Component168.Failure1 (1454)

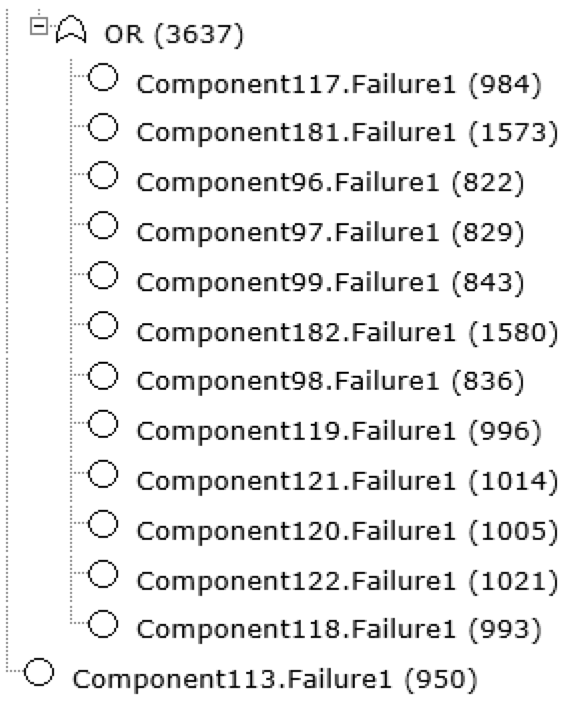
Figure 96 - BSS fault trees: “Hazard4” hazard – part 1 of 1.

Hazard5



```
Hazard5 ()
  OR (3602)
    Component4.Failure1 (22)
    Component61.Failure1 (492)
    Component55.Failure1 (450)
    Component25.Failure1 (219)
    Component46.Failure1 (366)
    Component24.Failure1 (212)
    Component108.Failure1 (909)
    AND (952)
      OR (3611)
        Component115.Failure1 (970)
        Component114.Failure1 (961)
        Component110.Failure1 (923)
        Component12.Failure1 (79)
        Component64.Failure1 (513)
        Component172.Failure1 (1510)
        Component87.Failure1 (759)
        Component90.Failure1 (776)
        Component89.Failure1 (773)
        Component68.Failure1 (575)
        Component70.Failure1 (600)
        Component179.Failure1 (1559)
        Component88.Failure1 (766)
        Component79.Failure1 (693)
        Component111.Failure1 (932)
        Component13.Failure1 (111)
      OR (3637)
```

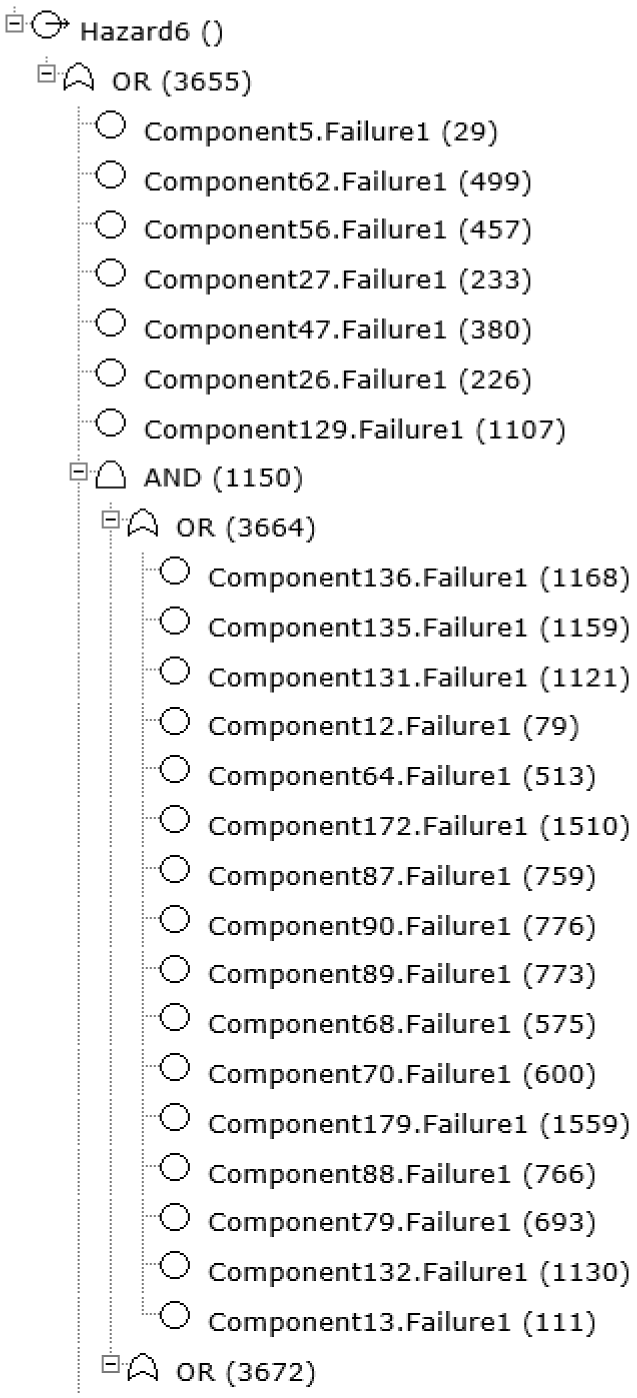
Figure 97 - BSS fault trees: “Hazard5” hazard – part 1 of 2.



```
OR (3637)
    Component117.Failure1 (984)
    Component181.Failure1 (1573)
    Component96.Failure1 (822)
    Component97.Failure1 (829)
    Component99.Failure1 (843)
    Component182.Failure1 (1580)
    Component98.Failure1 (836)
    Component119.Failure1 (996)
    Component121.Failure1 (1014)
    Component120.Failure1 (1005)
    Component122.Failure1 (1021)
    Component118.Failure1 (993)
Component113.Failure1 (950)
```

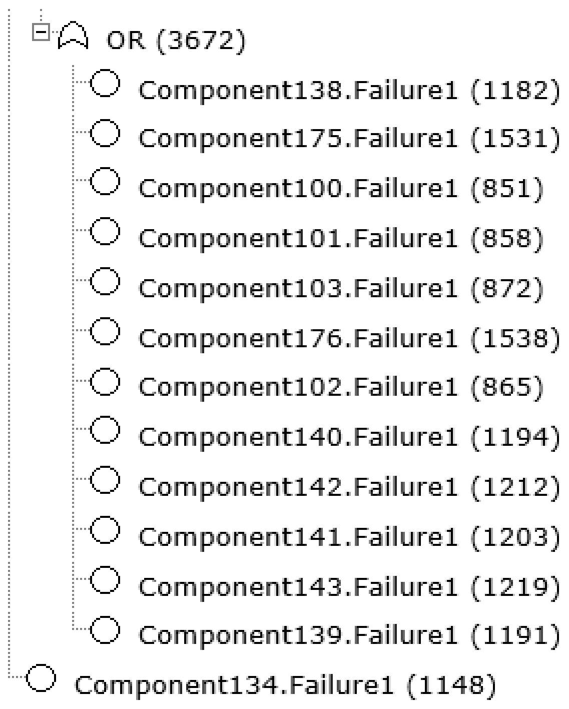
Figure 98 - BSS fault trees: “Hazard5” hazard – part 2 of 2.

Hazard6



```
Hazard6 ()
  OR (3655)
    Component5.Failure1 (29)
    Component62.Failure1 (499)
    Component56.Failure1 (457)
    Component27.Failure1 (233)
    Component47.Failure1 (380)
    Component26.Failure1 (226)
    Component129.Failure1 (1107)
    AND (1150)
      OR (3664)
        Component136.Failure1 (1168)
        Component135.Failure1 (1159)
        Component131.Failure1 (1121)
        Component12.Failure1 (79)
        Component64.Failure1 (513)
        Component172.Failure1 (1510)
        Component87.Failure1 (759)
        Component90.Failure1 (776)
        Component89.Failure1 (773)
        Component68.Failure1 (575)
        Component70.Failure1 (600)
        Component179.Failure1 (1559)
        Component88.Failure1 (766)
        Component79.Failure1 (693)
        Component132.Failure1 (1130)
        Component13.Failure1 (111)
      OR (3672)
```

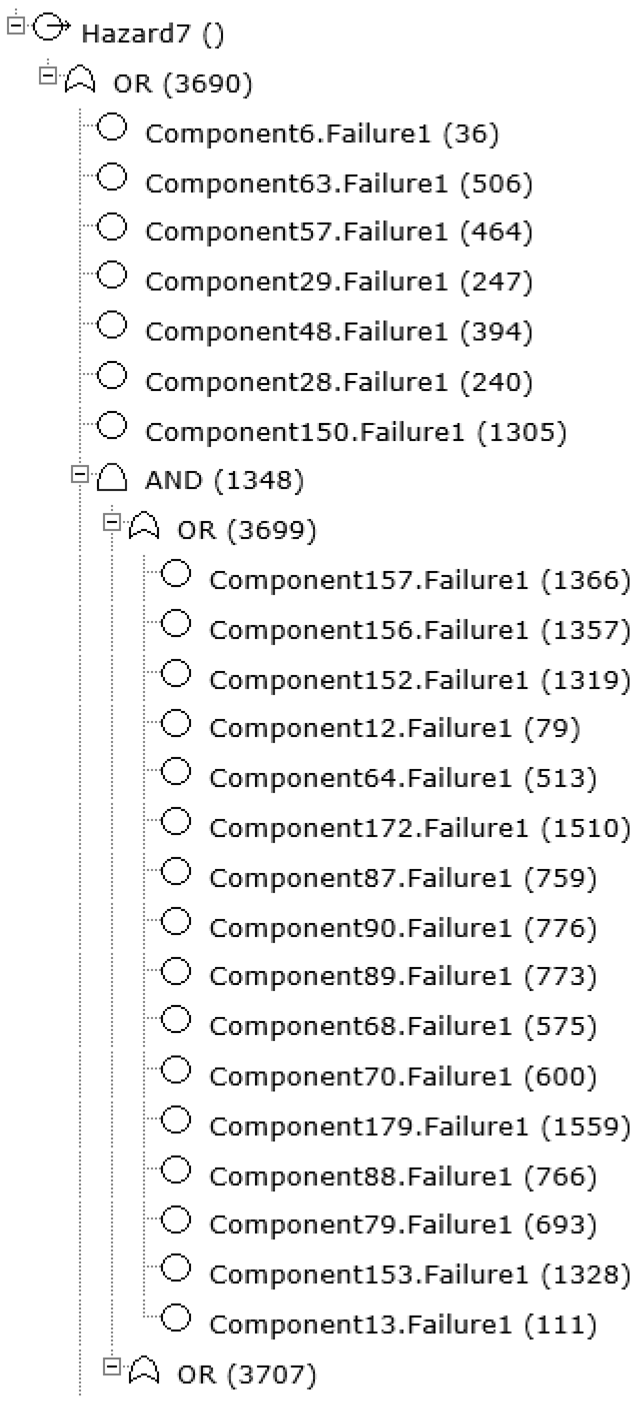
Figure 99 - BSS fault trees: “Hazard6” hazard – part 1 of 2.



```
OR (3672)
    Component138.Failure1 (1182)
    Component175.Failure1 (1531)
    Component100.Failure1 (851)
    Component101.Failure1 (858)
    Component103.Failure1 (872)
    Component176.Failure1 (1538)
    Component102.Failure1 (865)
    Component140.Failure1 (1194)
    Component142.Failure1 (1212)
    Component141.Failure1 (1203)
    Component143.Failure1 (1219)
    Component139.Failure1 (1191)
Component134.Failure1 (1148)
```

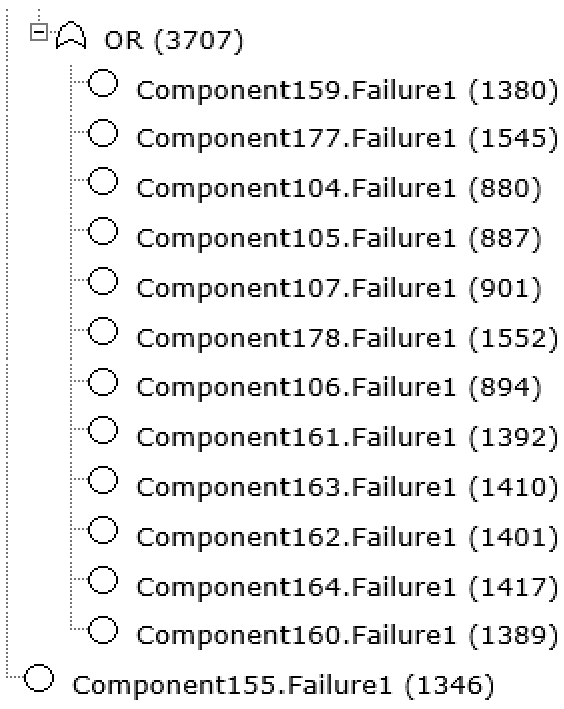
Figure 100 - BSS fault trees: “Hazard6” hazard – part 2 of 2.

Hazard7



```
Hazard7 ()
  OR (3690)
    Component6.Failure1 (36)
    Component63.Failure1 (506)
    Component57.Failure1 (464)
    Component29.Failure1 (247)
    Component48.Failure1 (394)
    Component28.Failure1 (240)
    Component150.Failure1 (1305)
    AND (1348)
      OR (3699)
        Component157.Failure1 (1366)
        Component156.Failure1 (1357)
        Component152.Failure1 (1319)
        Component12.Failure1 (79)
        Component64.Failure1 (513)
        Component172.Failure1 (1510)
        Component87.Failure1 (759)
        Component90.Failure1 (776)
        Component89.Failure1 (773)
        Component68.Failure1 (575)
        Component70.Failure1 (600)
        Component179.Failure1 (1559)
        Component88.Failure1 (766)
        Component79.Failure1 (693)
        Component153.Failure1 (1328)
        Component13.Failure1 (111)
      OR (3707)
```

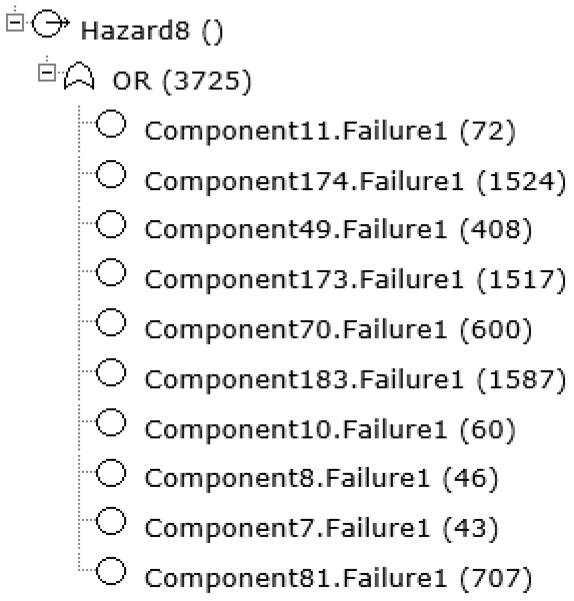
Figure 101 - BSS fault trees: “Hazard7” hazard – part 1 of 2.



```
OR (3707)
    Component159.Failure1 (1380)
    Component177.Failure1 (1545)
    Component104.Failure1 (880)
    Component105.Failure1 (887)
    Component107.Failure1 (901)
    Component178.Failure1 (1552)
    Component106.Failure1 (894)
    Component161.Failure1 (1392)
    Component163.Failure1 (1410)
    Component162.Failure1 (1401)
    Component164.Failure1 (1417)
    Component160.Failure1 (1389)
Component155.Failure1 (1346)
```

Figure 102 - BSS fault trees: “Hazard7” hazard – part 2 of 2.

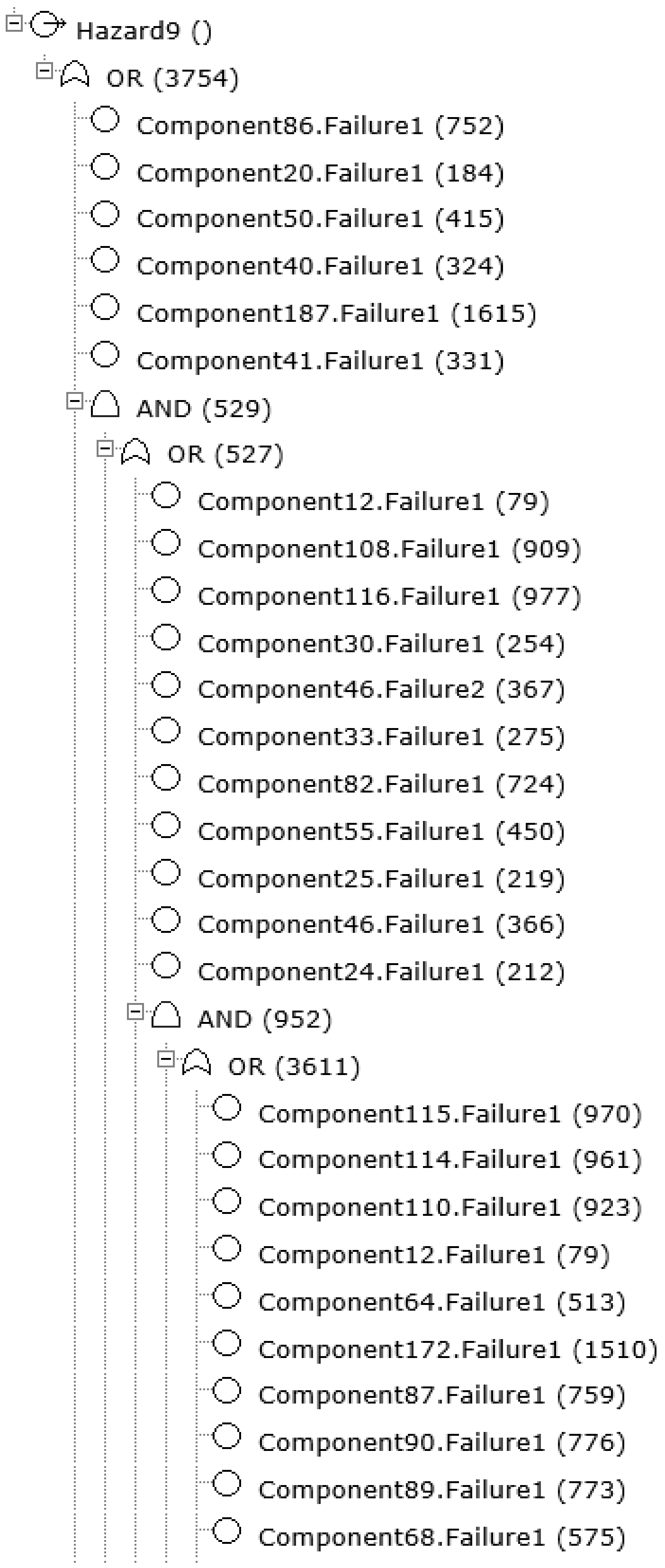
Hazard8



Hazard8 ()
OR (3725)
Component11.Failure1 (72)
Component174.Failure1 (1524)
Component49.Failure1 (408)
Component173.Failure1 (1517)
Component70.Failure1 (600)
Component183.Failure1 (1587)
Component10.Failure1 (60)
Component8.Failure1 (46)
Component7.Failure1 (43)
Component81.Failure1 (707)

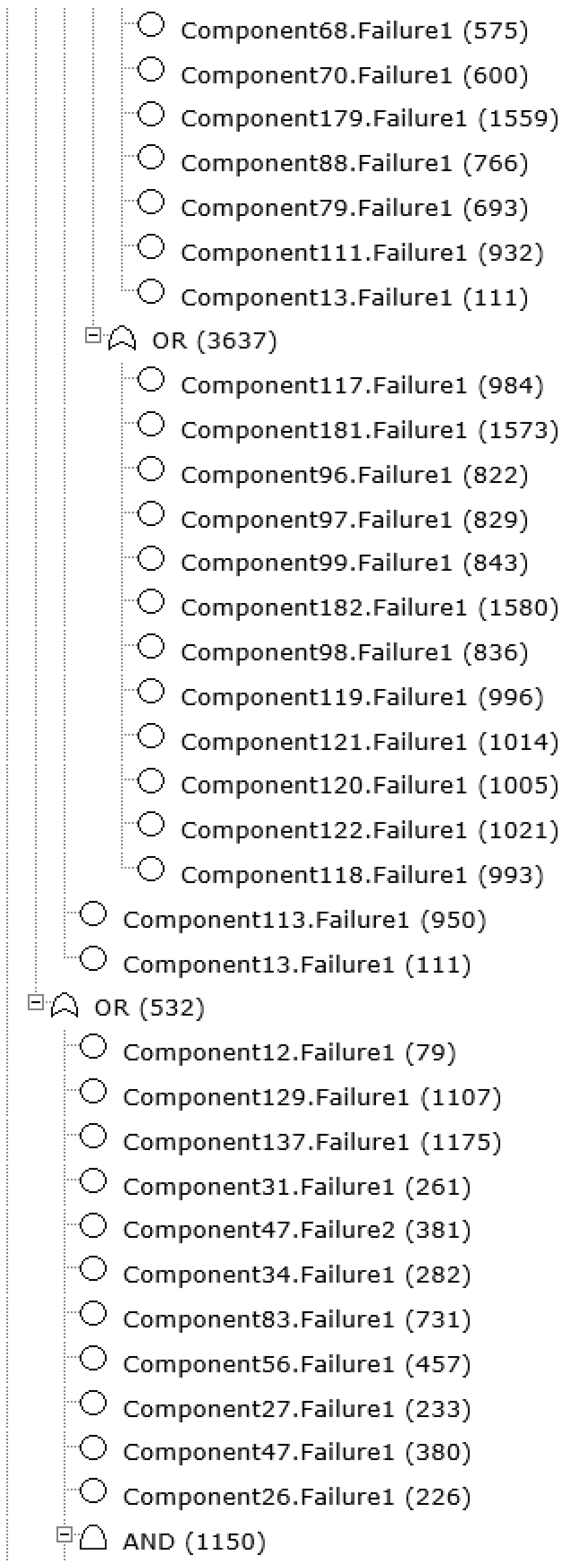
Figure 103 - BSS fault trees: “Hazard8” hazard – part 1 of 1.

Hazard9



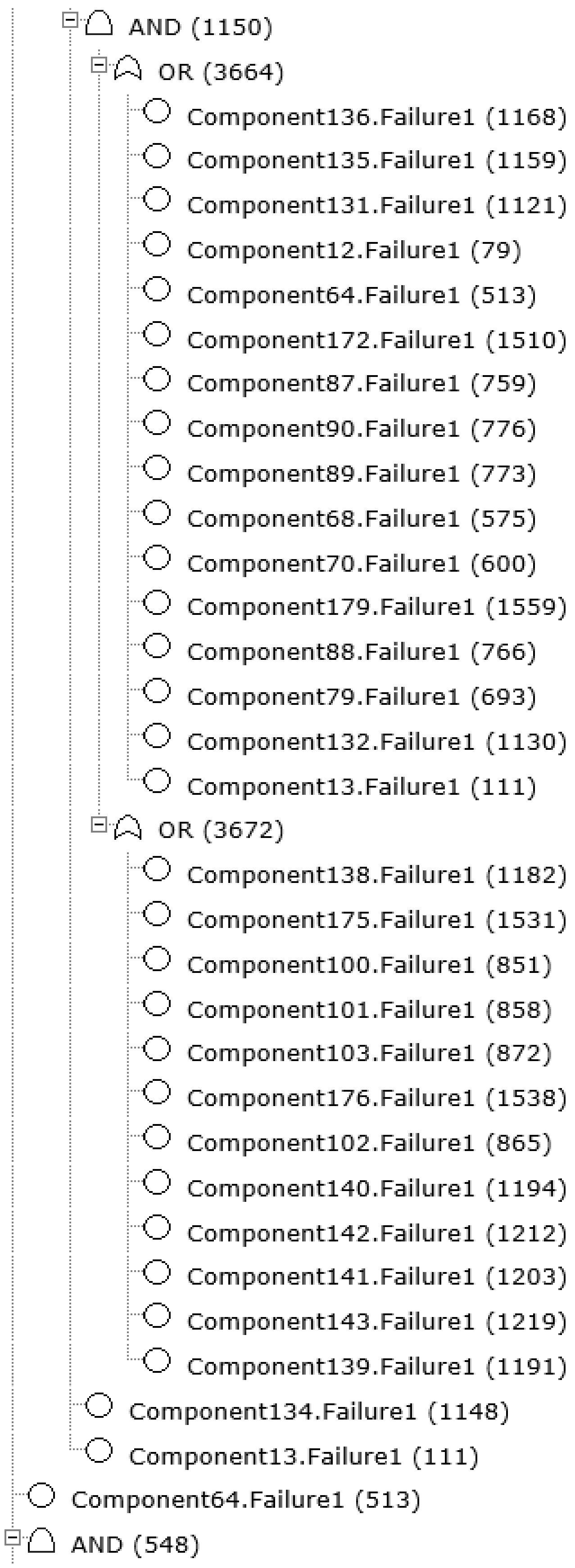
```
Hazard9 ()
  OR (3754)
    Component86.Failure1 (752)
    Component20.Failure1 (184)
    Component50.Failure1 (415)
    Component40.Failure1 (324)
    Component187.Failure1 (1615)
    Component41.Failure1 (331)
    AND (529)
      OR (527)
        Component12.Failure1 (79)
        Component108.Failure1 (909)
        Component116.Failure1 (977)
        Component30.Failure1 (254)
        Component46.Failure2 (367)
        Component33.Failure1 (275)
        Component82.Failure1 (724)
        Component55.Failure1 (450)
        Component25.Failure1 (219)
        Component46.Failure1 (366)
        Component24.Failure1 (212)
        AND (952)
          OR (3611)
            Component115.Failure1 (970)
            Component114.Failure1 (961)
            Component110.Failure1 (923)
            Component12.Failure1 (79)
            Component64.Failure1 (513)
            Component172.Failure1 (1510)
            Component87.Failure1 (759)
            Component90.Failure1 (776)
            Component89.Failure1 (773)
            Component68.Failure1 (575)
```

Figure 104 - BSS fault trees: "Hazard9" hazard – part 1 of 4.



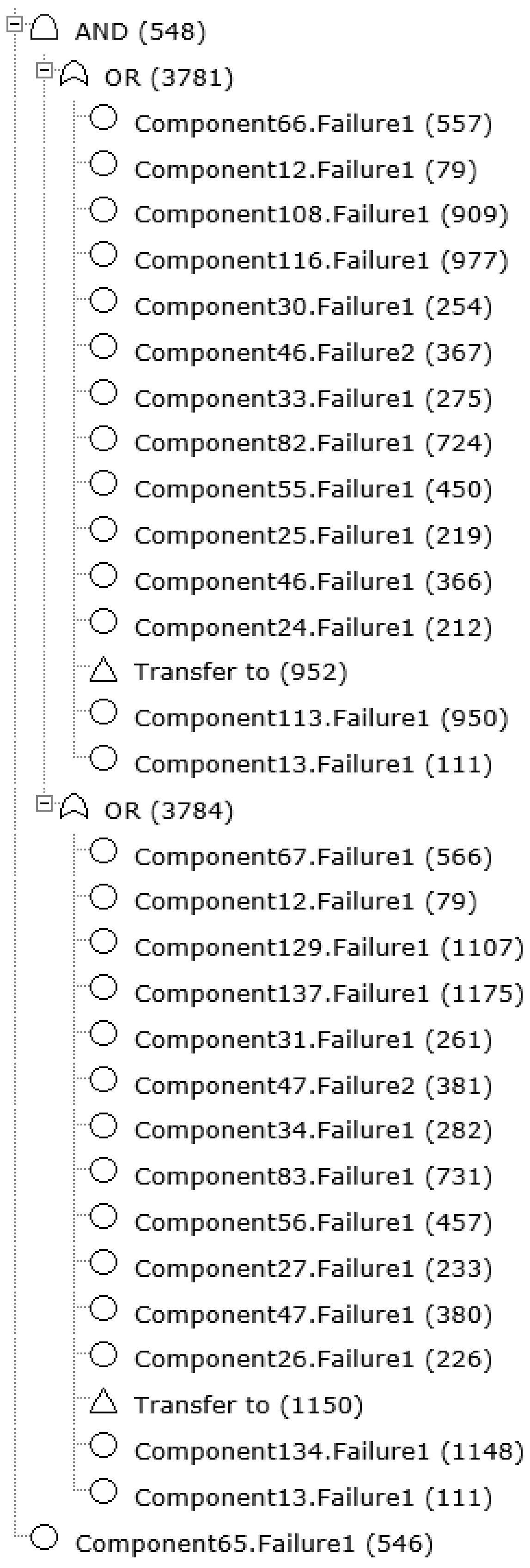
```
            ○ Component68.Failure1 (575)
            ○ Component70.Failure1 (600)
            ○ Component179.Failure1 (1559)
            ○ Component88.Failure1 (766)
            ○ Component79.Failure1 (693)
            ○ Component111.Failure1 (932)
            ○ Component13.Failure1 (111)
        ⊟ OR (3637)
            ○ Component117.Failure1 (984)
            ○ Component181.Failure1 (1573)
            ○ Component96.Failure1 (822)
            ○ Component97.Failure1 (829)
            ○ Component99.Failure1 (843)
            ○ Component182.Failure1 (1580)
            ○ Component98.Failure1 (836)
            ○ Component119.Failure1 (996)
            ○ Component121.Failure1 (1014)
            ○ Component120.Failure1 (1005)
            ○ Component122.Failure1 (1021)
            ○ Component118.Failure1 (993)
        ○ Component113.Failure1 (950)
        ○ Component13.Failure1 (111)
    ⊟ OR (532)
        ○ Component12.Failure1 (79)
        ○ Component129.Failure1 (1107)
        ○ Component137.Failure1 (1175)
        ○ Component31.Failure1 (261)
        ○ Component47.Failure2 (381)
        ○ Component34.Failure1 (282)
        ○ Component83.Failure1 (731)
        ○ Component56.Failure1 (457)
        ○ Component27.Failure1 (233)
        ○ Component47.Failure1 (380)
        ○ Component26.Failure1 (226)
        ⊟ AND (1150)
```

Figure 105 - BSS fault trees: "Hazard9" hazard – part 2 of 4.



```
  AND (1150)
    OR (3664)
      Component136.Failure1 (1168)
      Component135.Failure1 (1159)
      Component131.Failure1 (1121)
      Component12.Failure1 (79)
      Component64.Failure1 (513)
      Component172.Failure1 (1510)
      Component87.Failure1 (759)
      Component90.Failure1 (776)
      Component89.Failure1 (773)
      Component68.Failure1 (575)
      Component70.Failure1 (600)
      Component179.Failure1 (1559)
      Component88.Failure1 (766)
      Component79.Failure1 (693)
      Component132.Failure1 (1130)
      Component13.Failure1 (111)
    OR (3672)
      Component138.Failure1 (1182)
      Component175.Failure1 (1531)
      Component100.Failure1 (851)
      Component101.Failure1 (858)
      Component103.Failure1 (872)
      Component176.Failure1 (1538)
      Component102.Failure1 (865)
      Component140.Failure1 (1194)
      Component142.Failure1 (1212)
      Component141.Failure1 (1203)
      Component143.Failure1 (1219)
      Component139.Failure1 (1191)
    Component134.Failure1 (1148)
    Component13.Failure1 (111)
Component64.Failure1 (513)
AND (548)
```

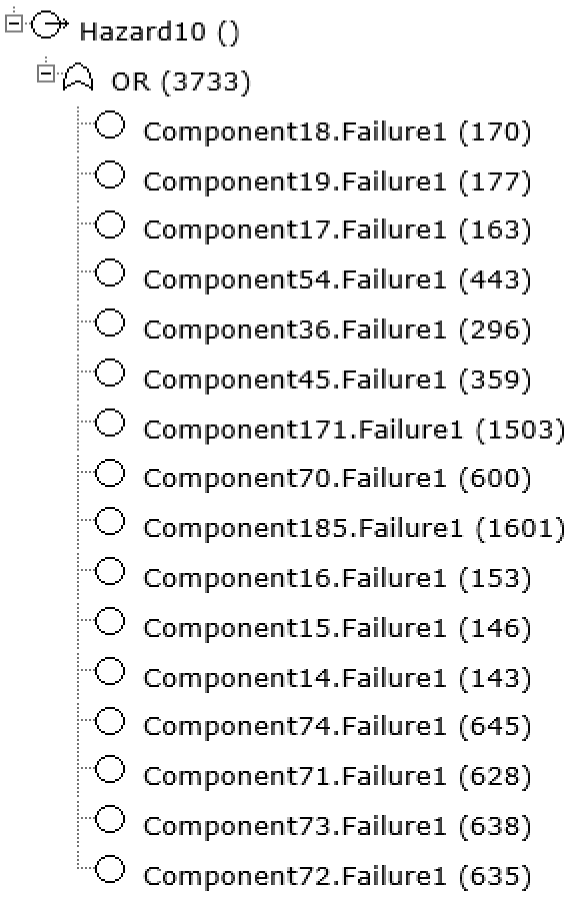
Figure 106 - BSS fault trees: “Hazard9” hazard – part 3 of 4.



```
AND (548)
    OR (3781)
        Component66.Failure1 (557)
        Component12.Failure1 (79)
        Component108.Failure1 (909)
        Component116.Failure1 (977)
        Component30.Failure1 (254)
        Component46.Failure2 (367)
        Component33.Failure1 (275)
        Component82.Failure1 (724)
        Component55.Failure1 (450)
        Component25.Failure1 (219)
        Component46.Failure1 (366)
        Component24.Failure1 (212)
        Transfer to (952)
        Component113.Failure1 (950)
        Component13.Failure1 (111)
    OR (3784)
        Component67.Failure1 (566)
        Component12.Failure1 (79)
        Component129.Failure1 (1107)
        Component137.Failure1 (1175)
        Component31.Failure1 (261)
        Component47.Failure2 (381)
        Component34.Failure1 (282)
        Component83.Failure1 (731)
        Component56.Failure1 (457)
        Component27.Failure1 (233)
        Component47.Failure1 (380)
        Component26.Failure1 (226)
        Transfer to (1150)
        Component134.Failure1 (1148)
        Component13.Failure1 (111)
    Component65.Failure1 (546)
```

Figure 107 - BSS fault trees: “Hazard9” hazard – part 4 of 4.

Hazard10



```
Hazard10 ()
  OR (3733)
    Component18.Failure1 (170)
    Component19.Failure1 (177)
    Component17.Failure1 (163)
    Component54.Failure1 (443)
    Component36.Failure1 (296)
    Component45.Failure1 (359)
    Component171.Failure1 (1503)
    Component70.Failure1 (600)
    Component185.Failure1 (1601)
    Component16.Failure1 (153)
    Component15.Failure1 (146)
    Component14.Failure1 (143)
    Component74.Failure1 (645)
    Component71.Failure1 (628)
    Component73.Failure1 (638)
    Component72.Failure1 (635)
```

Figure 108 - BSS fault trees: "Hazard10" hazard – part 1 of 1.